Spectrum-based Fault Localization in Embedded Software

Spectrum-based Fault Localization in Embedded Software

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus Prof.dr.ir. J.T. Fokkema,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen
op woensdag, 4 november 2009 om 12:30 uur door

Rui Filipe LIMA MARANHÃO de ABREU

Informatics Engineer - University of Minho
geboren te Fão, Portugal.

This work was carried out as part of the TRADER project under the responsibility of the Embedded Systems Institute. This project was partially supported by the Netherlands Ministry of Economic Affairs under the BSIK03021 program.

ISBN 978-90-79982-04-2

Typeset by the author with the LaTeX Documentation System.
Printed in the Netherlands by Wöhrmann Print Service.

Author email: `r.f.abreu@tudelft.nl`

In memoriam of my grandparents

Acknowledgements

I compare doing a Ph.D. to what the Portuguese sailors faced during the discoveries in the 16th century. Back then, they also had their setbacks, tempests, and yearning (probably the best translation for this very Portuguese feeling *saudade*), but the after-lull only made them believe that they could achieve their ultimate goal: discover unknown land. Although I am the commander-in-chief of *this man-o'-war*, I could not have weathered the storm without the help of my "crew". I would like to take this opportunity to acknowledge them.

First of all, I doubt I can properly express my gratitude to Prof.dr.ir. Arjan J.C. van Gemund. Arjan has been a great source of inspiration and I am very thankful for his continuous encouragement, support, patient, and enthusiasm.

My *man-o'-war* went plain sailing greatly due to dr.ir. Peter Zoeteweij. In his farewell dinner, Arjan said that Peter was a gentleman in science: I truly agree with him. It was a great pleasure to work with Peter on a daily basis. Thanks a million! I would also like to extend my gratitude to the M.Sc. students that worked under my supervision.

I would like to thank the members of my defense committee: prof.dr. A. van Deursen, prof.dr. C. Witteveen, prof.dr.ir. M. Akşit, prof.dr.ir. A.C. Brombacher, dr. J. de Kleer, and dr. W. Mayer for providing me with valuable feedback on this thesis. In particular, I would like to give a special thanks to dr. Mayer for the collaboration which resulted in Chapter 7, and dr. J. de Kleer for the many conversations which inspired the work of Chapter 5.

Many thanks to the members of the Trader project: Hasan, we still have to write that paper together! I would also like to acknowledge those, external to the TU Delft, that co-authored my papers: Rob Golsteijn, Markus Stumptner, and Wolfgang Mayer. Moreover, I acknowledge the (current and former) members of the Software Engineering Research Group (SERG) at TU Delft: Arie van Deursen, Andy Zaidman, Bas Cornelissen, Cathal Boogerd (we still have to publish a joint paper; thanks for helping with the "Samenvatting"), Marius Marin, Martin Pinzger, Gerd Gross, Ali Mesbah, Eelco Visser and his fellas, Leon Moonen, Alberto González, Michaela Greiler, Eric Piel, Hans Geers, Kees Pronk, Peter Kluit, Rini van Solingen, Frans Ververs (your retirement was a big loss for the group's social life), Teemu Kanstren, Eric Bouwers, Alex Feldman (aye man, the Northern crossing... what a trip!), and Paulo Anita. Finally, within TU Delft, I also wish to express my gratitude to Theodoros Zoumpoulidis: thanks for being such a great friend and put up with me for the past four years.

Concerning the non-academic side of my life in the Netherlands, I have to thank a great number of people for their friendship and the many enjoyable moments we spent together. To avoid failing to remember somebody, their names will not be brought to light (I guess you all know who you are anyway). I would also take this opportunity to thank all my friends back home. First of all, to "O Bando", a group of 13 apostles (and their respective ones) that met for the first time several years ago in Braga. Based on our experiences together, I am sure we are no fair-weather friends. Guys, thanks for making my trips back to Portugal as pleasant as they could possibly be: "E as gajas a cantar, e as gajas a cantar! Olé *Bando* olé...". Second, I would like to show my appreciation to my dear friend Ricardo Gomes: thanks for cheer me up in the many MSN conversations we had. Finally, I would also like to state my appreciation to all those that visited me in the Netherlands (I hope you came over for visiting me and not for what the Dutch law tolerates...).

A word (in Portuguese) of gratitude to my family: mommy Maria de Lurdes, pappy Bernardino, sister Bela, brother-in-law Rui, and nephew (in fact, my little bro) Rui Pedro. Este e certamente um dos projectos mais importantes da minha vida, e sem o vosso incondicional apoio e confiança nas minhas capacidades esta tese nunca teria sido possível. As minhas desculpas por me ter ausentado por tanto tempo.

Last, but certainly not least, I would like to thank my girlfriend, and above all my best friend, Liliana. I managed to finish this thesis greatly due to your support and comprehension. Your flair to motivate me, viz. in the not-so-good moments, was of great importance for me to finish this journey. Thank you, my love!

Delft,
October 7, 2009

Rui Abreu

Contents

List of Symbols

$\mathcal{P}$	-	Program under test
A	-	Program Activity Matrix ($M \times N$)
a_{ij}	-	Value of $A[i,j]$
b	-	Bloom Filter
C	-	Fault cardinality
$\mathcal{C}$	-	Set of components in program $\mathcal{P}$
c_j	-	Component number j, $c_j \in \mathcal{C}$
D	-	Diagnostic report (list of diagnosis candidates)
d_k	-	Diagnosis candidate
e	-	Error vector, with N positions
f	-	Probability a run behaves as expected
f_p	-	False positive rate
f_n	-	False negative rate
g_j	-	Goodness parameter for component j
$g(d_k)$	-	Goodness parameter for diagnosis d_k
$\mathcal{H}$	-	Heuristic function for guiding minimal hitting set computation
h_j	-	Health variable of component j
I	-	List of inspected components
M	-	Number of components in program $\mathcal{P}$
N	-	Number of runs/transactions
N_F	-	Number of failed runs/transactions
N_P	-	Number of passed runs/transactions
$n_{pq}(j)$	-	Number of times component j is involved ($p = 1$) or not ($p = 0$) in passed ($q = 0$) or failed ($q = 1$) runs
$\Pr(d_k)$	-	Probability diagnosis d_k being the true explanation
q_d	-	Quality of the diagnosis, aka Score, effort
q_e	-	Quality of the error detection
r	-	Probability a component is executed
$s(j)$	-	Similarity coefficient of component c_j with error vector
$\mathcal{T}$	-	Set of test cases
$\mathcal{T}_{\mathcal{F}}$	-	Set of test cases that fail
$\mathcal{T}_{\mathcal{P}}$	-	Set of test cases that pass
t_n	-	True negative rate
t_p	-	True positive rate
y	-	Hash function
W	-	Wasted testing/debugging effort (diagnostic quality metric)

List of Acronyms

AAAI	-	Association for the Advancement of Artificial Intelligence
ACM	-	Association for Computer Machinery
Barinel	-	Bayesian approach to diagnose intermittent faults
CBI	-	Cooperative Bug Isolation
CPU	-	Central processing unit
DD	-	Delta Debugging
DDD	-	Data Display Debugger
GCC	-	GNU Compiler Collection
Gcov	-	GNU Profiler
GDB	-	GNU Debugger
IEEE	-	Institute of Electrical and Electronics Engineers
J2EE	-	Java 2 Platform, Enterprise Edition
KLOC	-	Kilo lines of code
LCD	-	Liquid crystal display
LOC	-	Lines of code
MBD	-	Model-based diagnosis
MBSD	-	Model-based software debugging
MIPS	-	Microprocessor without Interlocked Pipeline Stages
MLOC	-	Mega lines of code
NN	-	Nearest Neighbor
NVM	-	Non-volatile memory
RAM	-	Read access memory
SFL	-	Spectrum-based fault localization
Staccato	-	Statistics-directed minimal hiting set algorithm
TRADER	-	TV Related Architecture to Design and Enhance Reliability

1 Introduction

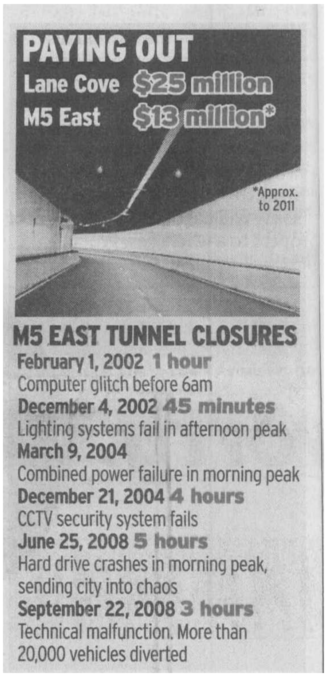
"There has never been an unexpectedly short software debugging period in the history of computers."

– Steven Levy

Modern daily devices such as televisions rely increasingly on (embedded) software. Features implemented in software are often cheaper, easier, flexible to future modifications, and more portable than when implemented in hardware. Such properties are extremely important as, nowadays, many devices serve no single purpose but, instead, have several functionalities which need to be easily modified or upgraded to adhere to the high expectations of the consumers. As an example, a mobile phone is used not only to make phone calls but also, e.g., as a navigation system which has to contain the most up-to-date navigation engine and maps.

Not only are more and more features implemented in software, but software is also developed by several in/off-shore teams or outsourced. As a consequence, software complexity increases drastically. Due to this complexity, software systems are difficult to maintain and typically have a high defect density which decreases the quality (i.e., correctness) of the system [Carey et al., 1999]. Defects in the system may lead to unintended behavior or even critical system failures. This problem is especially serious for embedded software, as such systems (1) often have to meet additional, non-functional requirements (resource-constrained systems), and (2) are generally mission-critical (e.g., embedded software running on an artificial pacemaker [Halperin et al., 2002]). Next to the well-known disasters caused by software defects [Garfinkel, 2005], the author's personal experience includes the effects of a critical computer glitch which led to a 3-hour shut down of the M5 East highway's tunnel in New South Wales, Australia during peak hours in September 2008. Due to a software defect, the component responsible for managing the tunnel's fire and air circulation could not be controlled reliably. The malfunction caused many vehicles to be diverted (see Figure 1.1). At the time of writing, the glitch's root cause is still unknown.

In addition to the increasing software complexity, increased market competitiveness leads to restricted systems testing, in an attempt to further reduce time-to-market. As exhaustive testing of complex systems is prohibitively expensive, testing would not reveal all faults in the software anyway. Consequently, software is shipped with residual defects. While some of the defects


PAYING OUT

Lane Cove $25 million

M5 East $13 million*

*Approx. to 2011

M5 EAST TUNNEL CLOSURES

February 1, 2002 **1 hour**
Computer glitch before 6am

December 4, 2002 **45 minutes**
Lighting systems fail in afternoon peak

March 9, 2004
Combined power failure in morning peak

December 21, 2004 **4 hours**
CCTV security system fails

June 25, 2008 **5 hours**
Hard drive crashes in morning peak, sending city into chaos

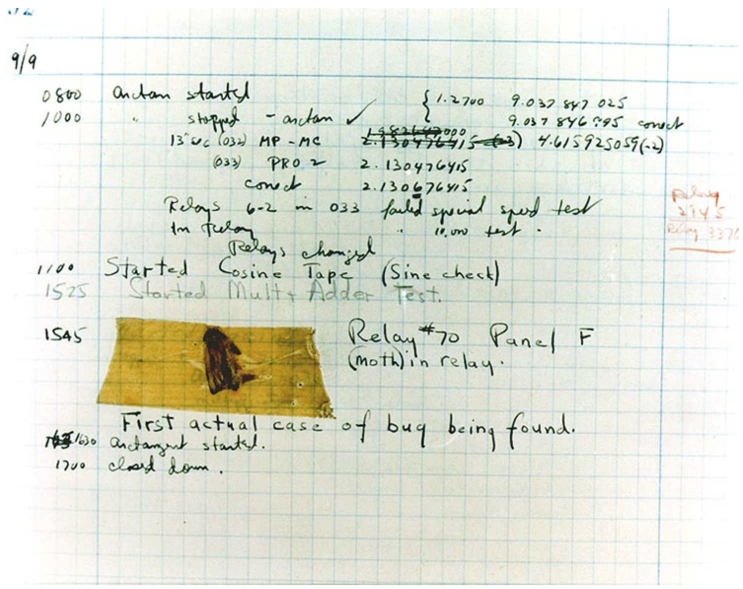
September 22, 2008 **3 hours**
Technical malfunction. More than 20,000 vehicles diverted

Figure 1.1 Computer glitch closes M5 highway during peak hour [source: The Sydney Morning Herald, September 23rd, 2008]

are either tolerated or never perceived by the users [de Visser, 2008, Keijzers et al., 2008], others may cause the system to critically fail, possibly entailing extremely serious financial or life-threatening consequences. Amongst the high-profile examples of the drastic financial consequences a defect can cause is the malfunction of the control software of Ariane 5, which caused the rocket to disintegrate 37 seconds after launch [Lions, 1996, Dowson, 1997]. An example of life-threatening consequences is the crash of a British Royal Air Force Chinook helicopter in 1994, killing 29 service men. Although initially dismissed as a pilot error, an investigation uncovered sufficient evidence that the accident was caused by a software defect in the helicopter's engine control computer [Rogerson, 2002].

When unexpected behavior is observed, developers need to identify the root cause(s) that makes the system deviate from its intended behavior. This task (also known as software debugging, fault localization, or fault diagnosis[1]) is the most time-intensive and expensive phase of the software development cycle [Hailpern and Santhanam, 2002], and is being performed since the beginning of computer history (see Figure 1.2 for an account of the first reported bug in computer history).

[1]In this thesis, the terms software debugging, fault diagnosis, and fault localization are used interchangeably.



```
9/9

0800  arctan started
1000    "    stopped - arctan ✓      { 1.2700   9.037 847 025
                                     {          9.037 846 795 conect
      13" uc (032) MP - MC    ~~1.982147000~~
                              ~~2.130476415~~ ~~(-3)~~) 4.615925059(-2)
            (033) PRO 2       2.130476415
                 conect       2.130676415
        Relays 6-2 in 033 failed special speed test
        in relay                  "   10,000 test .
                Relays changed
1100  Started Cosine Tape (Sine check)
1525  Started Mult + Adder Test.

1545                         Relay #70 Panel F
                             (moth) in relay.

           First actual case of bug being found.
1630  arctangent started.
1700  closed down .
```

Relay
2145
Relay 3370

Figure 1.2 First reported bug in computer history, 1947: Operators traced an error in the Harvard Mark II computer to a moth trapped in a relay, coining the term bug.

A traditional approach to fault localization is to insert *print* statements in the program to cause the program to generate additional debugging information to help identifying the root cause of the observed failure. Essentially, the developer adds these statements to the program to get a glimpse of the runtime state, variable values, or to verify that the program has reached a particular program point. Another common technique is the use of a symbolic debugger which supports additional features such as breakpoints, single stepping, and state modifying. Examples of symbolic debuggers are GDB [Stallman, 1994], DBX [DBX, 1990], DDD [Zeller and Lütkehaus, 1996], EXDAMS [Balzer, 1969], and the debugger proposed by Agrawal, Demillo, and Spaord [Agrawal et al., 1991]. Symbolic debuggers are included in many integrated development environments (IDE) such as Eclipse[2], Microsoft Visual Studio[3], Xcode[4], and Delphi[5].

These traditional, manual fault localization approaches have a number of important limitations. The placement of print statements as well as the inspection of their output are unstructured and ad-hoc, and are typically based on the developer's intuition. In addition, developers tend to use only test cases that reveal the failure, and therefore do not use valuable information from passing test cases. Furthermore, the size of the program state at each point

[2]http://www.eclipse.org
[3]http://msdn.microsoft.com/en-us/vstudio/default.aspx
[4]http://developer.apple.com/tools/xcode/
[5]http://www.codegear.com

can be large, and there are many combinations of program executions that have to be examined. Hence, such techniques still require a detailed knowledge of the program, and also suffer from a substantial execution overhead in terms of execution time and space to store historical run-time data. Last, but not least, manual debugging is extremely expensive in terms of labor cost. As an indication of the downtime, debugging, and repair costs involved, a 2002 landmark study indicated that software bugs pose an annual $60 billion cost to the US economy alone [RTI, 2002].

Aimed at drastic cost reduction, much research has been performed in developing automatic software debugging techniques and tools. This thesis aims to contribute to advancing the state-of-the-art in automatic fault localization. Most of the work presented in this thesis was carried out in the context of the TRADER project [Trader, 2009], involving several Dutch universities, Philips Tass, IMEC, and NXP Semiconductors. The project was conducted under the responsibility of the Embedded Systems Institute (ESI) in Eindhoven, the Netherlands. The main goal of the TRADER project was to develop methods and techniques for ensuring reliability of consumer electronic products, minimizing product failures that are exposed to the end user.

1.1 CONCEPTS AND DEFINITIONS

Throughout this thesis, we use the following terminology [Avižienis et al., 2004].

- A *failure* is an event that occurs when delivered service deviates from correct service.
- An *error* is a system state that may cause a failure.
- A *fault* (defect/bug) is the cause of an error in the system.

In this thesis, we apply this terminology to computer programs, where faults are bugs in the program code. Typically, these programs transform input data into output data in a single run, and a failure occurs when the output for a given input differs from the expected output for that input. Failures and errors are *symptoms* caused by faults in the program.

To illustrate these concepts, consider the simple C function in Figure 1.3. It is meant to sort, using the bubble sort algorithm, a sequence of `n` rational numbers whose numerators and denominators are stored in the parameters `num` and `den`, respectively. There is a fault (bug) in the swapping code within the body of the `if` statement. Only the numerators of the rational numbers are swapped while the denominators are left in their original order. In this case, a failure occurs when `RationalSort` changes the contents of its argument arrays in such a way that the result is not a sorted version of the original. An error may occur after the code inside the conditional statement is executed, while `den[j]` ≠ `den[j+1]`. Such errors can be temporary, and do not

```
void RationalSort(int n, int *num, int *den){
   int i,j,temp;
   for (i=n-1; i>=0; i--){
      for (j=0; j<i; j++){
         if (RationalGT(num[j],den[j],num[j+1],den[j+1])){
            /* Bug: forgot to swap denominators */
            temp = num[j];
            num[j] = num[j+1];
            num[j+1] = temp;
         }
      }
   }
}
```

Figure 1.3 A faulty C function for sorting rational numbers

automatically lead to failures. For example, if we apply `RationalSort` to the sequence $\langle \frac{4}{1}, \frac{2}{2}, \frac{0}{1} \rangle$, an error occurs after the first two numerators are swapped. However, this error is "canceled" by later swapping actions, and the sequence ends up being sorted correctly. Note that faults do not automatically lead to errors either: no error will occur if the sequence is already sorted, or if all denominators are equal.

A program under analysis comprises a set of M components (such as statements, basic blocks, functions) c_j where $j \in \{1, \dots, M\}$, and can have multiple faults, the number being denoted C (fault cardinality). A *diagnostic candidate* is a set of component indices that may explain observed failures. A *diagnostic report* $D = < \dots, d_k, \dots >$ is a ranked set of diagnostic candidates d_k ordered in terms of likelihood to be the true diagnosis.

Error detection is a prerequisite for triggering a fault localization technique. One must know that a symptom has occurred before trying to locate the responsible fault. Program failures constitute a rudimentary form of error detection, as many errors remain latent and never lead to a failure. An example of a technique that increases the number of errors that can be detected is program instrumentation with invariants such as checks on null pointers and array bounds checking [Jones and Kelly, 1997].

1.2 FAULT LOCALIZATION

The process of pinpointing the fault(s) that led to symptoms (failures/errors) is called fault localization, and has been an active area of research for the past decades. Based on a set of observations, automatic approaches to software fault localization yield a list of likely fault locations, which is subsequently used either by the developer to focus the software debugging process, or as an input to automatic recovery mechanisms [Patterson et al., 2002, Sözer, 2009]. Depending on the amount of knowledge that is required about the system's internal component structure and behavior, the most predominant approaches can be classified as (1) statistical approaches or (2) reasoning approaches. The former approach uses an abstraction of program traces, dynamically collected at runtime, to produce a list of likely candidates to be at fault, whereas the latter combines a static *model* of the expected behavior with a set of observations to compute the diagnostic report.

Statistical approaches yield a diagnostic report with the M components ordered in terms of statistical evidence of being faulty (e.g., $< \{3\}, \{1\}, \ldots >$, in terms of the indices j of the components c_j). Reasoning approaches yield a diagnostic report that comprise candidates d_k (possibly multiple-fault) that explain the observations, ordered in terms of probability (e.g., $< \{4\}, \{1,3\}, \ldots >$, meaning that either component c_4 is at fault, or components c_1 *and* c_3 are at fault, etc.). For brevity, we will often refer to diagnostic candidates as diagnoses as well, as it is clear from the context whether we refer to a single diagnosis candidate or to the entire diagnosis.

1.2.1 *Statistical Approaches*

Statistics-based fault localization techniques use an abstraction of program traces, also known as *program spectra*, to find a statistical relationship with observed failures. Program spectra are collected at run-time, during the execution of the program, and many different forms exist [Harrold et al., 2000]. For example, component-hit spectra indicate whether a component was involved in the execution of the program or not. In contrast to model-based approaches, program spectra and pass/fail information are the only *dynamic* source of information used by statistics-based techniques.

Well-known examples of such approaches are the Tarantula tool by Jones, Harrold, and Stasko [Jones et al., 2002], the Nearest Neighbor technique by Renieris and Reiss [Renieris and Reiss, 2003], the Sober tool by Lui, Yan, Fei, Han, and Midkiff [Liu et al., 2006], the work of Liu and Hand [Liu and Han, 2006], PPDG by Baah, Podgurski, and Harrold [Baah et al., 2008], CrossTab by Wong, Wei, Qi, and Zap [Wong et al., 2008], the Cooperative Bug Isolation by Liblit and his colleagues [Liblit et al., 2005, Liblit, 2008, Nainar et al., 2007, Zheng et al., 2006], the Pinpoint tool by Cheng and his colleagues [Chen et al., 2002], the AMPLE tool by Dallmeier, Lindig, and Zeller [Dallmeier et al., 2005], the work by Steimann, Eichstädt-Engelen, and Schaaf [Steimann et al., 2008], and the Time Will Tell approach by Yilmaz, Paradkar, and Williams [Yilmaz et al., 2008]. Although differing in the way they derive the statistical fault ranking, all techniques are based on measuring program spectra, and will be analyzed in great detail later.

1.2.2 *Reasoning Approaches*

Reasoning approaches to fault localization use prior knowledge of the system, such as required component behavior and interconnection, to build a model of the correct behavior of the system. An example of a reasoning technique is model-based diagnosis (see, e.g., [de Kleer and Williams, 1987]), where a diagnosis is obtained by logical inference from the *static* model of the system, combined with a set of run-time observations. In the software engineering community this approach is often called model-based software debugging [Mayer and Stumptner, 2008]. Well-known approaches to model-based software debugging include the approaches of Friedrich, Stumptner, and Wotawa [Friedrich et al., 1999, Friedrich et al., 1996], Nica and Wotawa [Nica and Wotawa, 2008], Wotawa, Stumptner, and Mayer [Wotawa et al., 2002], and Mayer and Stumpter [Mayer and Stumptner, 2008], all of which are described in more detail later.

Other techniques that use prior information about the system being diagnosed are approaches based on model checkers. Such techniques automatically verify whether a model of the system meets a given specification or not. For instance, they verify whether a software system can reach a critical state that will cause it to fail, or check whether the software is deadlock or race condition-free. Examples of such approaches include Groce's Δ-slicing [Groce et al., 2006] and `explain` [Groce, 2004], the JavaPathFinder work of Visser and his colleagues [Visser et al., 2003, Visser and Mehlitz, 2005], the CEGAR approach of Sharygina *et al.* [Sharygina et al., 2009], the SPIN tool by Holzmann and his colleagues [Holzmann, 1997], and the work of Dolby, Vaziri, and Tip [Dolby et al., 2007].

1.3 PROBLEM STATEMENT

As explained in the previous section, statistics-based approaches take as their only input dynamic information collected at run-time, using no prior knowledge of the system under analysis. Such approaches have therefore intrinsically no modeling costs attached. Amongst the best statistical approaches in terms of diagnostic cost/performance ratio are those based on the computation of a statistical similarity between component activity and failure behavior using a so-called similarity coefficient s [Jones and Harrold, 2005, Liu et al., 2006, Wong et al., 2008], which we refer to as spectrum-based fault localization (SFL). Component activity is recorded as program spectra (collected in a matrix A, explained in detail in Chapter 2), and information on whether each of the program spectra in A corresponds to a passed of failed execution is collected in a vector e. The diagnostic process can be explained as

$$(A, e) \xrightarrow{s} D$$

SFL is a light-weight approach since for each component only a similarity coefficient (scalar operation) has to be computed, after which the M components in the system are sorted into D in order of likelihood to be at fault. Besides, dynamically collecting (A, e) requires only marginal overhead.

Given these properties, SFL is particularly interesting because (1) it has the potential to scale well to very large code bases, (2) it may be applicable to resource-constrained environments, and (3) it is transparent to the development process (entailing no extra effort, assuming a test suite is available). While (A, e) appears to be a good basis for localizing software faults, the diagnostic accuracy of SFL is inherently limited since no reasoning is applied. For example, SFL creates a ranking containing *all* M components, and not only with the subset of components that logically explain observed failures (reasoning would discard invalid candidates). Moreover, SFL considers only single faults. However, software bug density is high, and therefore in real software programs the probability of multiple faults is very high. Hence, approaches to multiple-fault localization are of great importance.

In contrast to SFL, reasoning approaches not only reason in terms of multiple faults, but also have a strong theoretical foundation in terms of the logical theories that haven been proposed in the past years, e.g., [de Kleer and Williams, 1987, De Kleer et al., 1992, Struss and Dressler, 1989, Console and Torasso, 1991]. Exploiting the information present in the model, the diagnostic performance of reasoning approaches is higher than SFL. However, current state-of-the-art techniques have several limitations that hinder their application to large, industrial software systems. Manually creating a model of the intended behavior may be as complex and/or error-prone as building the system itself. Hence, it is extremely difficult to guarantee that the model is correct and, moreover, that it is consistent with the ever evolving software system. To overcome these shortcomings, approaches to automatically derive a model from software system have been proposed (e.g. [Mayer, 2007]). Unfortunately, these approaches are prohibitively complex because (1) they are dependent on computationally expensive static analysis techniques and (2) it is difficult to abstract from the program code (e.g., the expression level). Consequently, the complexity of the models as well as the underlying diagnostic algorithms prohibit the application of reasoning to anything but toy programs of a few hundred lines of code.

With the aim to capture the best of both worlds, in this thesis we study a spectrum-based reasoning approach to fault localization. By abstracting from program topology and dependencies, the (A, e)-based modeling allows the reasoning to be relatively cheap, while benefiting from the increased accuracy due to reasoning. In particular, the main research question addressed in this thesis is as follows:

> *What is the inherent performance limitation of (traditional) SFL, and what are the benefits and costs of applying a spectrum-based* reasoning *approach?*

Apart from this main research question, this thesis also addresses the following, peripheral questions:

- Traditionally, SFL takes as input error information e from test oracles (e.g., during testing phase, a specification of expected behavior is used to determine whether a test case fails or not). However, at the operational phase, test oracles are typically not available. Aimed at total automation of fault localization, can simple, generic program invariants replace test oracles for pass/fail input to SFL?

- Reasoning techniques typically generate an excessive number of diagnosis candidates, most of which are highly improbable. Can an SFL-based heuristic function be used to focus the search to highly probable diagnosis candidates, rendering reasoning approaches amenable to large software systems?
- Model-based software debugging suffers from being extremely complex, and returns to the user a diagnostic report without ranking information. Consequently, all components in the report need to be inspected. Can SFL be integrated with model-based software debugging (1) to focus the search of the latter, reducing its high time complexity, and (2) to improve its diagnostic quality?

1.4 CONTRIBUTIONS

Overall, this thesis makes six contributions. The first three contributions are related to SFL, and the last three are based on combining SFL with reasoning:

1. We perform a thorough study on SFL in order to clearly understand its fundamental limitations. In particular, we study the diagnostic accuracy as a function of (1) similarity coefficient, (2) quantity of observations, and (3) quality of the error detectors. We present a new similarity coefficient that consistently outperforms all coefficients investigated, independent of the experimental environment.
2. Owing to its small time and space complexity, SFL is amenable to resource-constrained software systems, such as embedded software. We report our experiences in applying this fault localization approach to the control software which serves as the basis for televisions sets manufactured by NXP Semiconductor's customers [NXP, 2009].
3. We show that fault screeners (error detectors originating from the hardware domain) have the capabilities to replace test oracles with respect to the diagnostic performance of SFL, at limited overhead, enabling a fully automated approach to software fault localization.
4. As a central contribution, we present a low-cost, Bayesian reasoning approach to spectrum-based multiple fault localization. We show that the Bayesian reasoning approach clearly outperforms SFL, at marginal increase of complexity.
5. Computing the set of valid diagnosis candidates from program spectra includes computing the minimal hitting set which is a NP-hard problem. We present an SFL-based heuristic to focus the search of minimal hitting

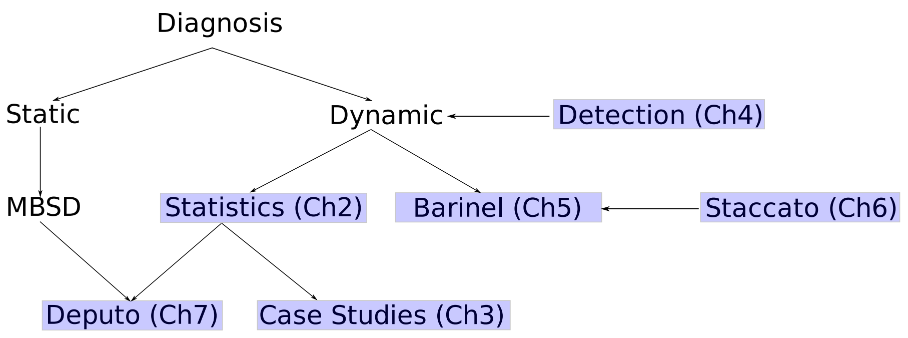


Figure 1.4 Overview

sets, decreasing the time complexity by orders of magnitude while capturing all relevant solutions.

6. Finally, we combine SFL with model-based software debugging to render the latter amenable to larger programs, as well as to rank its diagnostic report. We show that the combination of semantics-based analysis as undertaken in model-based diagnosis and the dynamic aspects obtained from program execution spectra yield better diagnostic reports.

1.5 THESIS OUTLINE

The six contributions outlined in the previous section are described in terms of six chapters, respectively. Chapter 2 describes the thorough study performed on SFL. In Chapter 3 we report our findings on applying SFL to large, industrial case studies. Chapter 4 studies the usage of fault screeners for automatic error detection. Chapter 5 presents a low-cost, (Bayesian) reasoning approach to spectrum-based multiple fault localization, coined Barinel. Chapter 6 describes a low-cost, heuristic approach to generate the set of valid diagnosis candidates, coined Staccato. Chapter 7 describes an approach to combine statistics-based fault localization with model-based software debugging (MBSD), coined Deputo, to focus the search of MBSD. Figure 1.4 depicts the relationships between the various topics chapters as well as the main research topic of each chapter. Each core chapter in this thesis is directly based on at least two peer-reviewed publications. Finally, in Chapter 8 we draw conclusions and present recommendations for future work.

1.6 ORIGIN OF CHAPTERS

Most of the publications have been co-authored with Zoeteweij and Van Gemund. The publications of Chapter 3 have been co-authored with Zoeteweij, Golsteijn, and Van Gemund. The publication of Chapter 6 has been co-authored with Van Gemund only. The publication of Chapter 7 has been co-authored with Mayer, Stumptner, and Van Gemund. The following list gives an overview of these publications:

Chapter 2 has been published in *Journal of Systems & Software (JSS)*, 2009 [Abreu et al., 2009c]. An earlier version of the chapter appeared in the *Proceedings of the IEEE Testing: Academic and Industrial Conference - Practice and Research Techniques (TAIC PART'07)* [Abreu et al., 2007].

Chapter 3 has been published in *Journal of Systems and Software (JSS)*, 2009 [Abreu et al., 2009c], and in the *Proceedings of the International Conference on the Engineering of Computer Based Systems (ECBS'07)* [Zoeteweij et al., 2007];

Chapter 4 has been published in *Lecture Notes in Communications in Computer and Information Science (LNCCIS)*, 2009 [Abreu et al., 2009a]. An earlier version of this work appeared in *Proceedings of the ACM Symposium on Applied Computing (SAC'08)* [Abreu et al., 2008a], and in the *Proceedings of the International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE'08)* [Abreu et al., 2008b].

Chapter 5 has been published in the *Proceedings of the AAAI International Joint Conference on Artificial Intelligence (IJCAI'09)* [Abreu et al., 2009d], and has also been accepted for publication in the *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE'09)* [Abreu et al., 2009e]. An earlier version of this work appeared in the *Proceedings of the ACM Workshop on Dynamic Analysis (WODA'08)* [Abreu et al., 2008d];

Chapter 6 has been published in the *Proceedings of the Symposium on Abstraction, Reformulation and Approximation (SARA'09)* [Abreu and van Gemund, 2009a]. An earlier version of this work appeared in the *Proceedings of the International Workshop on Principles of Diagnosis (DX'09)* [Abreu and van Gemund, 2009b];

Chapter 7 has been published in the *Proceedings of the ACM Symposium on Applied Computing (SAC'09)* [Abreu et al., 2009b]. An earlier version was published in the *Proceedings of the International Workshop on Principles of Diagnosis (DX'08)* [Mayer et al., 2008].

Spectrum-based Fault Localization

2

ABSTRACT

Spectrum-based fault localization (SFL) shortens the test-diagnose-repair cycle by reducing the debugging effort. As a light-weight automated diagnosis technique it can easily be integrated with existing testing schemes. Since SFL is based on discovering statistical coincidences between system failures and the activity of the different parts of a system, its diagnostic accuracy is inherently limited. Using a common benchmark consisting of the Siemens set and the `space` program, we investigate this diagnostic accuracy as a function of several parameters (such as quality and quantity of the program spectra collected during the execution of the system), some of which directly relate to test design. Our results indicate that the superior performance the Ochiai similarity coefficient, taken from the molecular biology domain and introduced by us in the context of fault localization, is largely independent of test design. Furthermore, near-maximal diagnostic accuracy (exonerating over 80% of the blocks of code on average) is already obtained for low-quality error observations and limited numbers of test cases. The influence of the number of test cases is of primary importance for continuous (embedded) processing applications, where only limited observation horizons can be maintained.

//

Testing, debugging, and verification represent a major expenditure in the software development cycle [Hailpern and Santhanam, 2002], which to a large extent is due to the labor-intensive task of diagnosing the faults (bugs) that cause tests to fail. Because under typical market conditions, only those faults that affect the user most can be solved before the release deadline, the efficiency with which faults can be diagnosed and repaired directly influences software reliability. Automated diagnosis can help to improve this efficiency.

Diagnosis techniques are complementary to testing in two ways. First, for tests designed to verify correct behavior, they generate information on the root cause of test failures, focusing the subsequent tests that are required to expose this root cause. Second, for tests designed to expose specific potential root causes, the extra information generated by diagnosis techniques can help to further reduce the set of remaining possible explanations. Given its incremental nature (i.e., taking into account the results of an entire sequence of tests), automated diagnosis alleviates much of the work of selecting tests in the latter category, and can hence have a profound impact on the test-diagnose-repair cycle.

An important part of diagnosis and repair consist in localizing faults, and several tools for automated debugging and systems diagnosis implement spectrum-based fault localization (SFL), an approach to diagnosis based on an analysis of the differences in *program spectra* [Harrold et al., 2000, Reps et al., 1997] for *passed* and *failed* runs. Passed runs are executions of a program that completed correctly, whereas failed runs are executions in which an error was detected. A program spectrum is an execution profile that indicates which parts of a program are active during a run. Spectrum-based fault localization entails identifying the part of the program whose activity correlates most with the detection of errors. Examples of tools that implement this approach are Pinpoint [Chen et al., 2002], which focuses on large, dynamic on-line transaction processing systems, Tarantula [Jones et al., 2002], whose implementation focuses on the analysis of C programs, and AMPLE [Dallmeier et al., 2005], which focuses on object-oriented software (see Section 2.6 for a discussion).

Spectrum-based fault localization does not rely on a model of the system under investigation. It can easily be integrated with existing testing procedures, and because of the relatively small overhead with respect to CPU time and memory requirements, it lends itself well for application within resource-constrained environments [Zoeteweij et al., 2007]. However, the efficiency of SFL comes at the cost of a limited *diagnostic accuracy*. As an indication, in one of the experiments described in Section 2.4, on average 20% of a program still needs to be inspected after the diagnosis due to a low number of failed runs.

In SFL, a *similarity coefficient* is used to rank potential fault locations. In earlier work [Abreu et al., 2006a], we obtained preliminary evidence that the Ochiai similarity coefficient, known from the biology domain (see, e.g., [da Silva Meyer et al., 2004]), can improve diagnostic accuracy over eight other coefficients, including those used by the Pinpoint and Tarantula tools mentioned above. Extending as well as generalizing this previous result, in this chapter we investigate the main factors that influence the accuracy of SFL in a much wider setting. Apart from the influence of the similarity coefficient on the diagnostic accuracy, we also study the influence of the quality and quantity of the (pass/fail) observations used in the analysis.

Quality of the observations relates to the classification of runs as passed or failed. Since most faults lead to errors only under specific input conditions, and as not all errors propagate to system failures, this parameter is relevant because error detection mechanisms are usually not ideal. Quantity of the observations relates to the number of passed and failed runs available for the diagnosis. If fault localization has to be performed at run-time, e.g., as a part of a recovery mechanism, one cannot wait to accumulate many observations to diagnose a potentially disastrous error until sufficient confidence is obtained. In addition, quality and quantity of the observations both relate to test coverage. Varying the observation context with respect to these two observational parameters allows a much more thorough investigation of the influence of similarity coefficients. Our study is based on a widely-used set of benchmark faults (single faults) consisting of the Siemens set [Hutchins et al., 1994] and the `space` program, both of which are available from the Software-artifact Infrastructure Repository [Do et al., 2005].

The main contributions of this chapter are the following.

- We show that the Ochiai similarity coefficient consistently outperforms the other coefficients mentioned above. This can be attributed to the Ochiai coefficient being more sensitive to activity in passed runs than to activity in failed runs of potential fault locations, which is well suited to software fault diagnosis because execution of faulty code does not necessarily lead to failures, while failures always involve a fault.
- We establish this result across the entire quality space, and for varying numbers of runs involved. Furthermore, we show that near-optimal diagnostic accuracy (exonerating over 80% of all code on average) is already obtained for low-quality (ambiguous) error observations, while, in addition, only a few runs are required. In particular, maximum diagnostic performance is already reached at 6 failed runs on average. However, including up to 20 passed runs may improve but also degrade diagnostic performance, depending on the program and/or input data.

The remainder of this chapter is organized as follows. In Section 2.1 we introduce some basic concepts and terminology, and explain the diagnosis technique in more detail. In Section 2.2 we describe our experimental setup. In Sections 2.3, 2.4, and 2.5 we describe the experiments on the similarity coefficient, and the quality and quantity of the observations, respectively. Related work is discussed in Section 2.6. We summarize this chapter in Section 2.7.

2.1 PRELIMINARIES

In this section we introduce program spectra, and describe how they are used in spectrum-based fault localization.

2.1.1 *Program Spectra*

A program spectrum [Reps et al., 1997] is a collection of data that provides a specific view on the dynamic behavior of software. This data is collected at run-time, and typically consist of a number of counters or flags for the different parts of a program. As such, recording a program spectrum is a light-weight analysis compared to other run-time methods, such as, e.g., dynamic slicing [Korel and Laski, 1988]. Many different forms of program spectra exist, see Table 2.1 and [Harrold et al., 2000] for a comprehensive overview. Although we work with so-called block-hit spectra, the approach studied in this paper easily generalizes to other types of program spectra (e.g., path-hit spectra, data-dependence-hit spectra).

Form	Description
Statement-hit	statements that were executed
Statement-count	number of times a statement was executed
Block-hit	conditional branches executed
Block-count	number of times a conditional branch was executed
Path-hit	path executed
Path-count	number of times each path was executed
Complete-path	complete path that was executed
Data-dependence-hit	definition-use pairs executed
Data-dependence-count	number of times a definition-use pair was executed
Output	output that was produced
Execution trace	execution trace produced
Time Spectra	execution time of, e.g., functions

Table 2.1 A catalog of program spectra

```
void RationalSort(int n, int *num, int *den){
   /* block 1 */
   int i,j,temp;

   for ( i=n-1; i>=0; i-- ) {
      /* block 2 */
      for ( j=0; j<i; j++ ) {
         /* block 3 */
         if (RationalGT(num[j], den[j],
                        num[j+1], den[j+1])) {
            /* block 4 */
            /* Bug: forgot to swap denominators */
            temp = num[j];
            num[j] = num[j+1];
            num[j+1] = temp;
         }
      }
   }
}
```

Figure 2.1 A faulty C function for sorting rational numbers

A block hit spectrum contains a flag for every block of code in a program, that indicates whether or not that block was executed in a particular run. With a block of code we mean a C language statement, where we do not distinguish between the individual statements of a compound statement, but where we do distinguish between the cases of a switch statement[1]. As an illustration, we have identified the blocks of code in Figure 2.1. Suppose that the function `RationalSort` of Figure 2.1 is used to sort the sequence $\langle \frac{2}{1}, \frac{3}{1}, \frac{4}{1}, \frac{1}{1} \rangle$, which it happens to do correctly. This would result in the block count spectrum

[1]This is a different notion from a *basic block*, which is a block of code that has no branch.

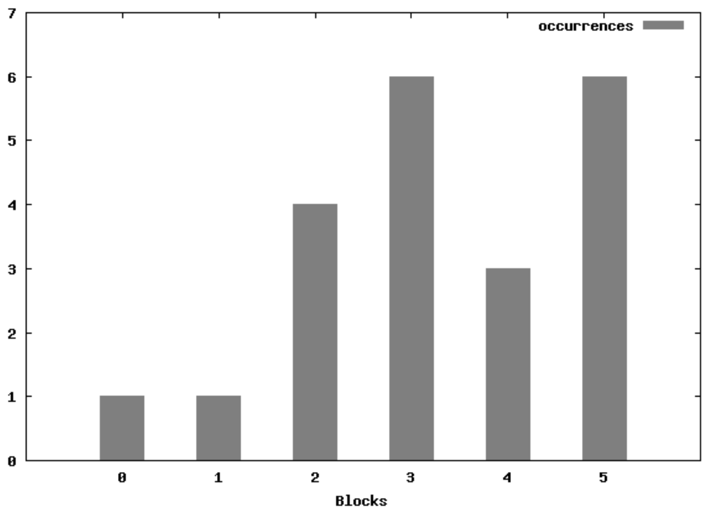


Figure 2.2 Block count spectrum

represented by the histogram in Figure 2.2, where block 5 refers to the body of the `RationalGT` function, which has not been shown in Figure 2.1. Block 1, the body of the function `RationalSort`, is executed once. Blocks 2 and 3, the bodies of the two loops, are executed four and six times, respectively. To sort our example array, three exchanges must be made, and block 4, the body of the conditional statement, is executed three times. Block 5, the `RationalGT` function body, is executed six times: once for every iteration of the inner loop.

If we are only interested in whether a block is executed or not, we can use binary flags instead of counters. In this case, the block count spectra revert to block *hit* spectra.

2.1.2 *Spectrum-based Fault Localization*

The hit spectra of N runs constitute a NxM binary matrix A, whose columns correspond to M different parts (blocks in our case) of a program (see Figure 2.3). The information in which runs an error was detected constitutes another column vector, the error vector e. This vector can be thought to represent a hypothetical part of the program that is responsible for all observed errors. Spectrum-based fault localization essentially consists in identifying the part whose column vector resembles the error vector most.

In the field of data clustering, resemblances between vectors of binary, nominally scaled data, such as the columns in our matrix of program spectra, are quantified by means of *similarity coefficients* (see, e.g., [Jain and Dubes, 1988]). Many similarity coefficients exist. As an example, below are two different similarity coefficients, namely the Jaccard coefficient s_J, which is used by the

$$\begin{array}{ccc}
 & M \text{ parts} & \text{errors} \\
N \text{ spectra} & \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1N} \\ a_{21} & a_{22} & \dots & a_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ a_{M1} & a_{M2} & \dots & a_{MN} \end{bmatrix} & \begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_M \end{bmatrix} \\
 & \begin{matrix} s(1) & s(2) & \dots & s(N) \end{matrix} &
\end{array}$$

Figure 2.3 The ingredients of spectrum-based fault localization

Pinpoint tool [Chen et al., 2002], the coefficient s_T, used in the Tarantula fault localization tool [Jones and Harrold, 2005]

$$s_J(j) = \frac{n_{11}(j)}{n_{11}(j) + n_{01}(j) + n_{10}(j)} \tag{2.1}$$

$$s_T(j) = \frac{\frac{n_{11}(j)}{n_{11}(j)+n_{01}(j)}}{\frac{n_{11}(j)}{n_{11}(j)+n_{01}(j)} + \frac{n_{10}(j)}{n_{10}(j)+n_{00}(j)}} \tag{2.2}$$

where $n_{11}(j)$ is the number of failed runs in which part j is involved, $n_{10}(j)$ is the number of passed runs in which part j is involved, $n_{01}(j)$ is the number of failed runs in which part j is not involved, and $n_{00}(j)$ is the number of passed runs in which part j is not involved, i.e., referring to Figure 2.3,

$$\begin{aligned}
n_{00}(j) &= |\{i \mid a_{ij} = 0 \wedge e_i = 0\}| \\
n_{01}(j) &= |\{i \mid a_{ij} = 0 \wedge e_i = 1\}| \\
n_{10}(j) &= |\{i \mid a_{ij} = 1 \wedge e_i = 0\}| \\
n_{11}(j) &= |\{i \mid a_{ij} = 1 \wedge e_i = 1\}|
\end{aligned}$$

In addition, we introduce the Ochiai coefficient s_O, used in the molecular biology domain [da Silva Meyer et al., 2004]

$$s_O(j) = \frac{n_{11}(j)}{\sqrt{(n_{11}(j) + n_{01}(j)) \cdot (n_{11}(j) + n_{10}(j))}} \tag{2.3}$$

Note that $n_{10}(j) + n_{11}(j)$ equals the number of runs in which part j is involved, and that $n_{10}(j) + n_{00}(j)$ and $n_{11}(j) + n_{01}(j)$ equal the number of passed and failed runs, respectively. The latter two numbers are equal for all j. Similarly, for all j, the four counters sum op to the number of runs N.

Under the assumption that a high similarity to the error vector indicates a high probability that the corresponding parts of the software cause the detected errors, the calculated similarity coefficients rank the parts of the program with respect to their likelihood of containing the faults. Algorithm 1 concisely describes the SFL approach to fault localization.

To illustrate the approach, suppose that we apply the `RationalSort` function in Figure 2.1 to the input sequences $t_1, \dots, t_6$ shown in Table 2.2. The

Algorithm 1 SFL Algorithm

```
Input: Program $\mathcal{P}$, set of test cases $\mathcal{T}$, and similarity coefficient $s$
Output: Diagnostic report $D$
$N \leftarrow |\mathcal{T}|$
$M \leftarrow$ GET_NUMOFCOMPONENTS($\mathcal{P}$)
$D \leftarrow \emptyset$
for $j = 0$ to $M$ do
    $n_{11}(j) \leftarrow 0$
    $n_{10}(j) \leftarrow 0$
    $n_{01}(j) \leftarrow 0$
    $n_{00}(j) \leftarrow 0$
    $S[j] \leftarrow 0$                    ▷ Similarity $s$ of component $j$
end for
$(A, e) \leftarrow$ RUN_PROGRAM($\mathcal{P}, \mathcal{T}$)
for $i = 0$ to $N$ do
    for $j = 0$ to $M$ do
        if $a[i,j] = 1 \wedge e[i] = 1$ then
            $n_{11}(j) \leftarrow n_{11}(j) + 1$
        else if $a[i,j] = 0 \wedge e[i] = 1$ then
            $n_{01}(j) \leftarrow n_{01}(j) + 1$
        else if $a[i,j] = 1 \wedge e[i] = 0$ then
            $n_{10}(j) \leftarrow n_{10}(j) + 1$
        else if $a[i,j] = 0 \wedge e[i] = 0$ then
            $n_{00}(j) \leftarrow n_{00}(j) + 1$
        end if
    end for
end for
for $j = 0$ to $M$ do
    $S[j] \leftarrow s(n_{11}(j), n_{10}(j), n_{01}(j), n_{00}(j))$
end for
$D \leftarrow$ SORT($S$)
return $D$
```

block hit spectra for these runs are shown in the central part of the table ('1' denotes a hit), where block 5 corresponds to the body of the `RationalGT` function, which has not been shown in Figure 2.1. The test cases t_1, t_2, and t_6 are already sorted, and lead to passed runs. t_3 is not sorted, but the denominators in this sequence happen to be equal, hence no error occurs. The test case t_4 is the example from Section 1.1 of Chater 1: an error occurs during its execution, but goes undetected. For t_5 the program fails, since the calculated result is $\langle \frac{1}{1}, \frac{2}{2}, \frac{4}{3}, \frac{3}{4} \rangle$ instead of $\langle \frac{1}{4}, \frac{2}{2}, \frac{4}{3}, \frac{3}{1} \rangle$, which is a clear indication that an error has occurred. For this data, the calculated similarity coefficients $s_{x \in \{J,T,P\}}(1), \ldots, s_{x \in \{J,T,P\}}(5)$ listed at the bottom of Table 2.2 (correctly) identify

	block					
input	1	2	3	4	5	error
$t_1 = \langle\ \rangle$	1	0	0	0	0	0
$t_2 = \langle \frac{1}{4} \rangle$	1	1	0	0	0	0
$t_3 = \langle \frac{2}{1}, \frac{1}{1} \rangle$	1	1	1	1	1	0
$t_4 = \langle \frac{4}{1}, \frac{2}{2}, \frac{0}{1} \rangle$	1	1	1	1	1	0
$t_5 = \langle \frac{3}{1}, \frac{2}{2}, \frac{4}{3}, \frac{1}{4} \rangle$	1	1	1	1	1	1
$t_6 = \langle \frac{1}{4}, \frac{1}{3}, \frac{1}{2}, \frac{1}{1} \rangle$	1	1	1	0	1	0
s_J	0.17	0.20	0.25	0.33	0.25	
s_T	0.50	0.56	0.63	0.71	0.63	
s_O	0.41	0.45	0.50	0.58	0.50	

Table 2.2 SFL applied on six runs of the `RationalSort` program

block 4 as the most likely location of the fault.

2.2 EXPERIMENTAL SETUP

In this section we describe the benchmark set that we use in our experiments. We also detail how we extract the data of Figure 2.3, and define how we measure diagnostic accuracy.

2.2.1 *Benchmark Set*

In our study we work with two sets of faults that are available from the Software-artifact Infrastructure Repository (SIR [Do et al., 2005]):

- the Siemens set [Hutchins et al., 1994], which is a widely-used collection of benchmark faults in seven small C programs, and
- a set of faults in a somewhat larger program called `space`.

The Siemens set and `space` are the only programs in SIR that are ANSI-C compliant, and that could therefore be handled by our instrumentation tool (see below). Table 2.3 contains details about our benchmark set. For all eight programs, a correct version, and a number of faulty versions is available. Each faulty version contains a single fault, but this fault may span through multiple statements and/or functions. In addition, every program has a set of inputs (test cases) designed to provide full code coverage. For `space`, 1,000 test suites are provided that consist of a selection of (on average) 150 test cases.

In our experiments we were not able to use all the faults offered by the Siemens set and `space`. Because we conduct our experiments using block hit spectra, we cannot use faults that are located outside a block, such as global variable initializations. Versions 4 and 6 of `print_tokens` contain such faults and were therefore excluded. Version 9 of `schedule2`, version 32 of `replace`, and versions 1, 2, 32, and 34 of `space` were not considered in our experiments

Program	Faulty Versions	Blocks	Test Cases	Description
`print_tokens`	7	110	4,130	lexical analyzer
`print_tokens2`	10	105	4,115	lexical analyzer
`replace`	32	124	5,542	pattern recognition
`schedule`	9	53	2,650	priority scheduler
`schedule2`	10	60	2,710	priority scheduler
`tcas`	41	20	1,608	altitude separation
`tot_info`	23	44	1,052	information measure
`space`	38	777	13,585	Array definition language

Table 2.3 Set of programs used in the experiments

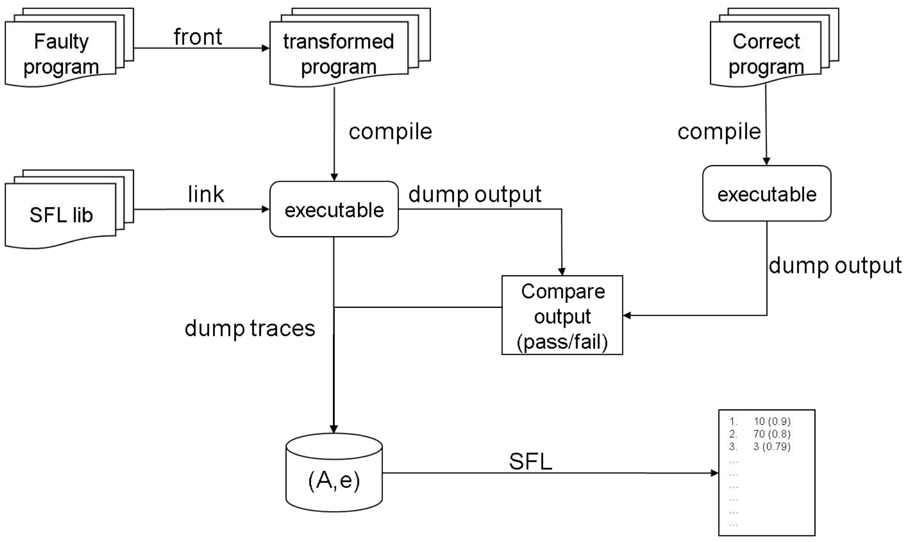


Figure 2.4 Experimental phases

because no test case fails and therefore the existence of a fault was never revealed. In total, we used 162 faulty versions in our experiments: 128 out of 132 faulty versions provided by the Siemens set, and and 34 out of 38 faulty versions of `space`.

2.2.2 *Data Acquisition*

An overview of the experimental environment is depicted in Figure 2.4, and its main steps described below.

Collecting Spectra To obtain block hit spectra, we automatically instrument the source code of all faulty versions of the programs in our benchmark set. A function call is inserted at the beginning of every block of code to log its execution. For the Siemens set, the spectra are generated for all test cases that are provided with the programs. For the faulty versions of `space`, we randomly choose one of the 1,000 test suites. For instrumentation we use the parser generator `Front` [Augusteijn, 2002], which is part of the development environment of NXP Semiconductors [NXP, 2009].The overhead of the instrumentation on the execution time is measured to be approximately 6% on average (with standard deviation of 5%). The programs were compiled on a `Fedora Core release 4` system with `gcc-3.2`. For details of the instrumentation process, see [Abreu et al., 2006a].

Error Detection As for each program our benchmark set provides a correct version, we use the output of the correct version of each program as error detection reference. We characterize a run as 'failed' if its output differs from the corresponding output of the correct version, and as 'passed' otherwise.

2.2.3 *Evaluation Metric*

As spectrum-based fault localization creates a ranking of blocks in order of likelihood to be at fault, we can retrieve how many blocks we still need to inspect until we hit the faulty block. If other blocks have the same similarity coefficient as the fault location, we use the average ranking position for these blocks. In those cases where the fault spans multiple locations, we verified that there is one block that is involved in the fault, and that is executed in all failed runs. This is the block that our evaluation metric is based on for the multiple-location faults. Repairing the fault at just this location would lead to iterative testing and debugging, but in our experiments we assume that the program is bug-free after the first iteration.

For all $j \in \{1, \ldots, M\}$, let $s(j)$ denote the similarity coefficient calculated for block j. Specifically, let j' be the index of the block that is known to contain the fault, and let $s(j')$ denote the similarity coefficient calculated for this block. Then, assuming that on average, half of the blocks j with $s(j) = s(f)$ are inspected before block f is found, the number of blocks that need to be inspected in total is given by

$$\tau = \frac{|\{j|s(j) > s(j')\}| + |\{j|s(j) \geq s(j')\}| - 1}{2}$$

We define accuracy, or quality of the diagnosis as the effectiveness to pinpoint the faulty block. This metric represents the fraction of all blocks that need not be considered when searching for the fault by traversing the ranking. It is defined as

$$q_d = 1 - \frac{\tau}{M-1} \tag{2.4}$$

In the remainder of this chapter, values for q_d will be expressed as percentages.

Sorensen-Dice	$\dfrac{2 \cdot n_{11}}{2 \cdot n_{11} + n_{01} + n_{10}}$
Anderberg	$\dfrac{n_{11}}{n_{11} + 2 \cdot (n_{01} + n_{10})}$
Simple-matching	$\dfrac{n_{11} + n_{00}}{n_{11} + n_{01} + n_{10} + n_{00}}$
Rogers and Tanimoto	$\dfrac{n_{11} + n_{00}}{n_{11} + n_{00} + 2 \cdot (n_{01} + n_{10})}$
Ochiai II	$\dfrac{n_{11} \cdot n_{00}}{\sqrt{(n_{11} + n_{01}) \cdot (n_{11} + n_{10}) \cdot (n_{00} + n_{01}) \cdot (n_{00} + n_{10})}}$
Russel and Rao	$\dfrac{n_{11}}{n_{11} + n_{01} + n_{10} + n_{00}}$

Table 2.4 Additional similarity coefficients evaluated; see [da Silva Meyer et al., 2004] for references

2.3 SIMILARITY COEFFICIENT IMPACT

At the end of Section 2.1.2 we reduced the problem of spectrum-based fault localization to finding resemblances between binary vectors. The key element of this technique is the calculation of a similarity coefficient. Many different similarity coefficients are used in practice, and in this section we investigate the impact of the similarity coefficient on the diagnostic accuracy q_d.

For this purpose, we evaluate q_d on all faults in our benchmark set, using nine different similarity coefficients. We only report the results for the Jaccard coefficient of Eq. (2.1), the coefficient used in the Tarantula fault localization tool as defined in Eq. (2.2), and the Ochiai coefficient of Eq. (2.3). We experimentally identified the latter as giving the best results among all eight coefficients used in a data clustering study in molecular biology [da Silva Meyer et al., 2004]. Table 2.4 contains the details of the coefficients that are involved in this study, and that have not already been introduced in Section 2.1.2. For brevity, the block index j as an argument to the counter functions $n_{11}, \ldots, n_{00}$ has been omitted.

In addition to the coefficient s_T of Eq. (2.2), the Tarantula tool uses a second coefficient, which amounts to the maximum of the two fractions in the denominator of Eq. (2.2). This second coefficient is interpreted as a *brightness* value for visualization purposes, but the experiments in [Jones and Harrold, 2005] indicate that s_T can be studied in isolation. For this reason, we have not taken the brightness coefficient into account.

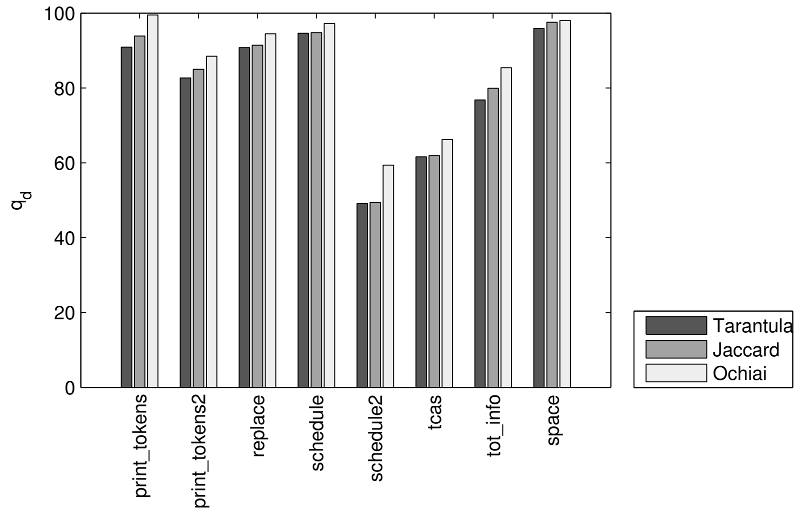


Figure 2.5 Diagnostic accuracy q_d

Figure 2.5 shows the results of this experiment. It plots q_d, as defined by Eq. (2.4), for the Tarantula, Jaccard, and Ochiai coefficients, averaged per program of our benchmark set. See [Abreu et al., 2006a] for more details on these experiments.

An important conclusion that we can draw from these results is that under the specific conditions of our experiment, the Ochiai coefficient gives a better diagnosis: it always performs at least as good as the other coefficients, with an average improvement of 4% over the second-best case, and improvements of up to 30% for individual faults. This effect can be explained as follows.

First, note that if $n_{11} = 0$, all three coefficients evaluate to 0, so blocks that are not executed in failed runs rank lowest. For the case that $n_{11} > 0$, the rankings produced by the Tarantula, Jaccard, and Ochiai coefficients are the same as the ranking produced by the respective coefficients below, where the block index j as an argument to the coefficients and counter functions n_{11} and n_{10} has been omitted for brevity.

$$s'_T = \frac{1}{1 + c_T \cdot \frac{n_{10}}{n_{11}}}$$

$$s'_J = \frac{n_{11}}{c_J + n_{10}}$$

$$s'_O = \frac{n_{11}}{1 + \frac{n_{10}}{n_{11}}}$$

Here $c_T = \frac{n_{11}+n_{01}}{n_{10}+n_{00}}$ and $c_J = n_{11} + n_{01}$, both of which are constant for all blocks, and do not influence the ranking. Coefficient s'_T is derived from s_T by dividing the numerator and denominator by $\frac{n_{11}}{n_{11}+n_{01}}$. Coefficient s'_O is derived by squaring s_O, dividing the denominator of the resulting fraction by the constant $n_{11} + n_{01}$, and dividing the numerator and denominator by n_{11}. Note that for $n_{11} > 0$, none of these operations modify the rankings implied by the Tarantula and Ochiai coefficients. The expression for s'_J is identical to that for s_J except for the introduction of c_J.

By thus rewriting the coefficients, it becomes apparent that the rankings implied by the Tarantula, Jaccard, and Ochiai coefficients depend only on n_{11} and n_{10}, i.e., the involvement of a block in passed and failed runs. It can also be seen that for $n_{10} = 0$, it follows that $s'_T = 1$, which implies that all blocks that are exclusively active in failed runs rank with the same, and highest s_T. This explains the improvement of Jaccard and Ochiai over Tarantula, because these coefficients both take n_{11} into account for ranking the blocks that have $n_{10} = 0$. The improved performance of the Ochiai coefficient over the Tarantula and Jaccard coefficients can be explained by observing that increasing n_{11} both increases the numerator, and decreases the denominator of s'_O, whereas to s'_T and s'_J, only one of these effects applies. As a result, compared to the other coefficients, Ochiai is much more sensitive to presence in failed runs than to presence in passed runs. This is well-suited to fault diagnosis because the execution of faulty code does not necessarily lead to a failure, while failures always involve a fault.

2.4 OBSERVATION QUALITY IMPACT

Before reaching a definitive decision to prefer one similarity coefficient over another, as suggested by the results in Section 2.3, we want to verify that the impact of this decision is independent of specific conditions in our experiments. Because of its relation to test coverage, and to the error detection mechanism used to characterize runs as passed or failed, an important condition in this respect is the quality of the error detection information used in the analysis.

In this section we define a measure of quality of the error observations, and show how it can be controlled as a parameter if the fault location is known, as is the case in our experimental setup. Thus, we verify the results of the previous section for varying observation quality values. Investigating the influence of this parameter will also help us to assess the potential gain of more powerful error detection mechanisms and better test coverage on diagnostic accuracy.

2.4.1 *A Measure of Observation Quality*

Correctly locating the fault is trivial if the column for the faulty part in the matrix of Figure 2.3 resembles the error vector exactly. This would mean that an error is detected if, and only if the faulty part is active in a run. In that case, any coefficient is bound to deliver a highly accurate diagnosis. However, spectrum-based fault localization suffers from the following phenomena.

- Most faults lead to an error only under specific input conditions. For example, if a conditional statement contains the faulty condition v<c, with v a variable and c a constant, while the correct condition would be v<=c, no error occurs if the conditional statement is executed, unless the value of v equals c.

- Similarly, as we have already seen in Section 1.1 of Chapter 1, errors need not propagate all the way to failures [Morell, 1990, Voas, 1992], and may thus go undetected. This effect can partially be remedied by applying more powerful error detection mechanisms, but for any realistic software system and practical error detection mechanism there will likely exist errors that go undetected.

As a result of both phenomena, the set of runs in which an error is detected will only be a subset of the set of runs in which the fault is activated[2]. We use the ratio of the size of these two sets as a measure of observation quality for a diagnosis problem. Using the notation of Section 2.1.2, we define

$$q_e = \frac{n_{11}(f)}{n_{11}(f) + n_{10}(f)} \tag{2.5}$$

where f is the known location of the fault, as in Section 2.2.3. This value can be interpreted as the unambiguity of the passed/failed data in relation to the fault being exercised, which may be loosely referred to as "error detection quality," hence the symbol q_e. In the remainder of this chapter, values for q_e will be expressed as percentages.

A problem with the q_e measure is that no information on undetected errors is available: $n_{10}(f)$ counts both the undetected errors, and the number of times the fault location was activated without introducing an error. This can be summarized as follows, where X, E, and D denote activation of the fault location, the occurrence of an error, and detection of an error, respectively:

$$\begin{array}{ccc|l}
X & E & D & \\
\hline
0 & 0 & 0 & n_{00}(f) \\
\left.\begin{array}{c}1\\1\end{array}\right. & \begin{array}{c}0\\1\end{array} & \left.\begin{array}{c}0\\0\end{array}\right\} & n_{10}(f) \\
1 & 1 & 1 & n_{11}(f)
\end{array}$$

Even though the ratio of the two contributions to $n_{10}(f)$ is unknown, it can still be influenced in our experimental setup. We now describe our procedure for doing so.

[2]In our experimental setup, we do not consider effects that carry over from one run to another, so conversely, if an error is detected, the fault is always active.

2.4.2 *Varying q_e*

Subject to various factors such as the nature of the fault, the similarity coefficient used in the diagnosis, the design of the test data, but also the compiler and the operating system, each faulty version of a program in our benchmark set has an inherent value for q_e, which can be evaluated by collecting spectra and error detection information for all available test cases, and performing the diagnosis of Section 2.1.2. For the Siemens set, this inherent value for q_e ranges from 1.4% for `schedule2` to 20.3% for `tot_info`, whereas for `space` this value is measured to be 50.9% on average for our selection of test suites.

We can construct a different value for q_e by excluding runs that contribute either to $n_{11}(f)$ or to $n_{10}(f)$ as follows.

- Excluding a run that activates the fault location, but for which no error has been detected lowers $n_{10}(f)$, and will *increase* q_e.
- Excluding a run that activates the fault location and for which an error has been detected lowers $n_{11}(f)$, and will *decrease* q_e.

Excluding runs to achieve a certain value of q_e raises the question of which particular selection of runs to use. For this purpose we randomly sample passed or failed runs from the set of available runs to control q_e within a 99% confidence interval. We verified that the variance in the values measured for q_d is negligible.

Note that for decreasing q_e, i.e., obscuring the fault location, we have another option: setting failed runs to 'passed.' In our experiments we have tried both options, but the results were essentially the same. The results reported below are generated by excluding failed runs. Conversely, setting passed runs that exercise the fault location to 'failed' is not a good alternative for increasing q_e: this may obstruct the diagnosis as we cannot be certain that an error occurs for a particular data input. Moreover, it may allocate blame to parts of the program that are not related to the fault. Thus, excluding runs is always to be preferred as this does not compromise observation consistency. This way, we were able to vary q_e from 1% to 100% for all programs.

2.4.3 *Similarity Coefficients Revisited*

Using the technique for varying q_e introduced in Section 2.4.2 we revisit the comparative study of similarity coefficients in Section 2.3. Figure 2.6 shows q_d for the three similarity coefficients, and values of q_e ranging from 1% to 100%. In this case, instead of averaging per program in our benchmark set, as we did in Figure 2.5, we arithmetically averaged q_d over all 162 faulty program versions to summarize the results (this is valid because q_d is already normalized with respect to program size). As in Figure 2.5, the graphs for the individual programs are similar, only having different offsets.

These results confirm what was suggested by the experiment in Section 2.3. The Ochiai similarity coefficient leads to a better diagnosis than the other

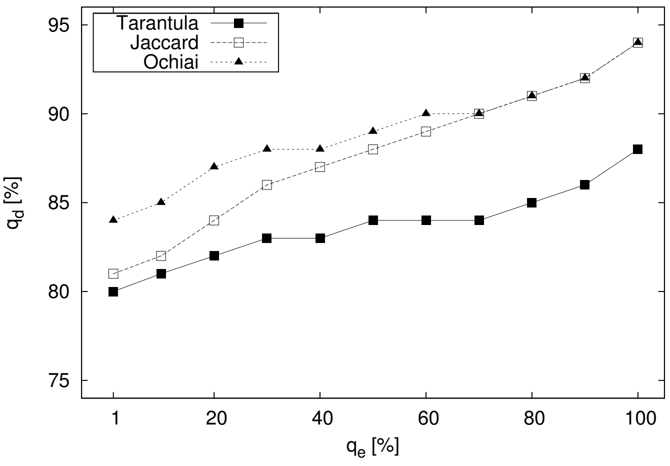


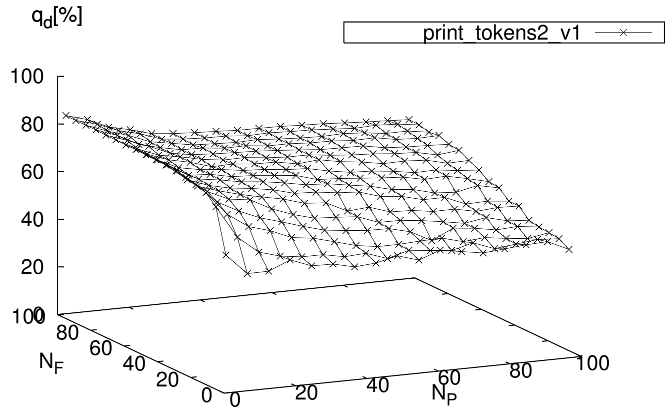
Figure 2.6 Observation quality impact

eight, including the Jaccard coefficient and the coefficient of the Tarantula tool. Compared to the Jaccard coefficient the improvement is greatest for lower observation quality. As q_e increases, the performance of the Jaccard coefficient approaches that of the Ochiai coefficient. The improvement of the Ochiai coefficient over the Tarantula coefficient appears to be consistent.

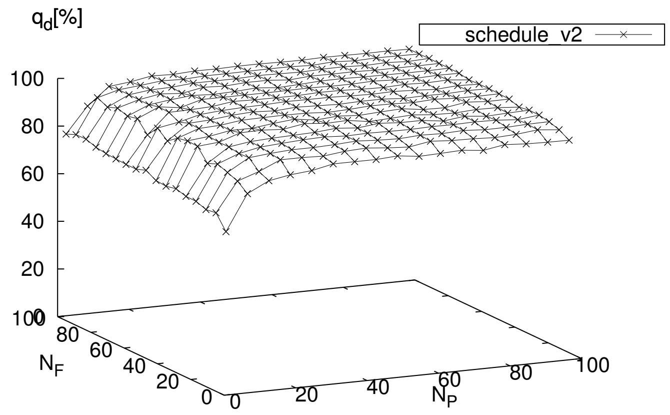
Another observation that can be made from Figure 2.6 is that all three coefficients provide a useful diagnosis (q_d around 80%) already for low q_e values ($q_e = 1\%$ implies that only around 1% of the runs that exercised the faulty block actually resulted in a failed run). The diagnostic accuracy increases as the quality of the error detection information improves, but the effect is not as strong as we expected. This suggests that more powerful error detection mechanisms, or test sets that cover more input conditions will have limited gain. In the next section we investigate a possible explanation, namely that not only the quality of observations, but also their quantity determines the diagnostic accuracy.

2.5 OBSERVATION QUANTITY IMPACT

To investigate the influence of the number of runs on the accuracy of spectrum-based fault localization, we evaluated q_d while varying the numbers of passed (N_P) and failed runs (N_F) that are involved in the diagnosis, across the benchmark set. Since all interesting effects appear to occur for small numbers of runs, we have focused on the range of 1..100 passed and



(a)



(b)

Figure 2.7 Observation quantity impact

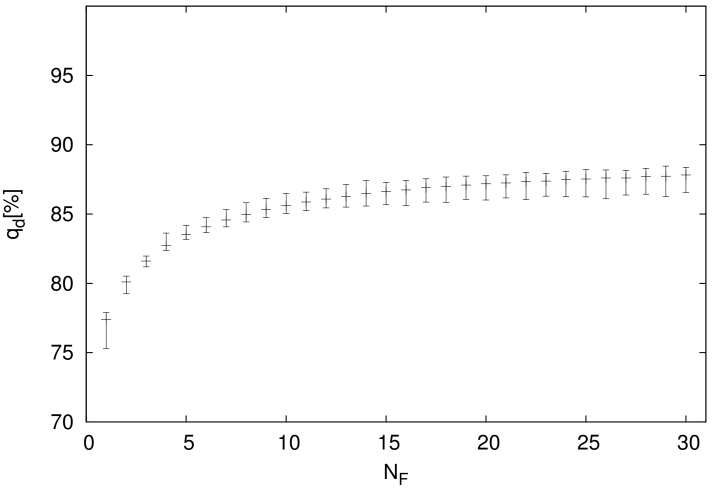


Figure 2.8 Impact of N_F on q_d, on average

failed runs. Although the number of available runs in the Siemens set ranges from 1,052 (`tot_info`) to 5,542 (`replace`), the number of runs that fail is comparatively small, down to a single run for `tcas` version 8. The situation is comparable for `space`, with only 7 out of 13,585 runs failing for version 33. For this reason, even in the range 1..100, some selections of failed runs are not possible for some of the faulty versions.

Figure 2.7 shows two representative examples of such evaluations, where we plot q_d according to the Ochiai coefficient for N_P and N_F varying from 1 to 100. For each entry in these graphs, we averaged q_d over 50 randomly selected combinations of N_P passed runs and N_F failed runs, where we verified that the variance in the measured values of q_d is negligible. Apart from the apparent monotonic increase of q_d with N_F, we observe that for version 1 of `print_tokens2`, q_d decreases when more passed runs are added (Figure 2.7(a)), while q_d increases for version 2 of `schedule` (Figure 2.7(b)).

Given a set of faulty program versions that all allow failed runs to be selected up to a given value for N_F, we can average the measured values for q_d again over these versions. This summarizes several graphs of the kind shown in Figure 2.7. This way, in Figure 2.8 we plot the average q_d using the Ochiai coefficient for $1 \leq N_F \leq 30$ and $1 \leq N_P \leq 100$, projected on the $N_F \times q_d$ plane. The ticks on the vertical bars in the graph indicate the minimum, maximum, and average observed for the 100 values for N_P. With this limited range for N_F we can still use 110 of the 162 versions in the benchmark set, whereas for $N_F \leq 100$, we can only use 60. We verified that for $N_F \leq 15$, for which we can use 128 versions, the results are essentially the same.

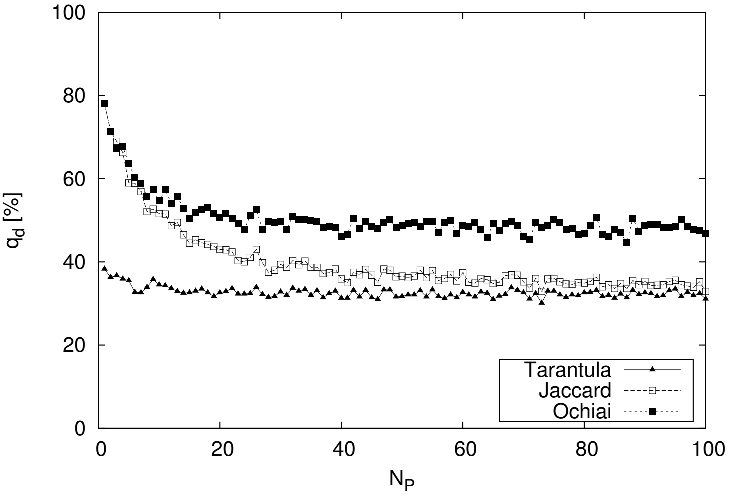


Figure 2.9 Impact of N_P on q_d for `print_tokens2` v1, and $N_F = 6$

A first conclusion that we draw from Figure 2.8 is that overall, adding failed runs improves the accuracy of the diagnosis. However, the benefit of having more than around 10 runs is marginal on average. In addition, because the measurements for varying N_P show little scattering in the projection, we can conclude that on average, N_P has little influence.

Inspecting the results for the individual program versions confirms our observation that adding failed runs consistently improves the diagnosis. However, although the effect does not show on average, N_P can have a significant effect on q_d for individual runs. As shown in Figure 2.7, this effect can be negative or positive. This shows more clearly in Figures 2.9 and 2.10, which contain cross sections of the graphs in Figure 2.7 at $N_F = 6$. To factor out any influence of N_F, we have created similar cross sections at the maximum number of failed runs. Across the entire benchmark set, we found that the effect of adding more passed runs stabilizes around $N_P = 20$.

Returning to the influence of the similarity coefficient once more, Figures 2.9 and 2.10 further indicate that the superior performance of the Ochiai coefficient is consistent also for varying numbers of runs. We have not plotted q_d for the other coefficients in Figure 2.7, but we verified this observation for all program versions, with N_P and N_F varying from 1 to 100.

From our experiments on the impact of the number of runs we can draw the following conclusions. All of these are in the context of our benchmark set, which has the important characteristic that most faults involve a single location in the source code. First, including more failed runs is safe because the accuracy of the diagnosis either improves or remains the same. This is

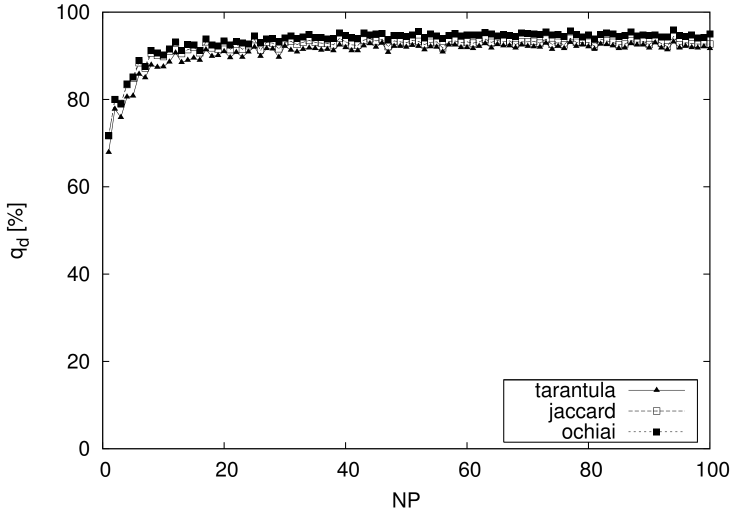


Figure 2.10 Impact of N_P on q_d for `schedule` v2, and $N_F = 6$

observed due to the fact that failed runs add evidence about the block that is causing the program to fail, and hence causing it to move up in the ranking. Our results show that the optimum value for N_F is in the order of 10 runs. To what extent this result depends on characteristics of the fault or program is subject to further investigation. Second, while stabilizing around $N_P = 20$, the effect of including more passed runs is unpredictable, and may actually decrease q_d. In fact, q_d decreases only if the faulty block is touched often in passed runs, as spectrum-based fault localization works under the assumption that if a block is touched often in passed runs, it should be exonerated. Besides, a large number of runs can apparently compensate weak error detection quality: even for small q_e, a large amount of runs provides sufficient information for good diagnostic accuracy, as shown in Figure 2.6. Lastly, the number of runs has no influence on the superiority of the Ochiai coefficient.

2.6 RELATED WORK

Program spectra themselves were introduced in [Reps et al., 1997], where hit spectra of intra-procedural paths are analyzed to diagnose year 2000 problems. The distinction between count spectra and hit spectra is introduced in [Harrold et al., 2000], where several kinds of program spectra are evaluated in the context of regression testing.

In the introduction of this chapter we already mentioned three practical diagnosis/debugging tools [Chen et al., 2002, Dallmeier et al., 2005, Jones et al., 2002] that are essentially based on spectrum-based fault localization. Pinpoint [Chen et al., 2002] is a framework for root cause analysis on the J2EE platform and is targeted at large, dynamic Internet services, such as webmail services and search engines. The error detection is based on information coming from the J2EE framework, such as caught exceptions. The Tarantula tool [Jones et al., 2002] has been developed for the C language, and works with statement hit spectra. AMPLE [Dallmeier et al., 2005] is an Eclipse plugin for identifying faulty classes in Java software. However, although we have recognized that it uses hit spectra of method call sequences, we didn't include its weight coefficient in our experiments because the calculated values are only used to collect evidence about classes, not to identify suspicious method call sequences.

Diagnosis techniques can be classified as white box or black box, depending on the amount of knowledge that is required about the system's internal component structure and behavior. An example of a white box technique is model-based diagnosis (see, e.g., [de Kleer and Williams, 1987]), where a diagnosis is obtained by logical inference from a formal model of the system, combined with a set of run-time observations. White box approaches to software diagnosis exist (see, e.g., [Wotawa et al., 2002]), but software modeling is extremely complex, so most software diagnosis techniques are black box. Since the technique studied in this chapter requires practically no information about the system being diagnosed, it can be classified as a black box technique.

Examples of other black box techniques are Nearest Neighbor [Renieris and Reiss, 2003], dynamic program slicing [Agrawal et al., 1993], Sober [Liu et al., 2006], Delta Debugging [Zeller, 2002, Zeller and Hildebrandt, 2002], CROSSTAB [Wong et al., 2008], TWT [Yilmaz et al., 2008], PPDG [Baah et al., 2008], Value replacement [Jeffrey et al., 2008, Xie and Notkin, 2005], and Predicate Switching [Zhang et al., 2006]. The Nearest Neighbor technique first selects a single failed run, and computes the passed run that has the most similar code coverage. Then it creates the set of all statements that are executed in the failed run but not in the passed run. Dynamic program slicing narrows down the search space to the set of statements that influence a value at a program location where the failure occurs (e.g., an output variable). Sober is a statistical debugging tool which analysis traces fingerprints and produces a ranking of predicates by contrasting the evaluation bias of each predicate in failing cases against those in passing cases. Delta Debugging compares the program states of a failing and a passing run, and actively searches for failure-inducing circumstances in the differences between these states. In [Gupta et al., 2005] Delta Debugging is combined with dynamic slicing in 4 steps: (1) Delta Debugging is used to identify the minimal failure-inducing input; step (2) computes the forward dynamic slice of the input variables obtained in step 1; (3) the backward dynamic slice for the failed run is computed; (4) finally it returns the intersection of the slices given by the previous two steps. This set of statements is likely to contain the faulty code. CROSSTAB exploits the joint distribution of two variables derived from coverage information of different program executions to compute a ranked list of possible faults. In TWT, discrepancies between execution time spectra obtained from correct and failing tests are used to locate possible faults. PPDG presents an innovative model of a program's behavior over a set of test inputs, which is called probabilistic program dependence graph, that facilitates reasoning about program failures. Value Replacement is a state-altering technique that alters the state of an execution program to locate faulty statements. Predicate Switching attempts to isolate erroneous code by identifying predicates whose outcomes can be altered during a failing run to cause it to pass.

Regarding our observation that for the benchmark faults in the Siemens set and `space` program the diagnostic quality does not change significantly when using more than 20 passed runs and 10 failed runs, in [Yu et al., 2008], the effect of several test-suite reduction strategies on the accuracy of spectrum-based fault localization is studied, and the authors reach the same conclusions. The SFL variants taken into account in this study include Tarantula, and the Jaccard and Ochiai coefficients. In [Baudry et al., 2006] a study on the dependency between test cases and SFL is given, and the authors conclude that test suites should be chosen with care to optimize diagnostic accuracy. To our knowledge, no other evaluations of the diagnostic quality of similarity coefficients in the context of varying observation quality and quantity exist.

2.7 SUMMARY

Reducing fault localization effort greatly improves the test-diagnose-repair cycle. In this chapter, we have investigated the influence of different parameters on the accuracy of the diagnosis delivered by spectrum-based fault localization. Our starting point was a previous study on the influence of the similarity coefficient, which indicated that the Ochiai coefficient, known from the biology domain, can give a better diagnosis than eight other coefficients, including those used by the Pinpoint [Chen et al., 2002] and Tarantula [Jones and Harrold, 2005] tools.

By varying the quality and quantity of the observations on which the fault localization is based, we have established this result in a much wider context. We conclude that the superior performance of the Ochiai coefficient in diagnosing single-site faults in the Siemens set is consistent, and does not depend on the quality or quantity of observations. We expect that this result is relevant for the Tarantula tool, whose analysis is essentially the same as ours. This superiority was already observed in other domains, such as clustering in the molecular biology [da Silva Meyer et al., 2004]. [Bolton, 1991] suggests the geometrical interpretation of the coefficient as a possible explanation for its good performance.

In addition, we find that even for the lowest quality of observation that we applied ($q_e = 1\%$, corresponding to a highly ambiguous error detection), the accuracy of the diagnosis is already quite useful: around 80% for all the programs in the Siemens set, which means that on average, only 20% of the code remains to be investigated to locate the fault. Furthermore, we conclude that while accumulating more failed runs only improves the accuracy of the diagnosis, the effect of including more passed runs is unpredictable. With respect to failed runs we observe that only a few (around 10) are sufficient to reach near-optimal diagnostic performance. Adding passed runs, however, can both improve or degrade diagnostic accuracy. In either case, including more than around 20 passed runs has little effect on the accuracy. Our findings imply that using current similarity coefficients in combination with all available test data does not guarantee optimum diagnostic results, and that if a selection can be made, care should be taken when selecting the passed runs to include in the analysis. A possible strategy for making this selection is the nearest neighbor technique of [Renieris and Reiss, 2003], but the influence of such strategies requires further investigation. The fact that a few observations can already provide a near-optimal diagnosis enables the application of spectrum-based fault localization methods within continuous (embedded) processing, where only limited observation horizons can be maintained.

3 Industrial Case Studies with SFL

ABSTRACT

Automated diagnosis of errors detected during software testing can improve the efficiency of the debugging process, and can thus help to make software more reliable. In this chapter we discuss the application of software fault localization through the analysis of program spectra (SFL) in the area of embedded software in high-volume consumer electronics products. We discuss why the technique is particularly well suited for this application domain, and through experiments on an industrial test case we demonstrate that it can lead to highly accurate diagnoses of realistic errors. This chapter shows that SFL lends particularly well to resource-constrained systems, due to its low time/space complexity and the fact that no modeling effort is required. Furthermore, our experiments show that the diagnostic quality yield by SFL lead developers to pinpoint the root cause of software failures quickly. In addition, compared to the previous chapter, we observed that the large the systems the better the diagnostic quality becomes (less than 1% of the code needs to be inspected in these experiments, in contrast to the 20% obtained in the Siemens benchmark set's experiments).

//

Software reliability can generally be improved through extensive testing and debugging, but this is often in conflict with market conditions: software cannot be tested exhaustively, and of the bugs that are found, only those with the highest impact on the user-perceived reliability can be solved before the release. In this typical scenario, testing reveals more bugs than can be solved, and debugging is a bottleneck for improving reliability. Automated debugging techniques can help to reduce this bottleneck.

The subject of this chapter is a particular automated debugging technique, namely software fault localization through the analysis of *program spectra* [Reps et al., 1997]. These can be seen as projections of execution traces that indicate which parts of a program were active during various runs of that program. The diagnosis consist in analyzing the extent to which the activity of specific parts correlates with errors detected in the different runs.

Locating a fault is an important step in actually solving it, and program spectra have successfully been applied for this purpose in several tools focusing on various application domains, such as Pinpoint [Chen et al., 2002], which focuses on large, dynamic on-line transaction processing systems, AMPLE [Dallmeier et al., 2005], which focuses on object-oriented software, and Tarantula [Jones et al., 2002], which focuses on C programs.

In this chapter, we discuss the applicability of the technique to embedded software, and specifically to embedded software in high-volume consumer electronics products. Software has become an important factor in the development, marketing, and user-perception of these products, and the typical combination of limited computing resources, complex systems, and tight development deadlines make the technique a particularly attractive means for improving product reliability.

To support our argument, we report the outcome of two experiments, where we diagnosed two different errors occurring in the control software of a particular product line of television sets from a well-known international consumer electronics manufacturer. In both experiments, the technique is able to locate the (known) faults that cause these errors quite well, and in one case, this implies an accuracy of a single statement in approximately 450K lines of code.

The remainder of this chapter is organized as follows. In Section 3.1 we discuss its applicability to embedded software in consumer electronics products. In Sections 3.2, 3.3 and 3.4 we describe our experiments with two industrial test cases. We summarize this chapter in Section 3.5.

3.1 RELEVANCE TO EMBEDDED SOFTWARE

The effectiveness of the diagnosis technique described in the previous chapter has already been demonstrated in several articles (see, e.g., [Abreu et al., 2006a], [Chen et al., 2002], [Jones et al., 2002]). Although we have already explained the adequacy of the technique for embedded software, in this section, we elaborate more in detail the benefits and discuss the issues specifically related to debugging embedded software in consumer electronics products. Especially because of constraints imposed by the market, the conditions under which this software is developed are somewhat different from those for other software products:

- To reduce unit costs, and often to ensure portability of the devices, the software runs on specialized hardware, and computing resources are limited.

- As a consequence, many facilities that developers of non-embedded software have come to rely on are absent, or are available only in rudimentary forms. Examples are profiling tools that give insight in the dynamic behavior of systems.

- At the same time, the systems are highly concurrent, and operate at a low level of abstraction from the hardware. Therefore, their design and implementation are complicated by factors that can largely be abstracted away from in other software systems, such as deadlock prevention, and timing constraints involved in, e.g., writing to the graphics display only in those fractions of a second that the screen is not being refreshed.

- On top of challenges that the entire software industry has to deal with, such as geographically distributed development organizations, the strong competition between manufacturers of consumer electronics makes it absolutely vital that release deadlines are met.

- Although important safety mechanisms, such as short-circuit detection, are sometimes implemented in software, for a large part of the functionality there are no personal risks involved in transient failures.

Consequently, it is not uncommon that consumer electronics products are shipped with several known software faults outstanding. To a certain extent, this also holds for other software products, but the combination of the complexity of the systems, the tight constraints imposed by the market, and the relatively low impact of the majority of possible system failures creates a unique situation. Instead of aiming for correctness, the goal is to create a product that is of value to customers, despite its imperfections, and to bring the reliability to a commercially acceptable level (also compared to the competition) before a product must be released. More than with other types of commercial software, instead of aiming for correctness, the goal is to reduce the unreliability of the system to an acceptable level (also compared to the competition) before a product must be released.

The technique of Chapter 2 can help to reach this goal faster, and may thus reduce the time-to-market, and lead to more reliable products. Specific benefits are the following:

- As a black-box diagnosis technique, it can be applied without any additional modeling effort. This effort would be hard to justify under the market conditions described above. Moreover, concurrent systems are difficult to model.

- The technique improves insight in the run-time behavior. For embedded software in consumer electronics, this is often lacking, because of the concurrency, but also because of the decentralized development.

- We expect that the technique can easily be integrated with existing testing procedures, such as overnight playback of recorded usage scenarios. In addition to the information that errors have occurred in some scenarios, this gives a first indication of the parts of the software that are likely to be involved in these errors. Given geographically distributed development organizations, it may also help to identify which teams of developers to contact.

- Last but not least, the technique is light-weight, which is relevant because of the specialized hardware and limited computing resources. All that is needed is some memory for storing program spectra, or for calculating the similarity coefficients on the fly (which reduces the space complexity from $O(M \times N)$ to $O(N)$, see Section 3.2.5). Profiling tools such as `gcov` are convenient for obtaining program spectra, but they are typically not available in a development environment for embedded software. However, the same data can be obtained through source code instrumentation.

While none of these benefits are unique, their combination makes program spectrum analysis an attractive technique for diagnosing embedded software in consumer electronics.

3.2 EXPERIMENTS WITH ADOC

While the benchmark problems are well-suited for studying the influence of parameters such as the similarity coefficient, and the quality and quantity of the observations, they give little indication on the accuracy of spectrum-based fault localization for large-scale codes, and the kind of problems that are encountered in practice. For this reason, in this section and the next we report our experience with implementing SFL for an industrial software product, namely the control software of a particular product line of hybrid analog/digital LCD television sets. These experiments are done in the context of the TRADER project [Trader, 2009], whose goal is to improve the user-perceived reliability of consumer electronics systems with embedded software. In this section we describe the experimental platform, and our implementation of SFL for it.

3.2.1 *Platform*

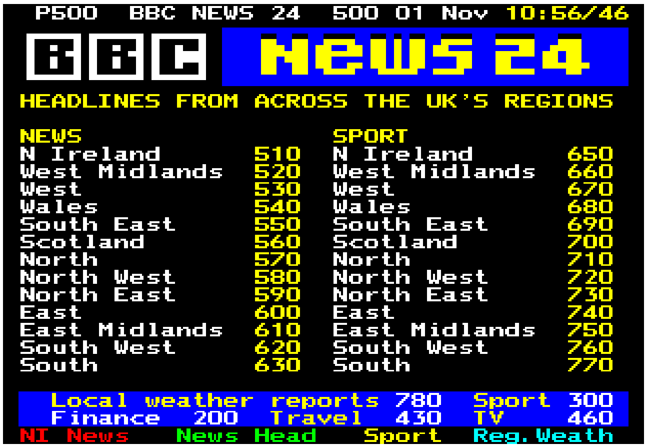
The subject of our experiments is the control software in a particular product line of analog television sets. All audio and video processing is implemented in hardware, but the software is responsible for tasks such as decoding remote control input, displaying the on-screen menu, and coordinating the hardware (e.g., optimizing parameters for audio and video processing based on an analysis of the signals). Most teletext[1] functionality is also implemented in software.

The software itself consists of approximately 450K lines of C code, which is configured from a much larger (several MLOC) code base of Koala software components [van Ommering et al., 2000].

The control processor is a MIPS[2] running a small multi-tasking operating system. Essentially, the run-time environment consists of several threads with

[1] A standard for broadcasting information (e.g., news, weather, TV guide) in text pages, popular in Europe. See Figure 3.1 for an example of a teletext page.

[2] MIPS is a reduced instruction set computing (RISC) microprocessor architecture developed by MIPS Technologies - for further information see [Sweetman, 2006].



```
 P500  BBC NEWS 24   500 01 Nov 10:56/46
BBC News 24
HEADLINES FROM ACROSS THE UK'S REGIONS

NEWS                    SPORT
N Ireland          510  N Ireland          650
West Midlands      520  West Midlands      660
West               530  West               670
Wales              540  Wales              680
South East         550  South East         690
Scotland           560  Scotland           700
North              570  North              710
North West         580  North West         720
North East         590  North East         730
East               600  East               740
East Midlands      610  East Midlands      750
South West         620  South West         760
South              630  South              770

  Local weather reports 780   Sport  300
  Finance   200  Travel   430   TV     460
NI News    News Head    Sport   Reg.Weath
```

Figure 3.1 A teletext page example

increasing priorities, and for synchronization purposes, the work on these threads is organized in 315 logical threads inside the various components. Threads are preempted when work arrives for a higher-priority thread.

The total available RAM memory in consumer sets is two megabytes, but in the special developer version that we used for our experiments, another two megabytes was available. In addition, the developer sets have a serial connection, and a debugger interface for manual debugging on a PC.

3.2.2 *Faults*

We diagnosed two faults, one existing, and one that was seeded to reproduce an error from a different product line.

Load Problem. A known problem with the specific version of the control software that we had access to, is that after teletext viewing, the CPU load when watching television (TV mode) is approximately 10% higher than before teletext viewing (see Figure 3.2).

This is illustrated in Figure 3.3, which shows the CPU load for the following scenario: one minute TV mode, 30 s teletext viewing, and one minute of TV mode. The CPU load clearly increases around the 60th sample, when the teletext viewing starts, but never returns to its initial level after sample 90, when we switch back to TV mode.

Teletext Lock-up Problem. Another product line of television sets provides a function for searching in teletext pages. An existing fault in this functionality entails that searching in a page without visible content locks up the teletext system. A likely cause for the lock-up is an inconsistency in the values of two state variables in different components, for which only specific combinations

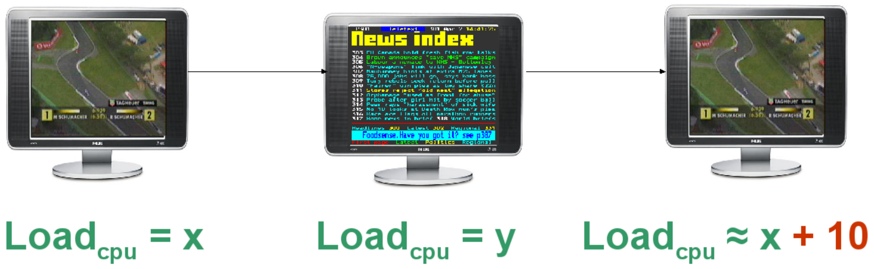


Figure 3.2 CPU load problem

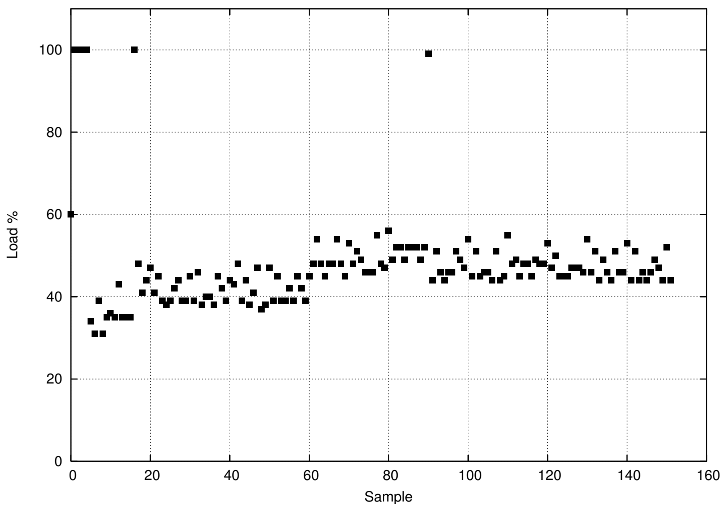


Figure 3.3 CPU load measured per second

are allowed. We hard-coded a remote control key-sequence that injects this error on our test platform.

3.2.3 *Implementation*

We wrote a small Koala component for recording and storing program spectra, and for transmitting them off the television set via the serial connection. The transmission is done on a low-priority thread while the CPU is otherwise idle, in order to minimize the impact on the timing behavior. Pending their transmission via the serial connection, our component caches program spectra in the extra memory available in our developer version of the hardware.

For diagnosing the load problem we obtained hit spectra for the logical threads mentioned in Section 3.2.1, resulting in spectra of 315 binary flags.

We approached the lock-up problem at a much finer granularity, and obtained block hit spectra for practically all blocks of code in the control software, resulting in spectra of over 60,000 flags.

The hit spectra for the logical threads are obtained by manually instrumenting a centralized scheduling mechanism. For the block hit spectra we automatically instrumented the entire source code using the Front [Augusteijn, 2002] parser generator. See [Abreu et al., 2006b] for details of the instrumentation process.

In Chapter 2 we use program spectra for different runs of the software, but for embedded software in consumer electronics, and indeed for most interactive systems, the concept of a run is not very useful. Therefore we record the spectra per *transaction*, instead of per run, and we use two different notions of a transaction for the two different faults that we diagnosed:

- for the load problem, we use a periodic notion of a transaction, and record the spectra per second.
- for the lock-up problem, we define a transaction as the computation in between two key-presses on the remote control.

3.2.4 *Diagnosis*

For the load problem we used the scenario of Figure 3.3. We marked the last 60 spectra, for the second period of TV mode as 'failed,' and those of earlier transactions as 'passed.' In the ranking that follows from the analysis using the technique described in Chapter 2, the logical thread that had been identified by the developers as the actual cause of the load problem was in the second position out of 315. In the first position was a logical thread related to teletext, whose activation is part of the problem, so in this case we can conclude that although the diagnosis is not perfect, the implied suggestion for investigating the problem is quite useful.

For the lock-up problem, we used a proper error detection mechanism. On each key-press, when caching the current spectrum, a separate routine verifies the values of the two state variables, and marks the current spectrum as failed if they assume an invalid combination. Although this is a special-purpose mechanism, including and regularly checking high-level assert-like statements about correct behavior is a valid means to increase the error-awareness of systems.

Using a very simple scenario of 23 key-presses that essentially (1) verifies that the TV and teletext subsystems function correctly, (2) triggers the error injection, and (3) checks that the teletext subsystem is no longer responding, we immediately got a good diagnosis of the detected error: the first two positions in the total ranking of over 60,000 blocks pointed directly to our error injection code. Adding another three key-presses to exonerate an uncovered branch in this code made the diagnosis perfect: the exact statement that introduced the state inconsistency was located out of approximately 450K lines of source code.

3.2.5 *Overhead*

Especially the results for the lock-up problem suggest that program spectra, and their application to fault diagnosis are a viable technique and useful tool in the area of embedded software in consumer electronics. However, there are a number of issues with our implementation.

First, we cannot claim that we have not altered the timing behavior of the system. Because of its rigorous design, the TV is still functioning properly, but everything runs much slower with the block-level instrumentation (e.g., changing channels now takes seconds). One reason is that currently, we collect block *count* spectra at byte resolution, and convert to block *hit* spectra off-line. Updating the counters in a multi-threaded environment requires a critical section for every executed block, which is hugely expensive. Fortunately, this information is not used, and we believe we can implement a binary flag update without a critical section.

Second, we cache the spectra of passed transactions, and transmit them off the system during CPU idle time. Because of the low throughput of the serial connection, this may become a bottleneck for large spectra and larger scenarios. In our case we could store 25 spectra of 65,536 counters, which was already slowing down the scenarios with more than that number of transactions, but even with a more memory-efficient implementation, this inevitably becomes a problem with, for example, overnight testing.

For many purposes, however, we will not have to store the actual spectra. In particular for fault diagnosis, ultimately we are only interested in the calculated similarity coefficients, and all similarity coefficients that we are aware of are expressed in terms of the four counters n_{00}, n_{01}, n_{10}, and n_{11} introduced in Chapter 2. If an error detection mechanism is available, like in our experiments with the lock-up problem, then these four counters can be calculated on the fly, and the memory requirements become linear in the number of components.

Although these two experiments were valuable to assess the diagnostic quality of SFL, they are still controlled experiments: faults were seeded to resemble faults seen during development. However, the results obtained gave us leverage and confidence to tackle current faults. Results of in-field experiments are reported in the next section.

3.3 EXPERIMENTS WITH TV520

In this section we describe our experience with applying the techniques of Chapter 2 to yet another industrial test case.

3.3.1 *Platform*

One of the products of NXP Semiconductors is the TV520 platform for building hybrid analog/digital LCD television sets (used in Philips television sets), which in turn serves as the basis for televisions sets manufactured by NXP's customers. The TV520 platform comprises one or two proprietary CPUs for audio and video signal processing, plus a MIPS CPU running the control software (under Linux). More details on TV520 can be found on the NXP website [NXP, 2009].

Our experiments are performed on development versions of television sets based on TV520. All problems that we diagnosed are in the control software of the sets, which is responsible for tasks such as

- decoding the remote-control input,
- navigating the on-screen menu,
- coordinating the hardware (e.g., the tuner),
- coordinating the audio and video processing on the proprietary CPUs, based on an analysis of the signals,
- teletext decoding, viewing, and navigation.

The control software comprises roughly one million lines of C code (configured from a much larger code base of software components), and $150,000$ blocks. With a block of code we mean a C language statement, where we do not distinguish between the individual statements of a compound statement, but where we do distinguish between the cases of a switch statement (see Section 2.1.1 of Chapter 2 for an example).

3.3.2 *Space Efficiency*

In a regular computing environment, storing all program spectra for a series of test cases is no problem, and this is how we implemented spectrum-based fault localization for the experiments reported in the previous section and chapter. In the embedded domain, however, memory is typically a scarce resource, and storing all spectra to be post-processed at diagnosis time is usually not an option. As an indication, the 64 MB that is available for our experiments is shared with the application binaries and variables, and depending on the usage scenario, only approximately 10 spectra can be stored in the remaining space, using a byte for each flag. Bytes are the smallest data unit that can be transferred to/from memory directly. There is a potential problem when different bytes in the same word are updated from within different threads. However, due to an extra layer of virtual threads running on top of the operating systems threads, the probability that two blocks that are mapped on the same word are executed in parallel is negligible in practice, and we chose to ignore this issue.

Fortunately, although the available storage space is quite limited, the set of spectra that a diagnosis is based on contains much more information than needed, and can easily be compacted at run-time. In the end, the only information that is needed to generate the ranking are the four sets of counters $n_{00}(j), \ldots, n_{11}(j)$, introduced in Chapter 2, and the space required to store these is linear in the size of the program, not in the number of test cases. To avoid having to store the actual spectra, we can update the counters right after a run has finished, and the passed/failed verdict has become available:

- For a passed run, and all blocks j: if block j has been active, increment $n_{10}(j)$, otherwise increment $n_{00}(j)$.
- For a failed run, and all blocks j: if block j has been active, increment $n_{11}(j)$, otherwise increment $n_{01}(j)$.

After thus having processed the program spectrum of a passed or failed run, the spectrum itself can be discarded. Any time after processing at least one failed run, the diagnosis can be performed by evaluating the similarity coefficient of choice for all blocks, and by ranking the blocks based on their calculated coefficients.

In our implementation, we use a small circular buffer to cache recently recorded spectra until they can be processed on a low priority thread (see Figure 3.4). Two pointers cycle through this buffer: i_1, pointing to the current spectrum, where the system activity is being recorded, and i_2, pointing to the first spectrum whose contributions must still be added to the sets of counters $n_{00}(j), \ldots, n_{11}(j)$. While in theory, spectra can be overwritten if insufficient idle time is available for processing spectra of previous runs, this is not a problem in our experiments, and spectra are cleaned up almost immediately after they are cached by advancing i_1. However, if runs are delimited automatically, it may be necessary to tune the rate at which spectra are generated to the size of the buffer.

3.3.3 *Implementation*

As we describe in Section 2.2.2, the spectra are obtained via instrumentation. Compared to the experiments on the benchmark set, additional, but nonfundamental difficulties that are encountered in the NXP development environment are the following.

- Although the code base is ANSI-C compliant, several GNU extensions that are inserted by the preprocessor cannot be handled by the `Front` parser, which is not normally applied to preprocessed code, and require a work-around.
- Parallel build threads must be disabled, to ensure that unique numbers are assigned to blocks.

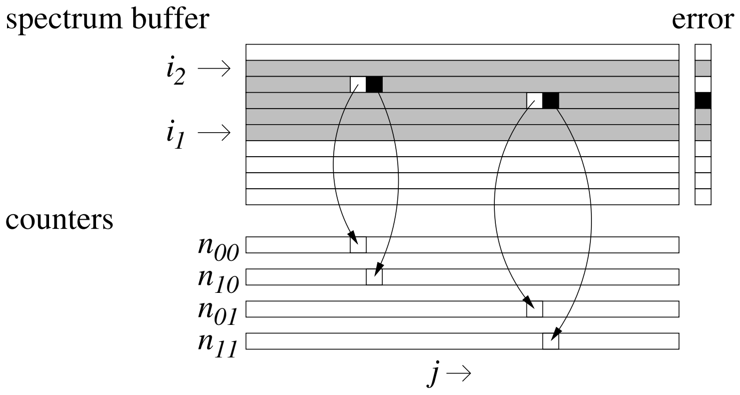


Figure 3.4 Data structures

- The possibilities for incremental builds are limited because we have to set a maximum number of blocks.
- In addition to constraints on the available memory, discussed in the previous section, CPU time is also a scarce resource, and the TV520 architecture imposes various timing constraints on different activities.

Regarding the last point, the sorting task that is involved in ranking the approximately 150,000 blocks does not violate any of the timing constraints. Although the block-level instrumentation noticeably slows down the operation of the TV, enough CPU idle time is available to support the extra load on the MIPS. This is not the case for the proprietary CPUs though, and we have not yet found a practical solution to instrumenting the signal processing software.

The spectrum bookkeeping illustrated in Figure 3.4 is implemented in a small software component that is added to the control software. Communication with this component is via standard I/O, using a PC and terminal emulator connected to the TV set. On the terminal we can enter commands such as

- start a new run, and mark the previous run as passed,
- start a new run, and mark the previous run as failed,
- select a particular similarity coefficient,
- calculate the diagnosis, and print the n locations at the top of the ranking,

where we consider a "run" to be any given period of activity of the system. We used the Jaccard and Ochiai coefficients, introduced in Chapter 2, but because of the highly accurate (manual) error detection information involved in these experiments, the diagnoses were essentially the same. This confirms the observation made in Section 2.4.3, that the performance of the former coefficient approaches that of the latter as the quality of the error detection information improves (see Figure 2.6). Because of the superior performance of these two coefficients, we have not included the Tarantula coefficient in this case study.

3.4 EXPERIMENTS

We diagnosed four problems that were encountered during the development of television sets based on the TV520 platform. In the same way as the known location of a fault can be used to evaluate the quality of the diagnosis for the benchmark experiments, the location of the repairs that were made can be used as an indication of diagnostic quality here.

Selecting the problems to use for this case study was more difficult than we anticipated when planning the experiment. Enough problem reports and repairs are available, but in many cases we could not reproduce the problem, for reasons such as

- the source tree having been removed from the version repository,
- the version of the hardware for which the problem manifested itself no longer being available,
- the problem residing in the streaming code on the proprietary CPUs, for which our tooling is not yet available, and
- the problem being hard to reproduce in itself.

In Sections 3.4.1 to 3.4.4 below we give a description of the four problems, our approach to diagnose them, and the result delivered by SFL. The quality of these diagnoses is discussed in Section 3.4.5.

3.4.1 *NVM Corrupted*

PROBLEM DESCRIPTION. The TV sets that we used in our experiments contain a small amount of non-volatile memory (NVM), whose contents are retained without the set being powered. In addition to storing information such as the last channel watched, and the current sound volume, the NVM contains several parameters of a TV's configuration, for example to select a geographical region. These parameters can be set via the so-called service menu, which is not normally accessible to the user.

A subset of the parameters stored in NVM are so important for the correct functioning of the set, that it has been decided to implement them with triple redundancy and majority voting. This provides a basic protection against memory corruption, since at least two copies of a value have to be corrupted to take effect. The problem that we analyze here entails that two of the three copies of redundant NVM parameters are not updated when changes are made via the service menu.

Approach. To diagnose this problem, we extend our diagnosis component such that once per second, it starts a new run. Knowing that the problem manifests itself in NVM, we add a consistency check on the redundant items to characterize the runs as passed or failed. The runs are taken from a simple scenario where we first activate the general menu-browsing functionality, to exonerate that part of the code. Then we make several changes to nonredundant NVM parameters, before changing the value of a redundant parameter, and performing the diagnosis based on a single failed run. The number of passed runs depends on the time to run the scenario, which is in the order of one or two minutes.

Diagnosis. In the ranking produced by SFL, 96 blocks have the highest similarity to the error vector. These blocks are in ten files, one of which is part of the NVM stack, making this the obvious place to continue the diagnosis. Inside its component, this files' functions access modules for normal, and redundant access to NVM, which confirms that the problem is in this area. The bug, however, resides in a routine that is called at system initialization time to retrieve the status (redundant, or not), of the individual NVM items to populate a table describing the NVM layout. Since this routine is always used at initialization, while the problem does not yet manifest itself, there is no way that SFL can associate it with the failures that occur later on, so in this case, the actual diagnosis is indirect at best. In general, SFL based on block-hit spectra cannot be expected to directly locate data-dependent faults, or faults in code that is always executed. However, debugging is usually an iterative process, and in this sense, zooming in on the code that accesses the table describing the NVM layout can still be seen as a valuable suggestions for where to look next.

3.4.2 *Scrolling Bug*

Problem description. The TV has several viewing modes to watch content with different aspect ratios. In 16:9 viewing mode, only part of a 4:3 image is displayed on screen, and the "window" through which the image is watched can be positioned using the directional buttons on the remote control (scrolling). The problem considered here entails that after scrolling in a vertical direction, switching to dual-picture mode and back re-centers the screen. Continuing to scroll after this re-centering has occurred makes the screen jump back to the position that it had before entering dual-picture mode, and scrolling continues from that position. It should be noted that in dual-picture mode, one of the two screen halves displays the original picture, and the other half displays teletext.

Approach. To diagnose this problem, we rerun the above scenario as follows:

1. enter 4:3 mode, and switch to dual-picture and back,
2. enter 16:9 mode, and scroll up and down,
3. demonstrate the problem in both vertical scrolling directions,
4. switch to teletext and back.

The runs are defined by the various actions such as scrolling, selecting the viewing modes, etc., leading to approximately 20 runs, two of which are marked as failed. Because we do not know where exactly the problem occurs, the two failed runs both involve two key-presses: one to switch back from dual-picture mode (which re-centers the picture), and another to scroll (which makes the picture jump).

Diagnosis. The repair of this problem involves three locations, and one of these is right on top of the ranking produced by SFL, sharing the first place with four other blocks. The second location is in the top 13 of the ranking, with the second-highest similarity, but the third location is much further down: so many other blocks have the same similarity that effectively, SFL cannot find it. However, all three fixes are in the same file, and the third fix is a natural extension of the other two.

Given the small number of locations that have to be examined before we hit two of the locations where this problem is repaired, we consider this diagnosis quite accurate. However, the last step of the scenario, where we exonerate the teletext functionality, appears to be essential for getting a good result consistently. Why the first step of the scenario, which also activates teletext in one of the two screen halves, is not sufficient, is still subject of further investigation.

3.4.3 *Pages Without Visible Content*

Problem description. In the particular product line where this problem manifests itself, it is possible to highlight a word on a teletext page, and then search the whole database of teletext pages for the current channel for other occurrences of that word. However, the teletext standard provides for pages with invisible content, through which, for example, certain control messages can be broadcast: the characters are there, but a special flag marks them invisible to the user. The problem that we investigate here entails that the word search function also finds occurrences of a word on invisible pages, and that hitting such an occurrence locks up the search functionality. In addition, attempting to highlight a word on an invisible page locks up the entire teletext functionality until the user returns to TV mode.

Approach. To diagnose this problem we use a scenario where we activate the relevant teletext browsing functionality, including the word search, and where we start new runs after, for example, changing the page, navigating to words of interest, and finding new occurrences of those words. We manually mark runs as passed or failed depending on whether the TV enters the locked-up state, or not. In the end, we could not improve the diagnosis by using more than a single failed run, and around 10 passed runs.

Diagnosis. Because this particular problem is still under investigation at NXP, it is not possible to evaluate the quality of the diagnosis based on the locations of the fixes. However, several code locations at the top of the ranking generated by SFL involve statements whose execution depend on whether a page contains invisible content. According to NXP developers, this would serve as a reminder that pages can have invisible content, and that this information provides a good suggestion on the nature of a possible fix.

3.4.4 *Repeated Tuner Settings*

Problem description. Some broadcasters' signals contain regional information in a protocol that is recognized by many television sets, and which specifies, for example, a preferred order for the television channels. The problem that we investigate here entails that after an installation (finding all channels) is performed in presence of this regional information, tuning twice in a row to an analog signal at the same frequency results in a black screen.

Approach. There are two ways in which the same frequency can be set repeatedly: by entering the same channel number on the remote-control twice, and by switching from an analog channel to an external video source (which does not change the tuner frequency) and back. We run a scenario where we demonstrate the problem in both ways, on both a single-digit and a two-digit channel, and where we also include several examples of changing the channel without triggering the problem. The general strategy is to start a new run after each channel change, and to mark the previous run as passed or failed depending on whether the problem manifests itself, or not, resulting in 4 failed runs, and depending on the exact scenario, around 15 passed runs.

Diagnosis. The repairs for this problem involve modifications in 13 code blocks, all in the same file. Although none of the exact locations appears at a high position in the ranking generated by SFL, depending on the exact scenario, typically 11 other blocks are found at the highest level of similarity, 10 of which are from the file where the problem has been repaired, making this the obvious place to start debugging. Given the fact that over 1,800 C files are involved in the build, with approximately a dozen files related to low-level tuner functionality, this can be considered a reasonably accurate diagnosis yet. We have not been able to exploit the information that the problem only occurs after an installation in presence of the regional information.

3.4.5 *Evaluation*

These experiments demonstrate that the integration of SFL in an industrial software development process is feasible: although much more time was in-

Case	estimated q_d	inspect
NVM corrupt	99.96 %*	96 blocks, 10 files
Scrolling bug	> 99.99 %	5 blocks
Invisible pages	> 99.99 %	12 blocks
Tuner problem	99.97 %	2 files

* indirect, see Section 3.4.1

Table 3.1 Diagnostic accuracy for the industrial test cases; total numbers of blocks and files are 150,000 and 1,800, respectively

vested in these experiments, the estimated costs for an analysis are 2 hours for building the application binary of the control software with instrumentation enabled, plus another few hours for running the experiments, and analyzing the data. If automated error detection is required, as we described for the NVM corruption in Section 3.4.1, some more time must be reserved for writing the special-purpose error detectors. In any case, running the analysis within a working day is feasible, and in some debugging scenario's this can be a sensible investment. In addition to that, opportunities for integrating SFL with automated testing schemes still have to be explored.

Of the four cases that we have considered thus far, one diagnosis is quite good (the scrolling bug), and in the other three cases, SFL provides a useful suggestion, where in the case of the NVM corruption, this suggestion is indirect because of the data dependencies involved. In Table 3.1 we give an estimate of the quality of the diagnosis in terms of q_d, as defined in Section 2.2.3. For the NVM corruption, this is based on the 96 blocks and 10 files on top of the ranking, as described in Section 3.4.1. For the scrolling bug we use the highest-ranking location where a repair has been made. In case of the teletext lock-up at invisible pages, we use the rank of the block that directs our attention to the flag for invisible content. For the tuner problem, the estimate is based on two out of approximately 1,800 files, instead of blocks, as discussed in Section 3.4.4. Because of the high percentages involved, we have also included an indication of the amount of code that must be investigated, based on the number of blocks with an equal or higher calculated similarity coefficient.

While the estimates in Table 3.1 are debatable, the experiments demonstrate that spectrum-based fault localization scales well, and that it can be applied as a practical tool in industrial software development. Note that while the estimated q_d values in Table 3.1 clearly indicate the power of SFL on large codes, these numbers are not indicative for the added value for an experienced developer. For example, as we discussed in Section 3.4.4, an NXP developer would immediately concentrate on the dozen of files related to low-level tuner functionality, lowering q_d to just over 95% of files that do not have to be investigated. Nevertheless, SFL confirms such a decision as well as improves on it in terms of q_d.

3.5 SUMMARY

In this chapter we have demonstrated software fault diagnosis though the analysis of program spectra (SFL) on two large-scale industrial test cases in the area of embedded software in consumer electronics devices. In particular, we showed that SFL is well suited for resource-constrained environments due to its low time/space complexity. Moreover, our experiments indicate that the diagnostic accuracy of SFL leads the developer quickly to the fault. In comparison with the effectiveness results obtained in the previous chapter, we conclude that the large the system under analysis is the better the diagnostic accuracy, even if only a few failures are observed. Furthermore, we are not aware that any of the other techniques mentioned above have successfully been applied for diagnosing software faults in resource-constrained systems.

While our current experiments focus on development-time debugging, they open corridors to further applications, such as run-time recovery by rebooting only those parts of a system whose activities correlate with detected errors [Sözer, 2009].

Using Fault Screeners for Error Detection

4

ABSTRACT

Despite extensive testing in the development phase, residual defects can be a great threat to dependability in the operational phase. This chapter studies the utility of low-cost, generic invariants ("screeners") in their capacity to act as error detectors. Fault screeners are simple software (or hardware) constructs that detect variable value errors based on unary invariant checking. In this chapter we evaluate and compare the performance of three low-cost screeners (Bloom filter, bitmask, and range screener) that can be automatically integrated within a program, and trained during the testing phase. We compare the performance of the fault screeners with test oracles in terms of false positives and negatives, and show that "ideal"-screeners (e.g., screeners that store each individual value during training) are slower learners than simple screeners (e.g, range screeners), but have less false negatives. We present a novel analytic model that predicts the false positive and false negative rate for ideal and simple screeners. We show that the model agrees with our empirical findings. Furthermore, we describe an application of the screeners, where the screener's error detection output is used as input to a fault localization process that provides automatic feedback on the location of residual defects during operation in the field.

//

In many domains such as consumer products the residual defect rate of software is considerable, due to the trade-off between reliability on the one hand and development cost and time-to-market on the other. Proper *error detection* is a critical factor in successfully recognizing, and coping with (recovering from) failures during the *deployment phase* [Patterson et al., 2002, Kephart and Chess, 2003, Sözer, 2009]. Even more than during testing at the *development phase*, errors may otherwise go unnoticed, possibly resulting in catastrophic failure later on.

Error detection is typically implemented through tests (invariants) that usually trigger some exception handling process. The invariants range from *application-specific* (e.g., a user-programmed test to assert that two state variables in two different components are in sync) to *generic* (e.g., a compiler-generated value range check). While application-specific invariants cover many failures anticipated by the programmer and have a low false positive and false negative rate (see Table 4.1 for a summary of the relation between warnings and

	Error	No error
Warning	true positive	false positive
No warning	false negative	true negative

Table 4.1 Relationship between error vs. warning

errors), their (manual) integration within the code is typically a costly, and error-prone process. Despite the simplicity of generic invariants, and their higher false positive and false negative rates, they can be *automatically* generated within the code, while their application-specific *training* can also be *automatically* performed as integral part of the testing process during the development phase. Furthermore, generic invariants correlate to some extent with application-specific invariants. Consequently, violation of the latter is typically preluded by violation of the former type [Ernst et al., 2001].

In view of the above, attractive properties, generic invariants, often dubbed fault screeners, have long been subject of study in both the software and the hardware domain (see Section 4.5). Examples include value screeners such as simple bitmask [Hangal and Lam, 2002, Racunas et al., 2007] and range screeners [Hangal and Lam, 2002, Racunas et al., 2007], and more sophisticated screeners such as Bloom filters [Hangal and Lam, 2002, Racunas et al., 2007]. In previous work, screeners are used for fault localization [Hangal and Lam, 2002, Pytlik et al., 2003], albeit with limited success [Pytlik et al., 2003]. Also from this perspective, feeding their output to a specific (spectrum-based) fault localization algorithm increases diagnostic quality.

This chapter studies the utility of low-cost, generic error detectors ("screeners") as input to SFL in the operational phase. The motivation for this study is the following. In Chapter 2 it has been established that the diagnostic accuracy of SFL is not very sensitive to error detection quality, provided that the number of (test) runs (program execution profile information) is not too small. As this insensitivity especially applies to the false negatives (a weak point of low-cost screeners, as shown in the paper), low-cost screeners may already yield acceptable diagnostic accuracy. As low-cost and generic error detectors would:

- avoid costly programmer involvement, and
- minimize run-time time/space overhead, the combination screening-SFL seems an appealing prospect in the context of dependable (embedded) software development.

In this chapter we analytically and empirically investigate the performance of screeners. In particular, we make the following contributions:

- We develop a simple, approximate, analytical performance model that predicts the false positive and false negative rates in terms of the variable domain size and training effort. We derive a model for (ideal) screeners that store each individual value during training, one for bitmask screeners that express all observed values in terms of a bit array, and another model for range screeners that compress all training information in terms of a single range interval.

- We evaluate the performance of Bloom filters, bitmask, and range screeners based on instrumenting them within the Siemens benchmark suite, which comprises a large set of program versions, of which a subset is seeded with faults. We show that our empirical findings are in agreement with our model.

- As a typical application of screeners, we show how the Bloom filter, bitmask, and range screeners are applied as input for automatic fault localization, namely spectrum-based fault localization (SFL). It is shown that the resulting fault localization accuracy is comparable to one that is traditionally achieved at the design (testing) phase, namely for either Bloom filter or range screeners.

- Since program invariants violations can occur in other locations than the faulty one, we argue that the screener-SFL combination gives more educated guesses to find the faulty locations than the stand-alone screeners, such as in [Pytlik et al., 2003].

The significance of the empirical results is that no costly, application-specific modeling is required for diagnostic purposes, paving the way for truly automatic program debugging. Our main findings show that although the error detection quality of screeners is quite limited, the diagnostic quality of SFL using range screeners in the operational phase can match the quality of SFL based on test cases in the development phase, provide sufficient test cases are available for training (hundreds of runs).

The remainder of this chapter is organized as follows. In the next section we introduce the Bloom filter, bitmask and range screeners. In Section 4.2 the experimental setup is described and the empirical results are discussed. Section 4.3 presents our analytical performance model to explain the experimental results. The application of screeners as input for SFL is discussed in Section 4.4. A comparison to related work appears in Section 4.5. Section 4.6 summarizes the chapter.

4.1 FAULT SCREENERS

Program invariants, first introduced by Ernst *et al.* [Ernst et al., 2001] with the purpose of supporting program evolution, are conditions that have to be met by the state of the program for it to be correct. Many kinds of program invariants have been proposed [Ernst et al., 2001, Ernst et al., 2007, Racunas et al., 2007]. We focus on dynamic range invariants [Racunas et al., 2007], bitmask invariants [Hangal and Lam, 2002, Racunas et al., 2007], and Bloom filter invariants [Racunas et al., 2007]. Besides being generic, they require minimal overhead (lending themselves well for application within resource-constrained environments, such as embedded systems).

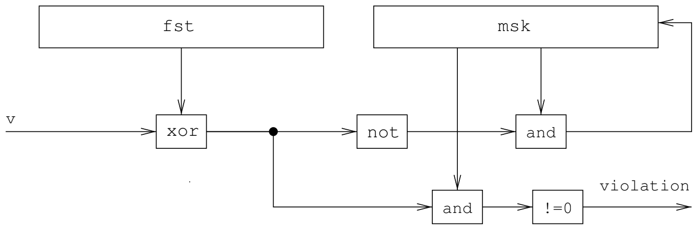


Figure 4.1 Bitmask invariant block diagram

A *bitmask invariant* (see Figure 4.1) is composed of two fields: the first observed value (*fst*) and a bitmask (*msk*) representing the activated bits (initially all bits are set to 1). Every time a new value v is observed, it is checked against the currently valid *msk* according to:

$$violation = (v \oplus fst) \wedge msk \tag{4.1}$$

where $\oplus$ and $\wedge$ are the bitwise *xor* and *and* operators respectively. If the $violation$ is non-zero, an invariant violation is reported. In error detection mode (operational phase) an error is flagged. During training mode (development phase) the invariant is updated according to:

$$msk := \neg(v \oplus fst) \wedge msk \tag{4.2}$$

An example of the life cycle of an eight-bitmask invariant can be seen in Table 4.2. During the training phase only odd values are observed, and thus the first bit is a 1 (fixed) in the bitmask. During the checking we observe an even value, and therefore a violation is reported.

Times visited	fst	msk	v	$violation$
Training				
0	-	-	0001 0101	-
1	0001 0101	1111 1111	0001 0111	0000 0010
2	0001 0101	1111 1101	0101 0001	0100 0100
3	0001 0101	1011 1001	0001 1111	0001 1000
Checking				
4	0001 0101	1011 0001	0000 0010	0000 0001

Table 4.2 The typical life cycle of a bitmask invariant

Although bitmask invariants were used with success by Hangal and Lam [Hangal and Lam, 2002], they have limitations. Their support for representing negative and floating point numbers is limited. Furthermore, the upper bound representation of an observed number is far from tight. Yet another issue is the problem with functions that can return 0 or −1. If an invariant is created with an initial value of 0 (0x00000000) and then the value −1 (0x*FFFFFFFF*) is returned in a subsequent pass, the bitmask will accept any value, even if only 0 and −1 are correct. For example, consider the return value of `strcmp()` in the following code:

```
if (strcmp(a, b) == 0) {
  printf("a = b");
} else {
  printf("a shorter than b");
}
```

During the training phase, we only observe `a` values that are shorter or equal than `b`, and thus `strcmp()` has returned 0 or −1 only. In the checking phase a value of `a` longer than `b` is used as input to the program, making `strcmp()` return 1. This should be reported as a violation, but this will not happen because the invariant is already at its most general form. To overcome these problems, we also consider *range invariants*, e.g., used by Racunas *et al.* in their hardware perturbation screener [Racunas et al., 2007].

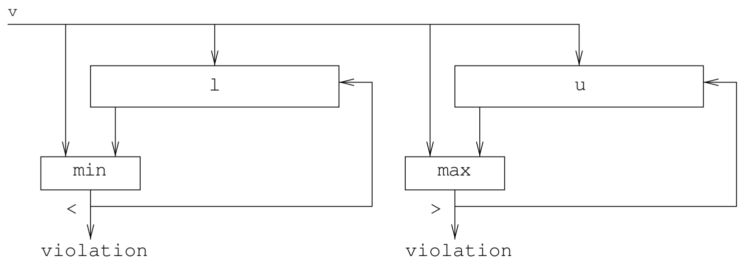


Figure 4.2 Range invariant block diagram

Range invariants (see Figure 4.2) are used to represent the (integer or real) bounds of a program variable. Every time a new value v is observed, it is checked against the currently valid lower bound l and upper bound u according to

$$violation = \neg(l < v < u) \tag{4.3}$$

If v is outside the bounds, an error is flagged in error detection mode (operational phase), whereas in training mode (development phase) the range is extended according to the assignment

$$l := \min(l, v) \tag{4.4}$$

$$u := \max(l, v) \tag{4.5}$$

Table 4.3 shows the evolution of a range invariant as values are observed. During the training values between 1 and 10 are observed. During the checking we observe a value grater than 10, and it is reported as a violation.

Times visited	l	u	v	*violation*
Training				
0	-	-	10	-
1	10	10	1	1
2	1	10	5	0
3	1	10	8	0
Checking				
4	1	10	12	1

Table 4.3 The typical life cycle of a range invariant

Bloom filters [Bloom, 1970] (see Figure 4.3) are a space-efficient probabilistic data structure used to check if an element is a member of a set. This screener is stricter than the range screeners, as it is basically a compact representation of a variable's entire history.

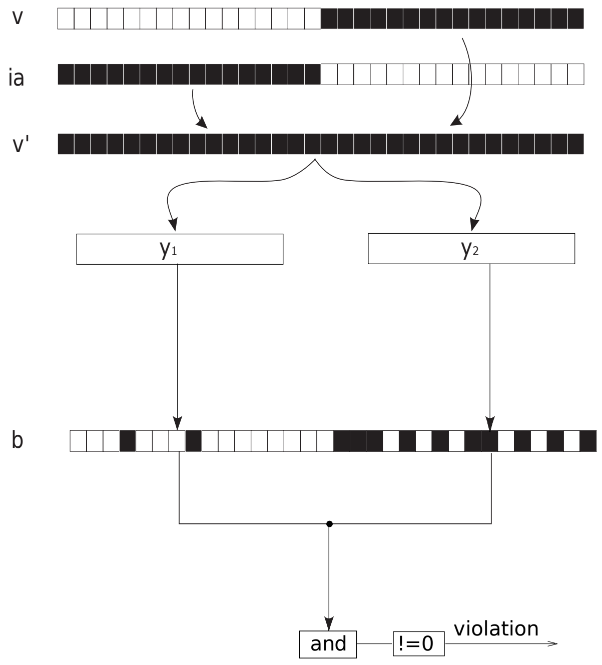


Figure 4.3 Bloom filter block diagram

All variables share the same Bloom filter, which is essentially a bit array ($64KB$, the size of the filter could be decreased by using a backup filter to prevent saturation [Racunas et al., 2007]). Each 32-bit value v and instruction address ia are merged into a single 32-bit number v':

$$v' = (v * 2^{16}) \vee (0xFFFF \wedge ia) \tag{4.6}$$

where $\vee$ and $\wedge$ are bitwise operators, respectively. This number v' is used as input to two hash functions [Knuth, 1997] (y_1 and y_2, see Appendix A for details on the hash functions used), which index into the Bloom filter b. In detection mode an error is flagged according to (see Figure 4.4)

$$violation = \neg(b[y_1(v')] \wedge b[y_2(v')]) \tag{4.7}$$

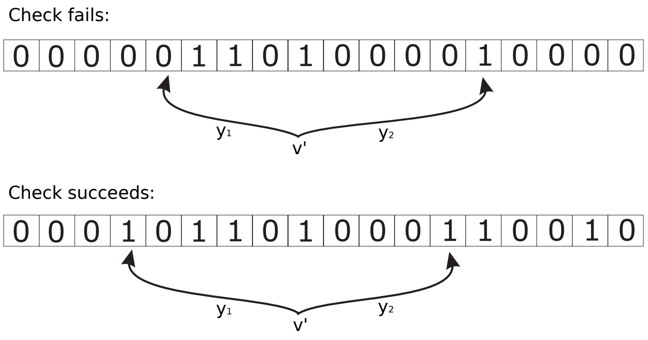


Figure 4.4 Checking whether value v' is in the bloom filter

During training mode, the outputs of the hash functions are used to update the Bloom filter according to the assignment (see Figure 4.5)

$$b[y_1(v')] := 1 \tag{4.8}$$
$$b[y_2(v')] := 1 \tag{4.9}$$

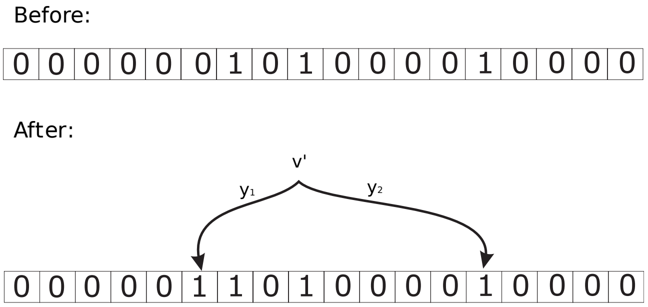


Figure 4.5 Adding a new value v' to the Bloom filter

An example of the life cycle of an 8-bit Bloom filter can be seen in Table 4.4. During the training phase we observe the values 1 and 2, which updated the Bloom filter as in the table. During the checking phase, value 2 is observed, and since it has not been observed before a violation is reported.

Times visited	y_1	y_2	b	*violation*
Training				
0	-	-	0000 0000	-
1	1	8	1000 0001	1
2	3	5	1010 1001	1
Checking				
3	3	6	1010 1001	1

Table 4.4 The typical life cycle of a Bloom filter invariant

Program	Faulty Versions	LOC	Test Cases	Description
`print_tokens`	7	539	4130	Lexical Analyzer
`print_tokens2`	10	489	4115	Lexical Analyzer
`replace`	32	507	5542	Pattern Recognition
`schedule`	9	397	2650	Priority Scheduler
`schedule2`	10	299	2710	Priority Scheduler
`tcas`	41	174	1608	Altitude Separation
`tot_info`	23	398	1052	Information Measure

Table 4.5 Set of programs used in the experiments

4.2 EXPERIMENTS

In this section the experimental setup is presented, namely the benchmark set of programs, the workflow of the experiments, and the evaluation metrics. Finally, the experimental results are discussed.

4.2.1 *Experimental Setup*

Benchmark set In our study, we use a set of test programs known as the *Siemens set* [Hutchins et al., 1994]. The Siemens set is composed of seven programs. Every single program has a correct version and a set of faulty versions of the same program. The correct version can be used as reference version. Each faulty version contains exactly one fault. Each program also has a set of inputs that ensures full code coverage. Table 4.5 provides more information about the programs in the package (for more information see [Hutchins et al., 1994]). Although the Siemens set was not assembled with the purpose of testing fault diagnosis and/or error detection techniques, it is typically used by the research community as the set of programs to test their techniques.

In total the Siemens set provides 132 programs. However, as no failures are observed in two of these programs, namely version 9 of `schedule2` and version 32 of `replace`, they are discarded. Besides, we also discard versions 4 and 6 of `print_tokens` because the faults in these versions are in global variables and the profiling tool used in our experiments does not log the execution of these statements. In summary, we discarded 4 versions out of 132 provided by the suite, using 128 versions in our experiments.

Workflow of Experiments Our approach to study the performance of fault screeners as error detectors in the operational phase comprises three stages. First, the target program is instrumented to generate program spectra (used by the fault localization technique, see Section 4.4) and to execute the invariants (see Figure 4.6). To prevent faulty programs to corrupt the logged information, the program invariants and spectra themselves are located in an external component ("Screener"). The instrumentation process is implemented as an optimization pass for the LLVM tool [Lattner and Adve, 2004] in C++ (for details on the instrumentation process see [González, 2007]). The program points screened are all memory loads/stores, and function argument and return values.

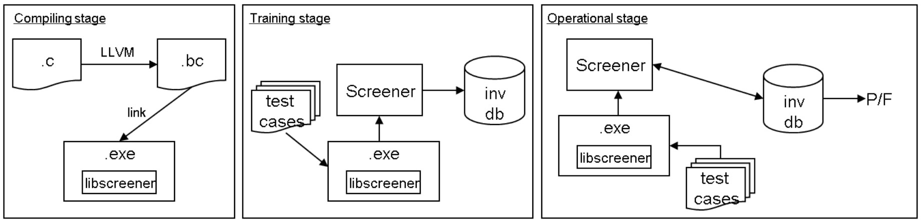


Figure 4.6 Workflow of experiments

Second, the program is run for those test cases for which the program passes (its output equals that of the reference version), in which the screeners are operated in training mode. The number of (correct) test cases used to train the screeners is of great importance to the performance of the error detectors at the operational (detection) phase. In the experiments this number is varied between 5% and 100% of all correct cases (134 and 2666 cases on average, respectively) in order to evaluate the effect of training.

Finally, we execute the program over all test cases (excluding the previous training set), in which the screeners are executed in detection, operational mode.

Error Detection Evaluation Metrics We evaluate the error detection performance of the fault screeners by comparing their output to the pass/fail outcome per program over the entire benchmark set. The ("correct") pass/fail information is obtained by comparing the output of the faulty program with the reference program.

Let N_P and N_F be the size of the set of passed and failed runs, respectively, and let F_p and F_n be the number of false positives and negatives, respectively. We measure the false positive rate f_p and the false negative rate f_p according to

$$f_p = \frac{F_p}{N_P} \tag{4.10}$$

$$f_n = \frac{F_n}{N_F} \tag{4.11}$$

4.2.2 *Results*

Figure 4.7 plots f_p and f_n in percents for bitmask (*msk*), range (*rng*), and Bloom filter (*blm*) screeners for different percentages of (correct) test cases used to train the screeners, when instrumenting all program points in the program under analysis. The plots represent the average over all programs, which has negligible variance (between $0 - 0.2\%$ and $3 - 5\%$, for f_p and f_n, respectively). From the figure, the following conclusions can be drawn for f_p: the more test cases used to train the screeners, the lower f_p (as screeners evolve with the learning [1] process). In addition, it can be seen that Bloom filter screeners learn slower than the range screener, which in turn learn slower than bitmask screeners. Furthermore, for all screeners f_n rapidly increases, meaning that even after minimal training many errors are already tolerated. This is due to:

- limited detection capabilities: only either single upper/lower bounds or a compact representation of the observed values are stored, i.e., simple and invariants, in contrast to the host of invariants conceivable, based on complex relationships between multiple variables (typically found in application-specific invariants)
- program-specific properties: certain variables exhibit the same values for passed and failed runs, see Section 4.3. Those cases lead to false negatives.
- limited training accuracy: although the plots indicate that the *quantity* of pass/fail training input is sufficient, the *quality* of the input is inherently limited. In a number of cases a (faulty) variable error does not result in a failure (i.e., a different output than the correct reference program). Consequently, the screener is trained to accept the error, thus limiting its detection sensitivity.

Due to its strictness, Bloom filter screeners have on the one hand lower f_n than range screeners. On the other, this strictness increases f_p. In the next section we provide a more theoretic explanation for the observed phenomena.

Because of their simplicity, the evaluated screeners entail minimal computational overhead. On average, the 494 (0.40 cov[2]) program points screened introduced an overhead of 14.2% (0.33% cov) for the range screener, and 46.2% (0.15% cov) was measured for the Bloom filter screener (when all program variable loads/stores and function argument/returns are screened).

4.3 ANALYTIC MODEL

In this section we present our analytic screening performance model. First, we derive some main properties that apply without considering the particular

[1]For, e.g., range screeners, "generalization" is a more appropriate term. However, we shall use "learning" to be more compatible with related work

[2]Coefficient of variance (standard deviation divided by mean).

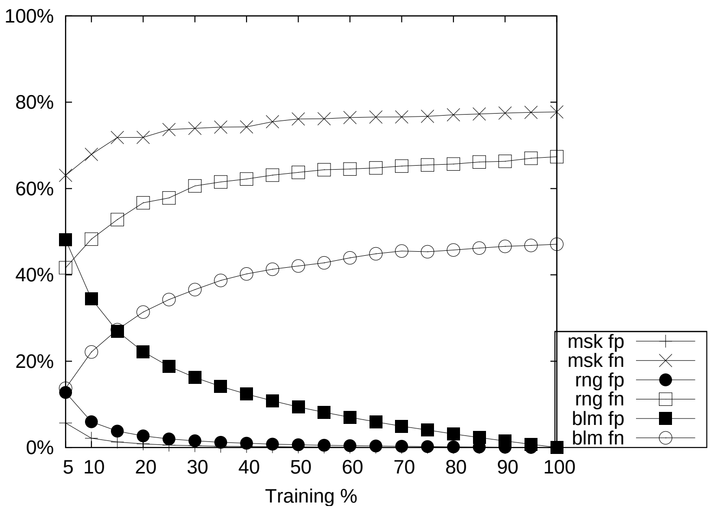


Figure 4.7 False positives and negatives on average

properties that (simple) screeners exhibit. Next we present a performance model for the bitmask screening. Finally, we focus on the range screener, which is a typical example of a simple, yet powerful screener, and which is amongst the screeners evaluated.

4.3.1 *Concepts and Definitions*

Consider a particular program variable x. Let P denote the set of values x takes in all N_P passing runs, and let F denotes the set of values x takes in all N_F failing runs. Let T denote the set of values recorded during training. Let $|P|, |F|, |T|$ denote the set sizes, respectively. Screener performance can generally be analyzed by considering the relationship between the three sets P, F, and T as depicted in Fig. 4.8.

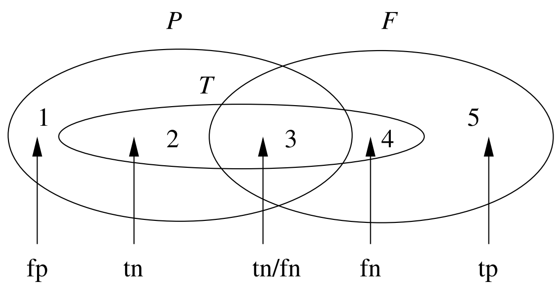


Figure 4.8 Domain of variable x

In the figure we distinguish between five regions, numbered 1 through 5, all of which associate with false positives (fp), false negatives (fn), true positives (tp), and true negatives (tn). For example, values of x which are within P (i.e., OK values) but which are (still) outside of the training set T, will trigger a false positive (region 1). Region 3 represents the fact that certain values of x may occur in both passing runs, as well as failing runs, leading to potential false negatives. Region 4 relates to the fact that for many simple screeners the update due to training with a certain OK value (e.g., in region 2) may also lead to acceptance of values that are exclusively associated with failed runs, leading to false negatives (e.g., an upper bound 10, widened to 15 due to $x = 15$, while $x = 13$ is associated with a failed run).

4.3.2 *Ideal Screening*

In the following we derive general properties of the evolution of the false positive rate f_p and the false negative rate f_n as training progresses. Assuming that all values exercised by the test cases are independent and uniformly distributed, for each new value of x in a passing run the probability p that x represents a value that is not already trained equals

$$p = \frac{|P| - |T|}{|P|} = 1 - \frac{|T|}{|P|} \tag{4.12}$$

Note that for ideal screeners region 4 does not exist. Hence T grows entirely within P. Consequently, the expected growth of the training set is given by

$$t_k - t_{k-1} = p_{k-1} \tag{4.13}$$

where t_k denotes the expected value of $|T|$, $\mathsf{E}[|T|]$, at training step k, and p_k denotes the probability that a value is not already trained p at step k. It follows that t_k is given by the recurrence relation

$$t_k = \alpha \cdot t_{k-1} + 1 \tag{4.14}$$

where $\alpha = 1 - 1/|P|$. The solution to this recurrence relation is given by

$$t_k = \frac{\alpha^k - 1}{\alpha - 1} \tag{4.15}$$

Consequently

$$\mathsf{E}[|T|] = |P| \cdot \left(1 - (1 - \frac{1}{|P|})^k\right) \tag{4.16}$$

Thus the fraction of T within P initially increases linearly with k, approaching P in the limit for $k \to \infty$.

Since in detection mode the false positive rate f_p equals p, from (4.12) it follows

$$f_p = (1 - \frac{1}{|P|})^k \tag{4.17}$$

Thus the false positive rate decreases with k, approaching a particular threshold after a training effort k that is (approximately) *proportional* to $|P|$. As the false negative rate is proportional to the part of T that intersects with F (region 3) it follows that f_n is proportional to the growth of T according to

$$f_n = f \cdot \left(1 - (1 - \frac{1}{|P|})^k\right) \tag{4.18}$$

where f denotes the fraction of P that intersects with F. Thus the false negative rate increases with k, approaching f in the limit when T equals P. From the above it follows

$$f_n = f \cdot (1 - f_p) \tag{4.19}$$

4.3.3 *Bitmask Screening*

In the following we introduce the constraint that the entire value domain of a variable is compressed in terms of a bitmask. Let *msk* be a bit array with a indices. Without loss of generality, let $p_i = p$ be the probability that the bit in index i equals 0 after a value is observed. The expected number H of indices set to 1 (aka Hamming weight) after that observation follows a binomial distribution, and amounts to $\mathsf{E}[H] = (1 - p) \cdot a$. Thus, *msk* has the following expected number of 1's after k observations

$$\mathsf{E}[H]_k = (1 - p^k) \cdot a \tag{4.20}$$

Consequently,

$$\mathsf{E}[|T|]_k = 2^{(1-p^k)\cdot a} \tag{4.21}$$

Note that every time a bit is flipped in *msk*, the number of accepted values doubles. From (4.12) it follows that

$$f_p = 1 - \frac{2^{(1-p^k)\cdot a}}{|P|} \tag{4.22}$$

Thus the false positive rate decreases exponentially with k, approaching a particular threshold after a training effort k that is (approximately) *proportional* to $|P|$. The analysis of f_n is similar to the previous section with the modification that for simple screeners such as the bitmask screener the fraction f' of T that intersects with F is generally greater than the fraction f for ideal screeners (regions 3 and 4, as explained earlier). Thus,

$$f_n = f' \cdot (1 - f_p) = f' \cdot \frac{2^{(1-p^k)\cdot a}}{|P|} > f \cdot \frac{2^{(1-p^k)\cdot a}}{|P|} \tag{4.23}$$

4.3.4 *Range Screening*

In the following we introduce the constraint that the entire value domain of variable x available for storage is compressed in terms of only one range, coded in terms of two values l (lower bound) and u (upper bound). Despite the potential negative impact on f_p and f_n we show that the training effort required for a particular performance is *independent* of the entire value domain, *unlike* the two previous screeners.

After training with k values, the range screener bounds have evolved to

$$l_k = \min_{i=1,\dots,k} x_i \tag{4.24}$$

$$u_k = \max_{i=1,\dots,k} x_i \tag{4.25}$$

Since x_i are samples of x, it follows that l_k and u_k are essentially the lowest and highest *order statistic* [David and Nagaraja, 1970], respectively, of the sequence of k variates taken from the (pseudo) random variable x with a particular probability density function (pdf). The order statistics interpretation allows a straightforward performance analysis when the pdf of x is known. In the following we treat two cases.

Uniform Distribution

Without loss of generality, let x be distributed according to a uniform pdf between 0 and r (e.g., a uniformly distributed index variable with some upper bound r). From, e.g., [David and Nagaraja, 1970] it follows that the expected values of l_k and u_k are given by

$$\mathsf{E}[l_k] = \frac{1}{k+1} \cdot r \tag{4.26}$$

$$\mathsf{E}[u_k] = \frac{k}{k+1} \cdot r \tag{4.27}$$

Consequently,

$$\mathsf{E}[|T|] = \mathsf{E}[u_k] - \mathsf{E}[l_k] = \frac{k-1}{k+1} \cdot r \tag{4.28}$$

Since $|P| = r$, from (4.12) it follows ($f_p = p$) that

$$f_p = 1 - \frac{k-1}{k+1} = \frac{2}{k+1} \tag{4.29}$$

The analysis of f_n is similar to the previous section, thus, with the modification that for simple screeners such as the range screener the fraction f' of T that intersects with F is generally greater than the fraction f for ideal screeners (regions 3 and 4, as explained earlier). Hence,

$$f_n = f' \cdot (1 - f_p) = f' \cdot \frac{k-1}{k+1} > f \cdot \frac{k-1}{k+1} \tag{4.30}$$

Normal Distribution

Let x be distributed according to a normal pdf with zero mean and variance σ (many variables such as loop bounds are measured to have a near-normal distribution over a series of runs with different input sets [Gautama and van Gemund, 2006]). From, e.g., [Gumbel, 1962] it follows that the expected values of l_k and u_k are given by the approximation (asymptotically correct for large k)

$$\mathsf{E}[l_k] = -\sigma \cdot \sqrt{2 \cdot \log(0.4 \cdot k)} \tag{4.31}$$

$$\mathsf{E}[u_k] = \sigma \cdot \sqrt{2 \cdot \log(0.4 \cdot k)} \tag{4.32}$$

Consequently,

$$\mathsf{E}[|T|] = \mathsf{E}[u_k] - \mathsf{E}[l_k] = 2 \cdot \sigma \cdot \sqrt{2 \cdot \log(0.4 \cdot k)} \tag{4.33}$$

The false positive rate equals the fraction of the normal distribution (P) not covered by T. Let erf be the error function encountered in integrating the normal distribution, in terms of the normal distribution's cumulative density function (cdf) it follows

$$f_p = 1 - \operatorname{erf} \frac{\sigma \cdot \sqrt{2 \cdot \log(0.4 \cdot k)}}{\sigma \cdot \sqrt{2}} \tag{4.34}$$

which reduces to

$$f_p = 1 - \operatorname{erf} \sqrt{\log(0.4 \cdot k)} \tag{4.35}$$

Note that, again, f_p is independent of the variance of the distribution of x. For the false negative rate it follows

$$f_n = f' \cdot (1 - f_p) = f' \cdot \operatorname{erf} \sqrt{\log(0.4 \cdot k)} \tag{4.36}$$

4.3.5 *Discussion*

Both the result for uniform and normal distributions show that the use of range screeners implies that the false positive rate (and, similarly, the false negative rate) can be optimized *independent* of the size of the value domain. Since the value domain of x can be very large this means that range screeners require much less training than "ideal" screeners to attain bounds that are close to the bounds of P. Rather than increasing one value at a time by "ideal" screeners, range screeners can "jump" to a much greater range at a single training instance. The associated order statistics show that $|T|$ approaches $|P|$ regardless their absolute size. For limited domains such as in the case of the uniform pdf the bounds grow very quickly. In the case of the normal pdf the bounds grow less quickly. Nevertheless, according to the model a 1 percent false positive rate can be attained after a few thousand training runs (few hundred in the uniform case). Although bitmask screeners are dependent on the size of variable x, they learn much faster than range and "ideal" screeners. This is due to the fact that every time a bit is flipped in the bitmask, the number of accepted values doubles.

The model is in good agreement with our empirical findings (see Figure 2). While exhibiting better f_n performance, the Bloom filter suffers from a less steep learning curve (f_p) compared to the range screener, which has a higher f_p rate if compared to the bitmask screener. Although it might seem that even the Bloom filter has acceptable performance near the 100 percent mark, this is due to an artifact of the measurement setup. For 100 percent training there are no passing runs available for the evaluation (detection) phase, meaning that there will never be a (correct) value presented to the screener that it has not already been seen during training. Consequently, for the 100 percent mark f_p is zero by definition, which implies that in reality the Bloom filter is expected to exhibit still a non-zero false positive rate after 2666 test cases (in agreement with the model). In contrast, for the range/bitmask screener it is clearly seen that even for 1066 tests f_p is already virtually zero (again, in agreement with the model).

4.4 FAULT SCREENING AND SFL

In this section we evaluate the performance of the studied fault screeners as error detector input for automatic fault localization tools, in particular SFL.

To recap, in SFL program runs are captured in terms of a spectrum. A program spectrum [Harrold et al., 1998] can be seen as a projection of the execution trace that shows which parts (e.g., blocks, statements, or even paths) of the program were active during its execution (a so-called "hit spectrum"). In the context of the experiments reported in this section, we consider a program part to be a statement. Basically, diagnosis consists in identifying the part whose activation pattern resembles the occurrences of errors in different executions. This degree of similarity is calculated using *similarity coefficients* taken from data clustering techniques [Jain and Dubes, 1988]. Amongst the best similarity coefficients for SFL is the Ochiai coefficient, which was introduced in Chapter 2. The output of SFL is a ranked list of parts (program statements in the context of these experiments) in order of likelihood to be at fault.

Given that the output of SFL is a ranked list of statements in order of likelihood to be at fault, we define quality of the diagnosis q_d as

$$q_d = 1 - (\tau/(M-1))$$

where τ is the position of the faulty statement in the ranking, and M the total number of statements. Hence q_d measures the number of statements that need not be inspected when following the ranking in searching for the fault. If there are more statements with the same coefficient, τ is then the average ranking position for all of them (see Section 2.2.3 of Chapter 2 for a more elaborate definition).

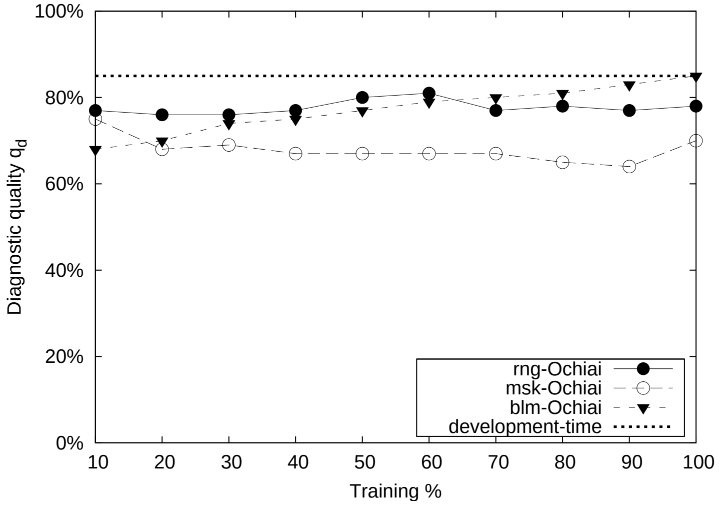


Figure 4.9 Diagnostic quality q_d on average

Figure 4.9 plots q_d for SFL using the three screeners versus the training percentage as used in Figure 4.7. In addition, we also plot the diagnostic accuracy yielded by SFL at development-time, i.e., using a reference program to flag runs as passed or failed (similar to the experiments in Chapter 2). From the figure, we conclude that the bitmask screener is the worst performing one. In general, the performance of bloom filter and range screeners is similar. The higher f_n of the range screener is compensated by its lower f_p, compared to the Bloom filter screener. The best q_d, 81% for the range screener is obtained for 50% training, whereas the Bloom filter screener has its best 85% performance for 100% (although this is due to an artifact of the measurement setup as explained in the Section 4.3.5). From this, we can conclude that, despite its slower learning curve, the Bloom filter screener can outperform the range screener if massive amounts of data are available for training (f_p becomes acceptable). On the other hand, for those situations where only a few test cases are available, it is better to use the range screener. Comparing the screener-SFL performance with SFL at development-time (85% on average [Abreu et al., 2006a], see Figure 4.10), we conclude that the use of screeners in an operational (operational) context yields comparable diagnostic accuracy to using pass/fail information available in the testing phase. As shown in [Abreu et al., 2008a] this is due to the fact that the quantity of error information compensates the limited quality (in particular, the false negative rate).

Due to their small overhead, fault screeners are attractive for being used as error detectors. A way to reduce the overhead is to carefully select which

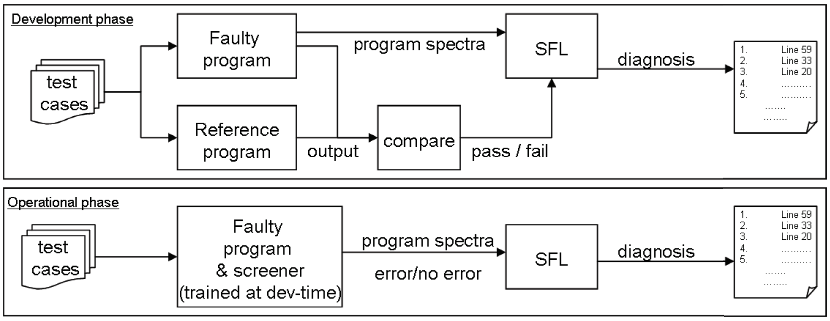


Figure 4.10 Screener-SFL vs. reference-based SFL

program points to instrument (e.g., currently we also store invariants for constants, but they do not give any relevant info - hence, they could be discarded). To obtain an indication of the potential improvement we have also varied the number of range screeners by considering ld/st and arg/ret points separately (as they outperform bitmasks we only consider ranges).

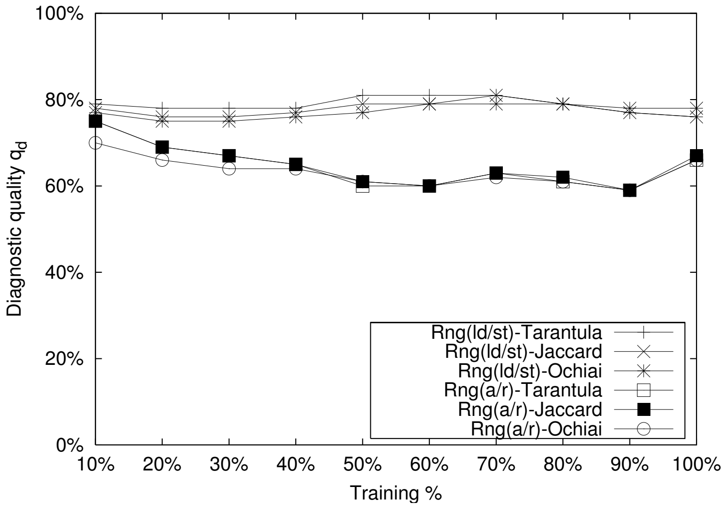


Figure 4.11 Diagnostic quality q_d for only either function arguments/returns (a/r) or loads/stores (ld/st)

Figure 4.11 shows q_d based on screening either ld/st points or arg/ret points. For simplicity, we only plot the results using range screeners as they yield better results than the other screeners. For many training situations, the (average 393) ld/st screeners approach achieves similar performance to total screening, whereas the (average 101) arg/ret screeners entail a drop in diagnostic performance. Figure 4.12 plots f_n/f_p for either ld/st points or arg/ret points, refer to Figure 4.7 for a comparison to arg/ret and ld/st screening.

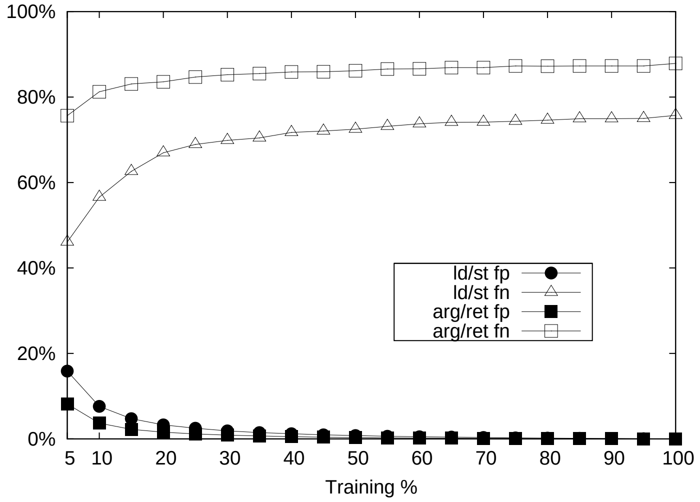


Figure 4.12 False positives and negatives on average for range screeners for only either function arguments/returns (a/r) or loads/stores (ld/st)

Table 4.6 summarizes the trade-off between diagnostic performance and overhead, where we report the training percentage that delivers the best results.

	arg/ret	ld/st	arg/ret and ld/st
# Program Points	101 ± 72.3	393 ± 172	494 ± 203.6
Overhead [%]	4.8 ± 1.9	11.6 ± 4.6	14.3 ± 5.5
q_d (training)	75% (10%)	81% (50%)	81% (50%)

Table 4.6 Range screener density vs. Performance

4.5 RELATED WORK

Dynamic program invariants have been subject of study by many researchers for different purposes, such as program evolution [Ernst et al., 2001, Ernst et al., 2007, Yang and Evans, 2004], fault detection [Racunas et al., 2007], and fault localization [Hangal and Lam, 2002, Pytlik et al., 2003]. More recently, they have been used as error detection input for fault localization techniques, namely SFL [Abreu et al., 2008a].

Daikon [Ernst et al., 2007] is a dynamic and automatic invariant detector tool for several programming languages, and built with the intention of supporting program evolution, by helping programmers to understand the code. It stores program invariants for several program points, such as call parameters, return values, and for relationships between variables. Examples of stored invariants are constant, non-zero, range, relationships, containment, and ordering. Besides, it can be extended with user-specified invariants. Carrot [Pytlik et al., 2003] is a lightweight version of Daikon, that uses a smaller set of invariants (equality, sum, and order). Carrot tries to use program invariants to pinpoint the faulty locations directly. Similarly to our experiments, the Siemens set is also used to test Carrot. Due to the negative results reported, it has been hypothesized that program invariants alone may not be suitable for debugging. DIDUCE [Hangal and Lam, 2002] uses dynamic bitmask invariants for pinpointing software bugs in Java programs. Essentially, it stores program invariants for the same program points as in this chapter. It was tested on four real world applications yielding "useful" results. However, the error detected in the experiments was caused by a variable whose value was constant throughout the training mode and that changed in the operational phase (hence, easy to detect using the bitmask screener). In [Racunas et al., 2007] several screeners are evaluated to detect hardware faults. Evaluated screeners include dynamic ranges, bitmasks, TLB misses, and Bloom filters. The authors concluded that bitmask screeners perform slightly better than range and Bloom filter screeners. However, the (hardware) errors used to test the screeners constitute random bit errors which, although ideal for bitmask screeners, hardly occur in program variables. IODINE [Hangal et al., 2005] is a framework for extracting dynamic invariants for hardware designs. In has been shown that dynamic invariant detection can infer relevant and accurate properties, such as request-acknowledge pairs and mutual exclusion between signals.

In [Hangal and Lam, 2002, Pytlik et al., 2003], screeners were used to directly pinpoint the faulty location (in this chapter, C-code statement). However, we observed that program invariants violations can occur in other locations than the faulty one, leading the developer to inspect code that is neither the faulty line itself nor related to it (this situation led to the conclusion that program invariants may not be useful for debugging in [Pytlik et al., 2003]). The advantages of our screener-SFL approach over stand-alone screeners are therefore twofold:

1. the set of unrelated candidate statements is much smaller than for stand-alone screeners

2. the set is ranked, which further reduces the probability of inspecting unnecessary code.

Dynamic program invariants are suitable for detecting errors that occur either in the data- or control-flow of a program. However, there are errors that cannot be detected using such mechanism, such as memory leaks. To automatically detect other type of errors, the following works have been presented.

Valgrind provides a memory error screener plugin that reports leaked memory blocks in a program [Seward and Nethercote, 2005, Seward and Nethercote, 2005]. CRED [Ruwase and Lam, 2004] is a memory overflow detector that determines pointer bounds by finding their referent object in an *object table*, which is a runtime structure that collects all base addresses and size information of all static, heap, and stack objects. This work is based on a previous GCC extension [Jones and Kelly, 1997]. Eraser [Savage et al., 1997] is a tool that implements a lockset algorithm to detect race conditions. Yet another recent method to detect deadlocks is presented in [Bensalem and Havelund, 2005]. As opposed to the generic fault screeners studied in this chapter, all the work above use bug-specific fault screeners.

To the best of our knowledge, none of the previous work has analytically modeled the performance of the screeners, nor evaluated their use in an automatic debugging context.

4.6 SUMMARY

In this chapter we have analytically and empirically investigated the performance of low-cost, generic program invariants (also known as "screeners"), namely bitmask, range and Bloom-filter invariants, in their capacity of error detectors. Empirical results show that near-"ideal" screeners, of which the Bloom filter screener is an example, are slower learners than range invariants, but have less false negatives. As major contribution, we present a novel, approximate, analytical model to explain the fault screener performance. The model confirms that the training effort required by near-"ideal" screeners, such as Bloom filters, increases with the variable domain size, whereas simple screeners, such as range screeners, only require constant training effort. Despite its simplicity, the model is in agreement with the empirical findings. Finally, we evaluated the impact of using such error detectors within a fault localization approach aimed at the operational (operational) phase, rather than just the development phase. We verified that, despite the simplicity of the screeners (and therefore considerable rates of false positives and/or negatives), the diagnostic performance of SFL is similar to the development-time situation. This implies that fault diagnosis with an accuracy comparable to that in the development phase can be attained at the operational phase with no additional programming effort or human intervention.

A Bayesian Approach to SFL

5

ABSTRACT

Fault diagnosis approaches can generally be categorized into spectrum-based fault localization approaches (SFL, reasoning over abstraction of program traces), and model-based diagnosis approaches (MBD, reasoning over a model of expected behavior). Although MBD approaches are inherently more accurate than SFL approaches, they are also more complex, prohibiting their use for large software systems. In this paper, we present a framework to combine the best of both worlds, coined Barinel. The program is modeled using abstraction of program traces (as in SFL) and a Bayesian reasoning framework is used to deduce multiple-fault candidates and their probabilities (as in MBD). A distinguishing feature of Barinel is the usage of a probabilistic component model that accounts for the fact that a faulty component may fail intermittently, which is determined using maximum likelihood estimation. Experimental results on both synthetic and real software programs (multiple-fault versions of the Siemens benchmark set and `space`) indicate that our approach outperforms current spectrum-based approaches to computer-aided fault localization at a cost complexity that is comparable to SFL. In the context of single faults this superiority is confirmed by formal proof.

//

As mentioned in the Introduction, two major approaches to software fault diagnosis can be distinguished, (1) the *spectrum-based fault localization* (SFL) approach, and (2) the *model-based diagnosis* or *debugging* (MBD) approach. SFL, uses abstraction of program traces to correlate software component activity with program failures (a *statistical* approach) [Abreu et al., 2007, Gupta et al., 2005, Jones et al., 2002, Liu et al., 2006, Renieris and Reiss, 2003, Zeller, 2002]. Although statistical approaches are very attractive from complexity point of view, there is no reasoning in terms of *multiple faults* that explains all failures. Consequently, the diagnostic report is ranked in terms of single components, inherently limiting diagnostic accuracy compared to multiple-fault techniques.

MBD approaches deduce component failure through *logic reasoning* [de Kleer and Williams, 1987, Feldman et al., 2008, Feldman and van Gemund, 2006, Mayer and Stumptner, 2008, Pietersma and van Gemund, 2006, Wotawa et al., 2002] using propositional models of component behavior. An inherent, strong point of MBD is that it reasons in terms of multiple-faults, which with current defect densities and program sizes is a fact of life. Its diagnostic report contain multiple-fault candidates, providing more diagnostic information compared to the one-dimensional list in SFL. Ranking is determined in terms of (multiple) fault *probability*, a more solid basis than similarity, whereas the purpose of the latter is ranking only. While inherently more accurate than statistical approaches, the main disadvantages of reasoning approaches are

(1) the need for model generation, usually with the help of static analysis that is unable to capture dynamic data dependencies, and

(2) the exponential cost of diagnosis candidates generation, prohibiting its use for programs larger than a few hundred lines [Mayer and Stumptner, 2008].

Aimed to combine the best of both worlds, in this paper we present a novel, probabilistic reasoning approach to spectrum-based multiple fault localization. Similar to SFL, we model program behavior in terms of program spectra, abstracting from modeling specific components and data dependencies. Similar to MBD, we employ a probabilistic (Bayesian) approach to deduce multiple-fault candidates and their probabilities, yielding an information-rich diagnostic ranking. To solve the inherent exponential complexity problem, we use a novel, heuristic approach to generate the most significant diagnosis candidates only, dramatically reducing computational complexity. As a result, our approach is in the same complexity class as SFL, allowing it to be used on large, real-world programs.

A particular feature of our Bayesian approach is the use of a probabilistic component failure model that accounts for the fact that a faulty component j may still behave as expected (with *health* probability h_j), i.e., need not contribute to a program failure (intermittent fault behavior). Such an intermittency model is crucial for MBD approaches where (deterministic) component behavior is abstracted to a modeling level where particular input and output values are mapped to, e.g., ranges, as shown in [Abreu et al., 2008c, de Kleer, 2007, de Kleer, 2009]. In our approach we use a Bayesian approach to compute the component health parameters h_j from the spectrum and to derive the associated multiple-fault candidate probabilities. Compared to previous Bayesian approaches in MBD that also use intermittency models [Abreu et al., 2008c, Abreu et al., 2008d], the one described in this paper represents a significant departure. Whereas the previous approaches used approximations for h_j, in this chapter h_j and the resulting candidate probabilities are computed using a maximum likelihood estimation approach, yielding a solid, probabilistic foundation.

Results on synthetic program models show that our approach outperforms the best similarity coefficient known to date in SFL, the Ochiai coefficient, as well as other spectrum-based MBD approaches (including our last contributions [Abreu et al., 2008c, Abreu et al., 2008d]). These results confirm that Bayesian reasoning inherently delivers better diagnostic performance than similarity-based ranking. Similarly, results on real programs demonstrate that our approach outperforms all SFL approaches at a time and space complexity that is comparable to SFL.

In particular, this chapter makes the following contributions

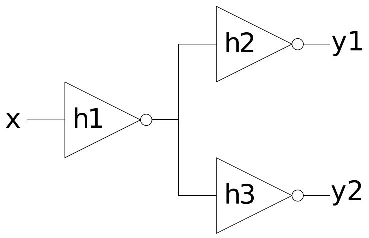
- We present our new approach for the candidate probability computation which features the algorithm to compute the h_j of all components involved in the diagnosis. The approach is coined Barinel[1], which is the name of the software implementation of our method;
- We present a diagnostic performance model for a probabilistic program model with parameters such as the number of faults, components, and test runs.
- We evaluate the inherent accuracy of the resulting diagnostic ranking using synthetic program spectra based on injected faults of which the h_j are given, and compare the accuracy with statistical approaches, as well as our previous reasoning work.
- We prove that for the single-fault case our approach is optimal. We demonstrate this result by comparing diagnostic accuracy of our approach with existing work for the (single-fault) Siemens benchmark suite;
- We compare Barinel with a large body of existing work such as Tarantula, Ochiai, and previous, approximate Bayesian approaches for the Siemens set and space, extended with multiple faults, demonstrating the superior performance of our approach and the low computation costs involved.

The chapter is organized as follows. Section 5.1 presents the basic principles of model-based diagnosis. In Section 5.2 our Bayesian reasoning approach to spectrum-based fault diagnosis is described. Our analytical performance model is presented in Section 5.3.4. The diagnostic performance on the Siemens set is evaluated in Section 5.4. We summarize this chapter in Section 5.6.

5.1 MODEL-BASED REASONING APPROACHES

In this section we briefly describe the principles underlying model-based reasoning approaches for fault localization as far as relevant to this chapter. Consider a system (in this chapter a software program) that applies some system function $\underline{y} = \mathscr{F}(\underline{x}, \underline{h})$, where $\underline{x}$ and $\underline{y}$ represent observations of system input and output, respectively, and where $\underline{h} = (h_1, \ldots, h_m)$ indicates the *health state* of the system. For each component c_j, the (binary) health states are: healthy

[1]Barinel stands for Bayesian AppRoach to dIagnose iNtErmittent fauLts. A barinel is a type of caravel used by the Portuguese sailors during their discoveries.



(a) 3-inverters circuit

```
   ($y_1$,$y_2$) 3inv(bool $x$) {
1.   $w = \neg x$
2.   $y_1 = \neg w$;
3.   $y_2 = w$; //should be $y_2 = \neg w$
     return ($y_1$,$y_2$);
   }
```

(b) Function's pseudo-code

Figure 5.1 A defective function

($h_j = 1$) or faulty ($h_j = 0$). Diagnosis can be understood as solving the inverse problem $\underline{h} = \mathscr{F}^{-1}(\underline{x}, \underline{y})$, i.e., finding the combinations of component health states that explain the observed output for a given input. Note that the internals of the system are not observable which distinguishes the diagnosis problem from a component testing problem.

To put MBD into perspective with respect to this chapter, consider the simple program function in Figure 5.1 which is composed of three inverting statements (with a fault in statement c_3), resembling a binary circuit example often used within the model-based diagnosis community (e.g., see [Pietersma and van Gemund, 2006]). The function takes one boolean input ($\underline{x} = x$), and returns two boolean outputs ($\underline{y} = (y_1, y_2)$). Each statement ($c_j$) is modeled in terms of the logical proposition

$$h_j \Rightarrow \text{lhs}_j = \neg\text{rhs}_j \tag{5.1}$$

which specifies nominal (required) behavior (information on faulty behavior could also be included in the logical proposition, but that requires more modeling / specification effort).

Given the data dependency of the program, the interconnection topology of the three inverting components is easily obtained, yielding the (combined) program model

$$h_1 \Rightarrow w = \neg x$$
$$h_2 \Rightarrow y_1 = \neg w$$
$$h_3 \Rightarrow y_2 = \neg w$$

Typically, the set of diagnostic candidates is computed using consistency-based reasoning algorithms (e.g., GDE [de Kleer and Williams, 1987], CDA* [Williams and Ragno, 2007], Safari [Feldman et al., 2008]), which determine the health states h_j such that the model is consistent with the set of observations. For instance, from the model above and the single observation $obs = ((x, y_1, y_2) = (1, 1, 0))$, it follows

$$\begin{aligned} h_1 &\Rightarrow \neg w \\ h_2 &\Rightarrow \neg w \\ h_3 &\Rightarrow w \end{aligned}$$

which equals

$$\begin{aligned} &(\neg h_1 \vee \neg w) \\ &(\neg h_2 \vee \neg w) \\ &(\neg h_3 \vee w) \end{aligned}$$

Resolution yields

$$(\neg h_1 \vee \neg h_3) \wedge (\neg h_2 \vee \neg h_3)$$

also known as *conflicts* [De Kleer et al., 1992], meaning that (1) at least c_1 or c_3 is at fault, and (2) at least c_2 or c_3 is at fault. The minimal diagnoses are the minimal hitting set [Reiter, 1987], given by

$$\neg h_3 \vee (\neg h_1 \wedge \neg h_2)$$

Thus either c_3 is at fault (single fault), or c_1 and c_2 are at fault (double fault). A number of other double faults ($\neg h_2 \wedge \neg h_3$, $\neg h_1 \wedge \neg h_3$), and a triple fault ($\neg h_1 \wedge \neg h_2 \wedge \neg h_3$), are diagnoses as well. However, they are *subsumed* by the previous two diagnoses (i.e., they not add information since c_1, c_2, c_3 are already indicted). Hence, they do not explicitly appear in the set of candidates.

Unlike statistical approaches which return all M component indices, model-based reasoning approaches only return diagnosis candidates d_k that are consistent with the observations at the price of reasoning cost. Despite this candidate reduction, the number of remaining candidates d_k is still too large in practice, and not all of them are equally probable. Hence, the computation of diagnosis candidate probabilities $\Pr(d_k)$, establishing a *ranking*, is critical to the diagnostic performance of model-based reasoning approaches. In MBD the probability that a diagnosis candidate is the actual diagnosis is computed using Bayes' rule, that updates the probability of a particular candidate d_k given new observational evidence (e.g., from a new program run).

The Bayesian probability update, in fact, can be seen as the foundation for the derivation of diagnostic candidates in any reasoning approach, i.e., (1) deducing whether a candidate diagnosis d_k is consistent with the observations, and (2) computing the posterior probability $\Pr(d_k)$ of that candidate being the actual diagnosis. Rather than computing $\Pr(d_k)$ for *all* possible candidates, just to find that most of them have $\Pr(d_k) = 0$, consistency-based algorithms are used as mentioned before, but the Bayesian probability framework remains the basis.

For each candidate the probability that it describes the actual system fault state depends on the extent to which it explains all observations. To compute the posterior probability that a candidate d_k is the true diagnosis given observation obs_i (i.e., input and expected output values) Bayes' rule is used:

$$\Pr(d_k|obs_i) = \frac{\Pr(obs_i|d_k)}{\Pr(obs_i)} \cdot \Pr(d_k|obs_{i-1}) \tag{5.2}$$

The denominator $\Pr(obs_i)$ is a normalizing term that is identical for all d_k and thus needs not be computed directly. $\Pr(d_k|obs_{i-1})$ is the prior probability of d_k. In absence of any observation, $\Pr(d_k|obs_{i-1})$ equals $\Pr(d_k) = p^{|d_k|} \cdot (1-p)^{M-|d_k|}$, where p denotes the *a priori* probability that component c_j is at fault, which in practice we set to $p_j = p$, and M denotes the number of components. $\Pr(obs_i|d_k)$ is defined as

$$\Pr(obs_i|d_k) = \begin{cases} 0 & \text{if } obs_i \wedge d_k \text{ are inconsistent} \\ 1 & \text{if } obs_i \text{ is unique to } d_k \\ \varepsilon & \text{otherwise} \end{cases} \tag{5.3}$$

As mentioned earlier, rather than updating each candidate only candidates derived from the consistency-based reasoning algorithm are updated, implying that the 0-clause need not be considered.

In model-based reasoning, many policies exist for ε [de Kleer, 2007], based on the chosen component modeling strategy. To illustrate the probability computation, consider the previous example. A well-known policy is to define $\varepsilon = 1/\#obs$ where $\#obs$ is the number of observations that can be explained by diagnosis d_k. As there are 4 possible observations that can be explained by $d_k = \{3\}$, and 8 that can be explained by $d_k = \{1,2\}$, it follows

$$\Pr(obs|\{3\}) = \frac{1}{4}$$

$$\Pr(obs|\{1,2\}) = \frac{1}{8}$$

Let $p = 0.01$ (1% prior fault probability for each c_j). It follows that

$$\Pr(\{3\}) = 0.01$$

$$\Pr(\{1,2\}) = 0.0001$$

Applying Eq. (5.2) yields

$$\Pr(\{3\}|(1,1,0)) = 0.995$$

$$\Pr(\{1,2\}|(1,1,0)) = 0.005$$

Consequently, the diagnostic report equals $D =< \{3\}, \{1,2\} >$. Thus, the most probable cause of the observed failure is c_3 being faulty. Therefore, debugging would start with the actual faulty statement.

5.2 THE BARINEL APPROACH

Our approach is characterized by the following features

1. The use of *program spectra* ((A, e) like SFL), abstracting from actual observation variables (unlike MBD);
2. The use of a *low-cost, heuristic* reasoning algorithm to extract the significant set of multiple-fault candidates d_k;
3. The use of abstract, intermittent component models, extending the binary h_j to a *real-valued* health parameter;
4. Candidate probability computation based on *maximum likelihood estimation* of h_j.

5.2.1 *Specific Features*

In the following we elaborate on each feature.

(1) Spectrum-based Reasoning

As explained in Section 5.1, model-based reasoning approaches take a model of the program and a set of observations to reason over inconsistencies between the expected and observed behavior. However, building a model of the software program can be as difficult and error prone as building the program itself. Besides, tools/approaches to automatically generate a model of a program (as in, e.g., dependency- and value-based models [Mayer and Stumptner, 2008]), are typically based on static analysis at the statement level. Consequently, dynamic information such as conditional control flow is not properly accounted for in the model. Furthermore, reasoning in terms of statement-level models do not scale to large software programs containing millions of lines. To overcome such limitations, in the same fashion as SFL we abstract from modeling the program in detail and use program spectra and pass/fail information (A, e) as the *only* dynamic source of information, from which both a model, and the input-output observations are derived. Apart from the fact that we exploit dynamic information, this approach also allows us to apply a generic component model, avoiding the need for detailed functional modeling, or relying, e.g., on invariants or pragmas for model information.

Abstracting from particular component behavior, each component c_j is modeled by the weak model

$$h_j \Rightarrow (x_j \Rightarrow y_j)$$

where h_j models the health state of c_j and x_j, y_j model its input and output variable value *correctness* (i.e., we abstract from actual variable values, in contrast to the earlier example). This weak model implies that a healthy component c_j translates a correct input x_j to a correct output y_j. However, a faulty component or a faulty input *may* lead to an erroneous output.

As each row in A specifies which components were involved, we interpret a row as a "run-time" model of the program as far as it was considered in that particular run. Consequently, A is interpreted as a sequence of typically different models of the program, each with its particular input and output correctness observation. The overall approach can be viewed as a sequential diagnosis that incrementally takes into account new program (and pass/fail) evidence with increasing N. A single row $A_{n,*}$ corresponds to the (sub)model

$$\begin{aligned}
&h_m \Rightarrow (x_m \Rightarrow y_m), \text{ for } m \in S_n \\
&x_{s_i} = y_{s_{i-1}}, \text{ for } i \geq 2 \\
&x_{s_1} = \text{true} \\
&y_{s'} = \neg e_n
\end{aligned}$$

where $S_n = \{m \in \{1,\ldots,M\} \mid a_{nm} = 1\}$ denotes the well-ordered set of component indices involved in computation n, s_i denotes the i^{th} element in this ordering, (i.e., for $i \leq j, s_i \leq s_j$), and s' denotes its last element. The resulting component chain logically reduces to

$$\bigwedge_{m \in S_n} h_m \Rightarrow \neg e_n$$

For example, consider the row ($M = 5$)

c_1	c_2	c_3	c_4	c_5	e
1	0	0	1	0	1

This corresponds to a model where components c_1, c_4 are involved. As the order of the component invocation is not given (and with respect to our above weak component model is irrelevant), we derive the model

$$\begin{aligned}
&h_1 \Rightarrow (x_1 \Rightarrow y_1) \\
&h_4 \Rightarrow (x_4 \Rightarrow y_4) \\
&x_4 = y_1 \\
&x_1 = \text{true} \\
&y_4 = \neg e_n
\end{aligned}$$

In this chain the first component c_1 is assumed to have correct input ($x_1 = \text{true}$, typical of a proper test), its output feeds to the input of the next component c_4 ($x_4 = y_1$), whose output is measured in terms of e_n ($y_4 = \neg e_n$). This chain logically reduces to

$$h_1 \wedge h_4 \Rightarrow \text{false}$$

If this were a passing computation ($h_1 \wedge h_4 \Rightarrow \text{true}$) we could not infer anything (apart from the exoneration when it comes to probabilistically rank the diagnosis candidates as explained in next section). However, as this run failed this yields

$$\neg h_1 \vee \neg h_4$$

which, in fact, is a conflict. In summary, each failing run in A generates a conflict

$$\bigvee_{m \in S_n} \neg h_m$$

As in MBD, the conflicts are then subject to a hitting set algorithm that generates the diagnostic candidates.

To illustrate this concept, again consider the example program. For the purpose of the spectral approach we assume the program to be run two times where the first time we consider the correctness of y_1 and the second time y_2. This yields the observation matrix A below

c_1	c_2	c_3	e	
1	1	0	0	obs_1
1	0	1	1	obs_2

From obs_2, it follows

$$\neg h_1 \vee \neg h_3$$

which equals the first conflict from the MBD approach discussed in the previous section, and the diagnosis trivially comprises the two single faults $\neg h_1$ and $\neg h_3$. Compared to the MBD approach, the second conflict $(\neg h_2 \vee \neg h_3)$ is missing due to the fact that no knowledge is available on component behavior and component interconnection. Although this suggests that the dynamic approach yields lower diagnostic performance, note that the example program does not have conditional control flow, and hence is ideally suitable to static analysis (MBD).

(2) Low-cost Reasoning

The observation (A, e) can be partitioned into two sets with respect to the outcome of the run. One set consists of the program spectra observed in failed runs $S_f = \{A_{i*} \mid e_i = 1\}$, and the other contains the program spectra collected in passed runs $S_p = \{A_{i*} \mid e_i = 0\}$. Essentially, failed runs indict components, whereas passed runs exonerate components. The set S_f is used to derive the set of diagnostic candidates d_k, essentially using a consistency-based algorithm (such as a minimal hitting set or minimal set cover algorithm). Instead of computing all diagnostic candidates d_k, however, (an exponential number, given the abstract modeling approach), only a subset of candidates is computed using a low-cost, heuristic reasoning algorithm called Staccato (STAtistiCs-direCted minimAl hiTing set algOrithm, see Chapter 6). Staccato not only uses just S_f but also S_p to select those candidates that are most likely to represent a significant part of the probability mass. As a result, Staccato returns a diagnostic report of limited size (typically, 100 candidates), yet capturing all significant probability mass at dramatically reduced reasoning cost.

(3) Component Intermittency Modeling

Although in traditional model-based approaches faults are typically assumed to be persistent (i.e., always induce a failure when involved), in many practical situations they manifest themselves intermittently. Consequently, in a spectrum-based context it may happen that a faulty component (statement) is involved in both failed and passed runs (e.g., different program input values). Although the component model in Eq. (5.1) allows a faulty component to exhibit correct behavior, the binary health and associated epsilon policy do not exploit the information contained in the *number* of fails or passes in which the component is involved. In order to allow us to further indict or exonerate a component as more information (runs) are available (refining the probability ranking), we model components in terms of intermittent health by extending h_j's binary definition to $h_j \in [0,1]$, where h_j expresses the probability that faulty component j produces correct output ($h_j = 0$ means persistently failing, and $h_j = 1$ essentially means healthy, i.e., never inducing failures). Consequently, for a given observation $obs_i = (A_{i*}, e_i)$, the epsilon policy in Eq. (5.3) becomes as follows

$$\varepsilon(d_k) = \begin{cases} \prod\limits_{j \in d_k \wedge a_{ij}=1} h_j & \text{if } e_i = 0 \\ 1 - \prod\limits_{j \in d_k \wedge a_{ij}=1} h_j & \text{if } e_i = 1 \end{cases} \tag{5.4}$$

Eq. (5.4) follows from the fact that the probability that a run passes is the product of the probability that each involved, faulty component exhibits correct behavior, in which we assume components fail independently (a standard assumption in fault diagnosis for tractability reasons). This epsilon policy allows us to optimally exploit indicting and exonerating information in the derivation of component health, which is a key step in deriving the diagnostic ranking.

(4) Health Probability Estimation

Before computing $\Pr(d_k)$ the h_j must be estimated from (A, e). Previous Bayesian approaches [Abreu et al., 2008c, Abreu et al., 2008d] have approximated h_j from (A, e) by computing the probability that the *combination* of components involved in a particular d_k produce a failure, instead of computing the *individual* component intermittency rate values [Abreu et al., 2008c, Abreu et al., 2008d]. Although such approaches already give significant improvement over the classical model-based reasoning (Section 5.1, see [Abreu et al., 2008c] for results), more accurate results can be achieved if the individual h_j can be determined by an exact estimator. In our approach we determine h_j per component based on their effect on the epsilon policy Eq. (5.4) to compute $\Pr(d_k)$. The key idea underlying our approach is that for each candidate d_k we compute the h_j for the candidate's faulty components that *maximizes the probability* $\Pr(e|d_k)$ *of the observation e occurring, conditioned on that candidate* d_k (maximum likelihood estimation for naive Bayes classifier d_k). Hence, h_j is solved

Algorithm 2 Diagnostic Algorithm: Barinel

```
Input: Activity matrix $A$, error vector $e$
Output: Diagnostic Report $D$
1  $\gamma \leftarrow \epsilon$
2  $D \leftarrow$ Staccato$((A, e))$                ▷ Compute MHS
3  for all $d_k \in D$ do
4      expr $\leftarrow$ GeneratePr$((A, e), d_k)$
5      $i \leftarrow 0$
6      $\Pr[d_k]^i \leftarrow 0$
7      $\forall_{j \in d_k} g_j \leftarrow 0.5$
8      repeat
9          $i \leftarrow i + 1$
10         for all $j \in d_k$ do
11             $g_j \leftarrow g_j + \gamma \cdot \nabla \mathrm{expr}(g_j)$
12         end for
13         $\Pr[d_k]^i \leftarrow$ evaluate$(\mathrm{expr}, \forall_{j \in d_k} g_j)$
14     until $|\Pr[d_k]^{i-1} - \Pr[d_k]^i| \leq \xi$     ▷ where $\xi > 0$ is the error tolerance
15 end for
16 return sort$(D, \Pr)$
```

by maximizing $\Pr(e|d_k)$ under the above epsilon policy, according to

$$H = \arg\max_{H} \Pr(e|d_k)$$

where $H = \{h_j \mid j \in d_k\}$.

5.2.2 *Algorithm*

In this section we present our approach to compute the h_j and the associated, posterior candidate probabilities $\Pr(d_k|obs)$ given a set of observations (A, e), along with some examples. Our approach is described in Algorithm 2 and comprises three main phases. In the first phase (line 2) a list of candidates D is computed from (A, e) using Staccato (typically, 100 candidates, yet capturing all significant probability mass). In the second phase $\Pr(d_k)$ is computed for each candidate in D (lines 3 to 15). First, GeneratePr derives for every candidate d_k a symbolic expression for the probability $\Pr(e|d_k)$ for the current set of observations (A, e). For example, suppose the following measurements where c_1, c_2 are faulty (ignoring other components):

c_1	c_2	e	$\Pr(e_i\|\{1,2\})$
1	0	1	$1 - h_1$
1	1	1	$1 - h_1 \cdot h_2$
0	1	0	h_2
1	0	0	h_1

$\Pr(e_i|d_k)$ is computed using the same probability computation as in Eq. (5.3). As the four observations are independent, the probability of observing e assuming $d_k = \{1,2\}$ equals

$$\Pr(e|d_k) = h_1 \cdot h_2 \cdot (1 - h_1) \cdot (1 - h_1 \cdot h_2)$$

In the next phase, all h_j are computed such that they maximize $\Pr(e|d_k)$. To solve the maximization problem we apply a simple gradient ascent procedure [Avriel, 2003] bounded within the domain $0 < h_j < 1$ (the ∇ operator signifies the gradient computation).

In the third and final phase, for each d_k the diagnoses are ranked according to $\Pr(d_k|(A,e))$, which is computed by Evaluate based on the usual Bayesian update (Eq. (5.2) for each row):

$$\Pr(d_k|(A,e)) = \frac{\Pr(e|d_k)}{\Pr(obs)} \cdot \Pr(d_k)$$

where $\Pr(e|d_k)$ is the probability that e is observed assuming d_k correct.

5.2.3 *Maximum Likelihood Estimation*

For single-fault conditions the maximization procedure is trivial. Consider candidate c_1 (i.e, $d_k = \{1\}$). Intuitively, the maximum likelihood estimator for h_1 equals the average health state $h_1 = n_{10}(1)/(n_{10}(1) + n_{11}(1))$ (fraction of passes for the runs c_1 is involved in). Consider the following (A,e) (only showing the column of c_1 and the rows where c_1 is hit), e, and the probability of that occurring (Pr):

c_1	e	$\Pr(e_i\|\{1\})$
1	0	h_1
1	0	h_1
1	1	$1 - h_1$
1	0	h_1

Averaging e (3 passes and 1 fail) yields the estimate $h_1 = \frac{3}{4}$. To prove that this is a perfect estimate, we show that h_1 maximizes the probability of this particular e (or any permutation with 1 fail and 3 passes) to occur. As $\Pr(e|\{1\})$ is given by $\Pr(e|\{1\}) = h_1^3 \cdot (1 - h_1)$, the value of h_1 that maximizes $\Pr(e|\{1\})$ is indeed $\frac{3}{4}$.

Proof Let $x = n_{10}(1)/(n_{10}(1) + n_{11}(1))$. denote our intuitive estimation of h_1. Let $N' = n_{10}(1) + n_{11}(1)$ denote the number of runs in which c_1 is involved. Thus $n_{10} = N' \cdot h_1$ and $n_{11} = N' \cdot (1 - h_1)$, respectively. Consequently, $\Pr(e|d_k)$ is given by

$$\Pr(e|d_k) = x^{N' \cdot h_1} \cdot (1 - x)^{N' \cdot (1 - h_1)}$$

Maximizing $\Pr(e|\{1\})$ implies maximizing $x^{h_1} \cdot (1 - x)^{1 - h_1}$ as N' is independent of x. The value x that maximizes this expression is the one for which its derivative to x equals zero. Consequently,

$$x \cdot h_1^{x-1} \cdot (1-h_1)^{1-x} - x^{h_1} \cdot (1-x) \cdot (1-h_1)^{(1-x-1)} = 0$$

which reduces to

$$x \cdot (1-h_1) = h_1 \cdot (1-x)$$

yielding $x = h_1$. □

For multiple-fault candidates the derivation of h_j is not so trivial. Instead of generalizing to the C-fault case, we just treat the $C = 2$ case for ease of exposition, as generalization to C faults is straightforward. Consider the same (A, e) given in Section 5.2.2 for c_1 and c_2 at fault. As there is a failing row that involves both c_1 and c_2 we cannot just estimate h_j through the above, single-fault approach (i.e., averaging over e) due to the mutual influence of both h_1 and h_2 on e_2, reflected in the $(1 - h_1 \cdot h_2)$ term in the $Pr(e|d_k)$ equation

$$\Pr(e|d_k) = h_1 \cdot h_2 \cdot (1-h_1) \cdot (1-h_1 \cdot h_2)$$

In general, for a double fault A will contain

a	10 0	entries (with Pr $= h_1$)
b	10 1	entries (with Pr $= 1 - h_1$)
c	01 0	entries (with Pr $= h_2$)
d	01 1	entries (with Pr $= 1 - h_2$)
e	11 0	entries (with Pr $= h_1 \cdot h_2$)
f	11 1	entries (with Pr $= 1 - h_1 \cdot h_2$)

where a, …, f are samples from the binomial distributions (with $\mu = N_F \cdot h_1$, $\sigma = N_F \cdot h_1 \cdot (1 - h_1)$, …, $\mu = N_F \cdot h_1 \cdot h_2$, $\sigma = N_F \cdot h_1 \cdot h_2 \cdot (1 - h_1 \cdot h_2)$, respectively). Consequently, we need to find the h_j that maximize

$$\Pr(e) = h_1^a \cdot (1-h_1)^b \cdot h_2^c \cdot (1-h_2)^d \cdot h_1^e \cdot h_2^e \cdot (1-h_1 \cdot h_2)^f$$

The above easily generalizes to the C-fault case although the formulae become much more complex. Due to the shape of the expression, an analytic solution to the maximization problem is not as straightforward as in the single-fault case (except when $f = 0$ in which case A is partitionable) which has prompted us to apply our numeric maximization approach (gradient ascent) as in real problems A is never partitionable.

As the formulae that need to be maximized are simple and bounded in the $[0, 1]$ domain, the time/space complexity of our approach is identical to previous spectrum-based reasoning policies, e.g., [de Kleer, 2007, Abreu et al., 2008c] (see next section for a detailed discussion) modulo a constant factor on account of the gradient ascent procedure, which exhibits reasonably rapid convergence for all M and C (see Section 5.4.3). Note that the linear convergence of the simple, gradient ascent procedure can be improved to a quadratic convergence (e.g., Newton's method), yielding significant speedup.

5.2.4 *Estimating Intermittency*

As said before, there are many ε policies that can be used in the Bayesian update formula (see Eq. (5.3) [de Kleer, 2007]). As calibration data on correct and incorrect component behavior is typically not available, previous efforts to diagnose intermittent component failures, such as [de Kleer, 2007, De Kleer et al., 2008, Kuhn et al., 2008, Abreu et al., 2008c, de Kleer, 2009], have instead *approximated* h_j for the faulty components in d_k, $g(d_k)$. Yet another reason for using approximation is that obtaining h_j from the activity matrix was far from trivial. This "effective" intermittency parameter $g(d_k)$ is estimated for the candidate d_k by counting how many times components in d_k are involved in passed and failed runs. The parameter $g(d_k)$ is defined as follows

$$g(d_k) = \frac{\sum\limits_{i=1..N} [(\bigvee\limits_{j \in d_k} a_{ij} = 1) \wedge e_i = 0]}{\sum\limits_{i=1..N} [\bigvee\limits_{j \in d_k} a_{ij} = 1]}$$

where $[\cdot]$ is Iverson's operator [Iverson, 1962] ([true] $= 1$, [false] $= 0$).

In the following we distinguish between two, intermittent policies, which we refer to as $\epsilon^{(1)}$, and $\epsilon^{(2)}$, which are defined as follows

$$\epsilon^{(1)} = \begin{cases} g(d_k) & \text{if run passed} \\ 1 - g(d_k) & \text{if run failed} \end{cases}$$

and

$$\epsilon^{(2)} = \begin{cases} g(d_k)^{\eta} & \text{if run passed} \\ 1 - g(d_k)^{\eta} & \text{if run failed} \end{cases}$$

where η is the number of faulty components according to d_k involved in the run i

$$\eta = \sum_{j \in d_k} [a_{ij} = 1]$$

We propose policy $\varepsilon^{(2)}$ as a variant of $\varepsilon^{(1)}$, proposed in [de Kleer, 2007]. It approximates the probability $\prod_{j \in d_k} h_j$ that the components in d_k all exhibit good behavior by $g(d_k)^{\eta}$, assuming that all components of d_k have equal goodness probabilities.

As a final example to illustrate the benefits of our approach, consider the following program spectra (c_1 and c_2 faulty):

c_1	c_2	c_3	c_4	c_5	e
1	1	0	1	0	1
0	1	0	0	1	1
1	0	0	0	1	1
1	1	0	0	1	1
0	1	0	0	1	1
1	0	0	1	1	0

d_k	$\varepsilon^{(0)}$	$\varepsilon^{(1)}$	$\varepsilon^{(2)}$	Barinel
$\{1,2\}$	0.25	0.25	0.40	0.91
$\{2,5\}$	0.25	0.25	0.47	0.07
$\{1,5\}$	0.25	0.25	0.07	0.018
$\{4,5\}$	0.25	0.25	0.06	0.002

Table 5.1 Diagnosis candidates' probabilities

Staccato yields the diagnostic candidates $\{1,2\},\{2,5\},\{1,5\},\{4,5\}$. As said before, many ε policies exist. Besides the two intermittent policies outlined above, in the following we also consider a *traditional* policy, $\epsilon^{(0)}$, which is defined as follows

$$\varepsilon^{(0)} = \begin{cases} \frac{E_P}{E_P+E_F} & \text{if run passed} \\ \frac{E_F}{E_P+E_F} & \text{if run failed} \end{cases} \tag{5.5}$$

where $E_P = 2^M$ and $E_F = (2^l - 1) \cdot 2^{M-l}$ are the number of passed and failed observations that can be explained by diagnosis d_k, respectively, and $l = |d_k|$ is the number of faulty components in the diagnosis. Note that this policy is slightly different from the one in Section 5.1, as the lack of component interconnection information allows more diagnoses (component combinations) as likely explanations for pass/fail outcomes.

The probabilities for the valid diagnosis candidates are presented in Table 5.1. Common to traditional policies, $\epsilon^{(0)}$ does not distinguish between candidates with the same cardinality. Hence, as they rank with the same probability, half of all candidates would have to be inspected on average. Although $\epsilon^{(1)}$ does distinguish between candidates with equal cardinality, in this example the $g(d_k)$ is estimated to be the same, therefore yielding the ranking as $\epsilon^{(0)}$. In addition to $g(d_k)$, $\epsilon^{(2)}$ uses the number of passed and failed runs the component is involved in to further distinguish between diagnosis candidates with equal cardinality, and it ranks $\{2,5\}$ at the first place. Still the developer has to inspect a component in vain. Barinel yields better results due to a correct estimation of the individual h_j, ranking the true fault $\{1,2\}$ at the first position.

5.3 ANALYTIC MODEL

In this section we derive a simple, approximate model to assess the influence of various parameters on the diagnostic performance of our framework. Diagnostic performance is measured in terms of a diagnostic performance metric W that measures the percentage of excess *work* incurred in finding the actual components at fault. The metric is an improvement on metrics typically found in software debugging which measure debugging effort [Abreu et al., 2007, Renieris and Reiss, 2003]. We use wasted effort instead of effort because in our multiple-fault research context we wish the metric to be independent of the number of faults C in the program to enable easier evaluation of the effect of C on W. For instance, consider a $M = 5$ component program with the following diagnostic report $D =< \{4,5\},\{4,3\},\{1,2\} >$, while c_1 and c_2 are actually faulty. The first diagnosis candidate leads the developer to inspect c_4 and c_5. As both components are healthy, W is increased with $\frac{2}{5}$ and $h_4 = h_5 = 1.0$. Using the knowledge that components 4 and 5 are 100% healthy, the probabilities of the remaining candidates are updated, leading to $\Pr(\{4,3\}) = 0$. Consequently, candidate $\{4,3\}$ is also discarded. The next components to be inspected are c_1 and c_2. As they are both faulty, no more wasted effort is incurred[2].

The evaluated parameters are number of components M, number of test cases N, testing code coverage r, testing fault coverage h, and fault cardinality C. Consider the example A in Figure 5.2(a), with $M = 5$ components of which the first $C = 2$ components are faulty. As a faulty component can still produce correct behavior which does not not cause a run to fail, we use an extended encoding where '1' denotes a component that is involved, and '2' denotes a (faulty) component whose involvement actually produced a failure (and consequently a failing run). Note, however, that a debugger only knows about component involvement and does not know whether a component is responsible for an observed failure or not.

c_1	c_2	c_3	c_4	c_5	e
1	0	1	0	1	0
0	2	1	0	0	1
0	2	1	1	0	1
1	1	1	1	0	0
2	1	0	1	0	1

(a) Example A

c_1	c_2	c_3	c_4	c_5	e
0	2	1	0	0	1
0	2	1	1	0	1
2	1	0	1	0	1

(b) A's failed runs only

Figure 5.2 Activity Matrix Example

In the following we focus on the hitting set since its constituents are primarily responsible for the asymptotic behavior of W. Although their individual ranking is influenced by component activity in passed runs, the hitting set itself is exclusively determined by the failing runs. Thus, we consider the sub-matrix shown in Figure 5.2(b).

From Figure 5.2(b) it can be seen that the first 2 columns together form a hitting set of cardinality 2 (which corresponds to our choice $C = 2$). This can

[2]Effort, as defined in [Abreu et al., 2007, Renieris and Reiss, 2003], would be increased by $\frac{2}{5}$ to account for the fact that both components were inspected.

be seen by the fact that in each row there is at least one set member involved, i.e., there is a so-called "chain" of c_1 and/or c_2 involvement that is "unbroken" from top row to bottom row.

While this chain exists by definition (given the fact that both are faulty there is always at least one of them involved in *every* failed run), other chains may also exist, and may cause W to increase. This occurs when those chains pertain to diagnostic candidates of equal or lower cardinality (B) than C. Generally, two types of chains can be distinguished: (1) chains (of cardinality $B < C$) within the faulty components set, called *internal* chains, and (2) chains (of cardinality $B \leq C$) completely outside the faulty components set, called *external* chains. In the above example after $N = 2$ (so considering only the first two failed runs, see Figure 5.2(b)), there is still one internal chain (corresponding to single fault c_2), and two external chains (corresponding to single fault $\{3\}$, and double fault $\{3, 4\}$). As their probability will be higher (due to the a priori probability computation) they will head the ranking. With respect to the internal fault this does not significantly influence W since this indicates a true faulty component (the real double fault $\{1, 2\}$ being subsumed by $\{2\}$). Consequently, there is no wasted debugging effort. With respect to $\{3\}$ however, this fault will induce wasted effort. After $N = 3$ both single faults has disappeared (both chain of '1's have been *broken* during the third failing run), while the double fault c_3, c_4 is still present. From the above example it follows that (1) W is primarily impacted by external chains, and (2) the probability of a B cardinality chain still "surviving" decreases with the number of failing runs. Assuming that components are executed randomly, the latter is the reason why in the limit for $N \rightarrow \infty$ all external (and internal) chains will have disappeared, exposing the true fault as only diagnosis.

5.3.1 *Number of Failing Runs*

As the number of failing runs is key to the behavior of W, in the following we first compute the fraction of failed runs f out of the total of N runs, given r and h. Consider C faulty components. Let f denote the probability of a run failing. A run passes when none of the C components induces a failure, i.e., does not generate a '2' in the matrix. Since the probability of the latter equals $1 - r \cdot (1 - h)$ and generating a '2' requires (1) being involved (probability r) and (2) producing a failure (probability $(1 - h)$), the probability of not generating a '2' in the matrix equals $(1 - r \cdot (1 - h))$. Consequently, the probability a run passes equals $(1 - r \cdot (1 - h))^C$, yielding

$$f = 1 - (1 - r \cdot (1 - h))^C$$

This implies that for high h (and/or low r) a very large number of runs N is required to generate a sufficient number $N_F = f \cdot N$ of failing runs in order to eliminate competing chains of equal of lower cardinality B. As r also affects the number of external chains which, however, is not affected by h, the effect of h can be seen orthogonal to r in that it only impacts the number of failed runs through f. Consequently, h and N are related in that a high h is compensated by a, possible huge, increase in N. In the sequel, we therefore only focus on the effect of r.

5.3.2 *Behavior for Small Number of Runs*

While for large N the determination of W depends on the probability that competing chains will have terminated, for small N a more simple derivation can be made. Consider the case of a single failing run ($N_F = f \cdot N = 1$). From the first (failing) row ($k = 1$) in the above example (Figure 5.2(b)) it can be seen that there are generally $r \cdot (M - C)$ external single-fault ($B = 1$) chains (c_3 and c_5) that induce wasted effort. As W denotes the ratio of wasted effort it follows

$$W = \frac{r \cdot (M - C)}{M} \tag{5.6}$$

which for large M approaches r. This is confirmed by the experiments discussed later.

After the second failed run ($k = 2$) the probability a $B = 1$ chain survives two failing runs equals r^2 (i.e., the probability of two '1's for a particular component). Consequently, the number of $B = 1$ chains equals $r^2 \cdot (M - C)$, which, in general, decreases negative-exponentially with the number of (failing) runs ($f \cdot N$). For $B = 2$ the situation is less restrictive as *any* combination of '1's of the first and second row qualifies as a double-fault chain. As on average there are $M' = \lfloor r \cdot (M - C) \rfloor$ '1's per row there are $\binom{M'}{2}$ double-faults.

After the third failing run ($k = 3$) the number of surviving $B = 1$ chains equals $r^3 \cdot (M - C)$, whereas the number of triple faults equals $\binom{M'}{3}$. As for sufficiently large M the higher-cardinality combinations outnumber the lower-cardinality combinations, W is dominated by the combinations that have the same cardinality as the fault cardinality C. Consequently, assuming $N_F \leq C$ it follows that the number of C-cardinality chains that compete with the actual C-cardinality diagnosis is approximated by $\binom{M'}{C}$. However, if there are more combinations than $M - C$ these combinations will overlap in terms of component indices. As W does not measure wasted effort on a component that was already previously inspected (and subsequently removed from the next diagnosis), the average number of "effective" C-cardinality chains will never exceed $\frac{M}{C}$ (as there are C indices per candidate). Hence, the number of competing C-cardinality chains is approximated by $\min(\frac{M}{C}, \binom{M'}{C})$.

5.3.3 *Behavior for Large Number of Runs*

For large N_F the trend of W can also be approximated from the probability that competing chains will still have survived after N_F runs, which we derive as follows. Consider a B-cardinality external chain. At each row there is a probability that this chain does not survive. Similar to the derivation of f we consider the probability that *all* B components involved in the chain have a '0' entry, which would terminate that particular chain. This probability equals $(1-r)^B$. Hence, the probability that a B-cardinality chain does not break per run equals $1-(1-r)^B$. Consequently, the probability that a chain survives N_F failing runs equals

$$(1-(1-r)^B)^{N_F}$$

Similar to the derivation for small N_F, we only consider C-cardinality chains. The largest number of competing chains at the outset equals $\binom{M'}{C}$. As there always exists an N_F for which this number is less than $\frac{M}{C}$ (in the asymptotic case we consider only a few chains) the number of competing chains after N_F runs is given by

$$(1-(1-r)^C)^{N_F} \cdot \binom{M'}{C}$$

Consequently, W is approximated by

$$W \approx \frac{(1-(1-r)^C)^{N_F} \cdot \binom{M'}{C}}{M} \tag{5.7}$$

We observe a negative-exponential (geometric) trend with N_F (N) while C postpones that decay to larger N_F (N) as the term $1-(1-r)^C$ approaches unity for large C.

In the following we asymptotically approximate the number of failing test runs N_F needed for an optimal diagnosis (i.e., W approaches 0). Considering Eq. (5.7), a constant number of diagnoses (ideally, a single diagnosis) is approximately reached for

$$(1-(1-r)^C)^{N_F} \cdot \binom{M'}{C} = W \cdot M = \text{very small} = K$$

It follows $N_F = -\log K/\log 1-(1-r)^C$. Since for sufficiently large C the term $1-(1-r)^C$ approaches unity, and since $\log 1-\epsilon \approx -\epsilon$ it follows that $N_F \sim \log K/(1-r)^C$. As $(1-r) < 1$ it follows $N_F \sim \log K \cdot ((1-r)^{-1})^C$ of which the second term increases exponentially with C. Since $K = \binom{M'}{C}$ for large M this term also increases exponentially with C. However, as the term is included in a logarithm, the effect of this term is less than the previous. In the next section we numerically verify the exponential trend of N.

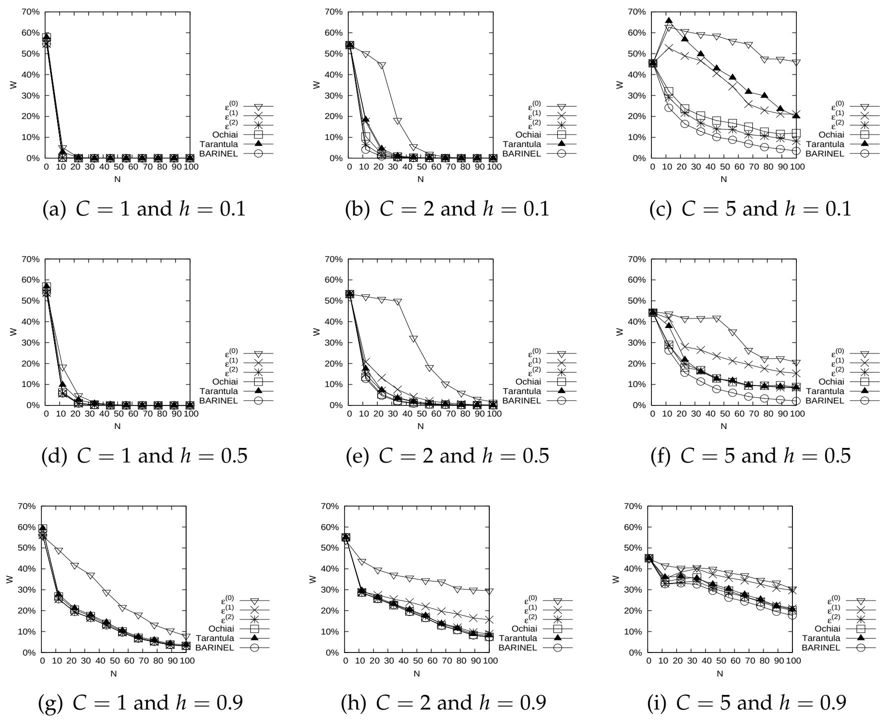
5.3.4 *Experimental Validation*

In order to experimentally validate the predictions of the model just outlined and assess the performance improvement of our framework, we generate synthetic observations based on random (A, e) generated for various values of N, M, and the number of injected faults C (cardinality). Component activity a_{ij} is sampled from a Bernoulli distribution with parameter r, i.e., the probability a component is involved in a row of A equals r. For the C faulty components c_j (without loss of generality we select the first C components, i.e., $c_1, \ldots, c_C$ are faulty). We also set the component healths (intermittency rates) h_j. Thus the probability of a component j being involved *and* generating a failure equals $r \cdot (1 - h_j)$. A row i in A generates an error ($e_i = 1$) if at least 1 of the C components generates a failure (or-model). Measurements for a specific (N, M, C, r, g) scenario are averaged over $1,000$ sample matrices, yielding a coefficient of variance of approximately 0.02.

We compare the accuracy of our Bayesian framework with the three previous Bayesian approaches $\epsilon^{(0)}$, $\epsilon^{(1)}$, and $\epsilon^{(2)}$, and the two spectrum-based fault localization methods Ochiai and Tarantula. The graphs in Figure 5.3 plot W versus N for $M = 20$, $r = 0.6$ (the trends for other M and r values are essentially the same, $r = 0.6$ is typical for the Siemens suite), and different values for C and h (in our experiments we set all $h_j = h$). A number of common properties emerge. All plots show that W for $N = 1$ is similar to r, which agrees with the fact that there are on average $(M - C) \cdot r$ components which would have to be inspected in vain. For sufficiently large N all approaches produce an optimal diagnosis, as there are sufficient runs for all approaches to correctly single out the faulty components. For small h_j, W converges quicker than for large h_j as computations involving the faulty components are much more prone to failure, while for large h_j the faulty components behave almost similar to healthy components, requiring more observations (larger N) to rank them higher. Also for increasing C more observations are required before the faulty components are isolated. This is due to the fact that failure behavior can be caused by much more components, reducing the correlation between failure and particular component involvement.

The plots confirm that Barinel is the best performing approach. $\epsilon^{(0)}$ is the worst performing approach, mainly because (1) it does not not distinguish between diagnosis with the same fault cardinality C, and (2) it does not exonerate components based on their involvement on passed runs. Only for $C = 1$ the $\epsilon^{(1)}/\epsilon^{(2)}$ approach has equal performance to Barinel, as for this trivial case the approximations for the h_j are equal. For $C \geq 2$ the plots confirm that Barinel has superior performance, demonstrating that an exact estimation of h_j is quite relevant. The more challenging the diagnostic problem becomes (higher fault densities), the more Barinel stands out compared to the SFL approaches and the previous Bayesian approaches.

The plots show that W for $N = 1$ is similar to r as predicted in Eq. (5.6), while for sufficiently large N all techniques produce an optimal diagnosis. Besides, from the plots we verify that the higher C the more runs N are needed to attain optimal diagnostic performance. As an example, for $h = 0.1$, $r = 0.4$, and $C = 1$, 10 runs would be enough for a perfect diagnosis, whereas for $C = 5$, 250 runs would be needed. For small h almost each run that involves the faulty component yields a failure ($f \approx 1$), already producing near-perfect diagnoses for only small N. For high h the transition between the small-N_F behavior and large-N_F behavior is visible. As the negative-exponential trend with N_F is clear from the analytical model we have determined the value of N (N_F) for which our C-cardinality fault remains as the only candidate, i.e., a



(a) $C = 1$ and $h = 0.1$ (b) $C = 2$ and $h = 0.1$ (c) $C = 5$ and $h = 0.1$

(d) $C = 1$ and $h = 0.5$ (e) $C = 2$ and $h = 0.5$ (f) $C = 5$ and $h = 0.5$

(g) $C = 1$ and $h = 0.9$ (h) $C = 2$ and $h = 0.9$ (i) $C = 5$ and $h = 0.9$

Figure 5.3 Wasted effort W vs. N for several settings of C and h

perfect multiple-fault diagnosis. Table 5.2 shows the values of N (N_F) where optimality is reached for different values of C and h. Apart from a scaling due to h one can clearly see the exponential impact of C on N_F and N.

h	0.1					0.9				
C	1	2	3	4	5	1	2	3	4	5
N^*	13	31	90	120	250	200	300	500	1000	1700
N_F	5	19	71	111	245	12	36	84	219	459

Table 5.2 Optimal N^* for perfect diagnosis ($r = 0.6$)

5.4 EMPIRICAL EVALUATION

Whereas the synthetic matrices used in the previous section are populated using a uniform distribution, this is not the case with matrices for actual software programs. In this section, we evaluate the diagnostic capabilities and efficiency of the diagnosis techniques for real-world programs.

Program	Faulty Versions	M	N	Description
`print_tokens`	7	539	4,130	Lexical Analyzer
`print_tokens2`	10	489	4,115	Lexical Analyzer
`replace`	32	507	5,542	Pattern Recognition
`schedule`	9	397	2,650	Priority Scheduler
`schedule2`	10	299	2,710	Priority Scheduler
`tcas`	41	174	1,608	Altitude Separation
`tot_info`	23	398	1,052	Information Measure
`space`	38	9.564	13,585	Array Definition Language

Table 5.3 The Siemens benchmark set

5.4.1 *Experimental Setup*

For evaluating the performance of our approach we use the well-known Siemens benchmark set and `space` [Do et al., 2005]. For ease of reference, Table 5.3 summarizes the Siemens benchmark set and `space`, where M corresponds to the number of lines of code (components in this context). The test suite package of `space` also provides $1,000$ test suites that consist of a random selection of (on average) 150 test cases and guarantees that each branch of the program is exercised by at least 30 test cases. In our experiments, the test suite used is randomly chosen from the $1,000$ suites provided.

For our experiments, we have extended the Siemens benchmark set and `space` with program versions where we can activate arbitrary combinations of *multiple* faults. For this purpose, we limit ourselves to a selection of 130 out of the 170 faults, based on criteria such as faults being attributable to a single line of code, to enable unambiguous evaluation.

The activity matrices are obtained using the GNU `gcov`[3] profiling tool and a script to translate its output into a matrix. As each program suite includes a correct version, we use the output of the correct version as reference. We characterize a run as failed if its output differs from the corresponding output of the correct version, and as passed otherwise.

5.4.2 *Performance Results*

In this section we evaluate the diagnostic capabilities of Barinel and compare it with several fault localization techniques. We first evaluate the performance in the context of single faults, and then for multiple fault programs.

Single Faults

We compare Barinel with several well-known statistics-based techniques which have used the Siemens benchmark set described in the previous section. Although the set comprises 132 faulty programs, two of these programs, namely version 9 of `schedule2` and version 32 of `replace`, are discarded as no

[3]http://gcc.gnu.org/onlinedocs/gcc/Gcov.html

failures are observed. Besides, we also discard versions 4 and 6 of `print_tokens` because the faults are not in the program itself but in a header file. In summary, we discarded 4 versions out of 132 provided by the suite, using 128 versions in our experiments. For compatibility with previous work in (single-) fault localization, we use the effort/*s*core metric [Abreu et al., 2007, Renieris and Reiss, 2003] which is the percentage of statements that need to be inspected to find the fault - in other words, the rank position of the faulty statement divided by the total number of statements. Note that some techniques such as in [Liu et al., 2006, Renieris and Reiss, 2003] do not rank all statements in the code, and their rankings are therefore based on the program dependence graph of the program.

Figure 5.4 plots the percentage of located faults in terms of debugging effort. Apart from the two SFL approaches, Ochiai and Tarantula, the following techniques are also plotted: Intersection and Union [Renieris and Reiss, 2003], Delta Debugging (DD) [Zeller, 2002], Nearest Neighbor (NN) [Renieris and Reiss, 2003], Sober [Liu et al., 2006], PPDG [Baah et al., 2008], and CrossTab [Wong et al., 2008], which are amongst the best statistics-based techniques (see Section 5.5). In the single fault context, as explained in the previous section, $\epsilon^{(1)}$ and $\epsilon^{(2)}$ perform equally well as Barinel, and all outperform $\epsilon^{(0)}$. As Sober is publicly available, we run it in our own environment. The values for the other techniques are, however, directly taken from their respective papers. From Figure 5.4, we conclude that Barinel is consistently

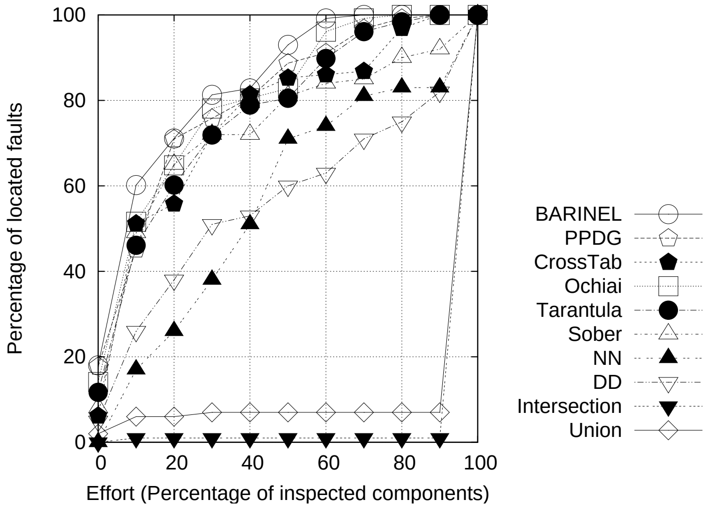


Figure 5.4 Effectiveness Comparison ($C = 1$)

the best performing technique, finding 60% of the faults by examining less than 10% of the source code. For the same effort, using Ochiai would lead a developer to find 52% of the faulty versions, and with Tarantula only 46%

`tcas`	AIM	`explain`	Δ-slicing	$\epsilon^{(0)}$	$\epsilon^{(1)}/\epsilon^{(2)}$/BARINEL
v1	0.74	0.51	0.91	0.13	0.99
v11	0.84	0.36	0.93	0.17	0.97
v31	0.77	0.76	0.93	0.17	0.98
v40	0.85	0.75	–	0.17	0.90
v41	0.73	0.68	0.88	0.18	0.99

Table 5.4 Comparison with model-based approaches

would be found. For an effort of less than 1% PPDG performs equally well as BARINEL. Our approach outperforms Ochiai, which is consistently better than Sober and Tarantula. The former two yield similar performance, as also concluded in [Liu et al., 2006]. Finally, the other techniques plotted are clearly outperformed by the spectrum-based techniques.

Table 5.4 compares the different policies used in our approach with AIM, `explain`, and Δ-slicing for 5 versions of `tcas`, because these are the versions to which `explain` and Δ-slicing could be applied to. From the table, we conclude that BARINEL and $\epsilon^{(1)}/\epsilon^{(2)}$ consistently outperforms all other techniques, with $\epsilon^{(0)}$ being the worst performing technique. The fact that the diagnostic quality of BARINEL and $\epsilon^{(1)}/\epsilon^{(2)}$ is due to the fact that the estimation used in $\epsilon^{(1)}/\epsilon^{(2)}$ is perfect.

The reason for BARINEL's superiority for single-fault programs is established in terms of the following theorem.

Theorem *For single-fault programs, given the available set of observations (A, e), the diagnostic ranking produced by* BARINEL *is theoretically optimal.*

Proof In the single-fault case, the maximum likelihood estimation for h_j reduces from a numerical procedure to a simple analytic expression given by

$$h_j = \frac{n_{10}(j)}{n_{10}(j) + n_{11}(j)} = \frac{x(j)}{x(j) + 1}$$

as, by definition, h_j is the pass fraction of the runs where c_j is involved. Consequently, $\Pr(d_k|e)$ can be written as

$$\Pr(d_k|e) = h_j^{x(j) \cdot n_{11}(j)} \cdot (1 - h_j)^{n_{11}(j)}$$

where $x(j) = n_{10}(j)/n_{11}(j)$. For $C = 1$ where a run fails the faulty component *has* to be involved. In BARINEL, when a candidate does not explain all failing runs its probability is set to 0 as a result from the consistency-based reasoning within BARINEL (cf. the 0-clause of Eq. (5.3)). This implies that for the remaining candidates $n_{11}(j)$ equals the number of failing runs, which is independent of j. Hence, with respect to the *ranking* the constant $n_{11}(j)$ can be ignored, yielding

$$\Pr'(d_k|e) = \frac{x(j)^{x(j)}}{(x(j) + 1)^{(x(j)+1)}}$$

Since $x(j) > 0$, $\Pr'(d_k|e)$, and therefore $\Pr(d_k|e)$, is monotonically decreasing with $x(j)$ and therefore with h_j. Consequently, the ranking in D equals the (inverse) ranking of h_j. As the maximum likelihood estimator for h_j is perfect by definition, the ranking returned by Barinel is optimal. □

While the above theorem establishes Barinel's optimality, the following corollary describes the consequences with respect to similarity coefficients.

Corollary *For single-fault programs, any similarity coefficient that includes $n_{10}(j)$ in the denominator is optimal, provided components c_j are removed from the ranking for which $n_{01}(j) \neq 0$.*

Proof From the above theorem it follows that the ranking in terms of h_j is optimal for the subset of components indicted by the reasoning process, i.e., those components that are always involved in a failing run. The latter condition implies that only components for which $n_{01}(j) = 0$ can be considered. The former implies that (for this subset) the similarity coefficient

$$s(j) = 1 - h_j = \frac{n_{11}(j)}{n_{11}(j) + n_{10}(j)}$$

is optimal. As for the components subset $n_{11}(j)$ is constant, $n_{10}(j)$ determines the ranking while $n_{01}(j)$ plays no role. As all similarity coefficients have an $n_{11}(j)$ term in the numerator, it follows that as long as $n_{10}(j)$ is present in the denominator (the only term that varies with j), such a coefficient yields the same, optimal, ranking as the above Barinel expression for $s(j)$. □

Experiments using the $n_{01}(j) = 0$ "reasoning" filter, combined with a simple similarity coefficient such as Tarantula or Ochiai indeed confirm that this approach leads to the best performance [Vayani, 2007] (equal to Barinel).

Multiple Faults

We now proceed to evaluate our approach in the context of multiple faults, using our extended Siemens benchmark set and `space`. In contrast to Section 5.4.2 we only compare with the same techniques as in Section 5.3.4 ($\epsilon^{(0)}$, $\epsilon^{(1)}$, $\epsilon^{(2)}$, Tarantula, and Ochiai) as for the other related work no data for multiple-fault programs are available. Similar to Section 5.3.4, we aimed at $C = 5$ for the multiple fault-cases, but for `print_tokens` insufficient faults are available. All measurements except for the four-fault version of `print_tokens` are averages over 100 versions, or over the maximum number of combinations available, where we verified that all faults are active in at least one failed run.

Table 5.5 presents a summary of the diagnostic quality of the different techniques. The diagnostic quality is quantified in terms of wasted debugging effort W (see Section 5.3.4 for an explanation of the difference between wasted effort and effort). Again, the results confirm that on average Barinel outperforms the other approaches, especially considering the fact that the variance of W is considerably higher (coefficient of variance up to 0.5 for `schedule2`) than in the synthetic case (1,000 sample matrices versus up to 100 matrices in the Siemens case). Only in 4 out of 24 cases, Barinel is not on top. Apart from the obvious sampling noise (variance), this is due to particular properties of the programs. Using the paired two-tailed Student's t-test, we verified that the differences in the means of W are not significant for those cases in which Barinel does not clearly outperforms the other approaches, and thus noise is the cause for the small differences in terms of W. As an example, for `print_tokens2` with $C = 2$ the differences in the means are significant, but it is not the case for `schedule` with $C = 1$. For `tcas` with $C = 2$ and $C = 5$, $\epsilon^{(1)}$ marginally outperforms Barinel (by less than 0.5%), Ochiai being the best performing approach. This is caused by the fact that (1) the program is almost branch-free and small ($M = 174$) combined with large sampling noise (σ_W = 5% for `tcas`), and (2) almost all failing runs involve all faulty components (highly correlated occurrence). Hence, the program effectively has a single fault spreading over multiple lines. For `schedule2` with $C = 2$ and $C = 5$, $\epsilon^{(0)}$ is better due to the fact that almost all failing runs involve all faulty components (highly correlated occurrence). Hence, the program effectively has a single fault spreading over multiple lines, which favors $\epsilon^{(0)}$ since it ranks candidates with cardinality one first. For `tcas` with $C = 2$ and $C = 5$, $\epsilon^{(2)}$ marginally outperforms Barinel (by less than 0.5%), being Ochiai the best performing approach which is caused by the fact that the program is almost branch-free and small ($M = 174$) combined with large sampling noise (significant variance of the individual diagnostic performances, σ_W = 5% for `tcas`).

Our results show that W decreases with increasing program size (M). This confirms our expectation that the effectiveness of automated diagnosis techniques generally improves with program size. As an illustration, near-zero wasted effort is measured in experiments with large NXP codes as described in Chapter 3, where the problem reports (tests) typically focus on a particular anomaly (small C).

5.4.3 *Time/Space Complexity*

In this section we report on the time/space complexity of Barinel, compared to other fault localization techniques.

In contrast to the M components in statistical approaches, the Bayesian techniques update $|D|$ candidate probabilities where $|D|$ is determined by Staccato. Although in all our measurements a constant $|D| = 100$ suffices, it is not unrealistic to assume that for very large systems $|D|$ would scale with M, again, yielding $O(N \cdot M)$ for the probability updates. However, there are two differences with the statistical techniques, (1) the cost of Staccato and (2) in case of Barinel, the cost of the maximization procedure. The complexity of Staccato is estimated to be $O(N \cdot M)$ (for a constant matrix density r) [Abreu et al., 2008d]. The complexity of the maximization procedure appears to be rather independent of the size of the expression (i.e., M and C) reducing this term to a constant. As, again, the report is ordered, the time complexity again

		print_tokens			print_tokens2			replace			schedule		
	C	1	2	4	1	2	5	1	2	5	1	2	5
	versions	4	6	1	10	43	100	23	100	100	7	20	11
MBD	$\epsilon^{(0)}$	13.7	18.2	22.8	21.6	26.1	30.8	16.2	25.1	33.8	17.2	23.5	28.6
	$\epsilon^{(1)}$	1.2	2.4	5.0	4.2	7.6	14.5	3.0	5.2	12.5	0.8	1.6	3.0
	$\epsilon^{(2)}$	1.2	2.4	4.8	5.1	8.9	15.5	3.0	5.2	12.4	0.8	1.5	3.1
	Barinel	**1.2**	**2.4**	**4.4**	**1.9**	**3.4**	**6.6**	**3.0**	**5.0**	**11.9**	**0.8**	**1.5**	**3.0**
SFL	Ochiai	2.6	5.3	11.5	3.9	7.0	13.5	3.0	5.6	12.4	1.1	2.0	3.7
	Tarantula	7.3	13.2	21.0	6.0	10.4	17.8	4.5	7.7	14.9	1.5	2.7	5.4

		schedule2			tcas			tot_info			space		
	C	1	2	5	1	2	5	1	2	5	1	2	5
	versions	9	35	91	30	100	100	19	100	100	28	100	100
MBD	$\epsilon^{(0)}$	29.3	**26.6**	**28.9**	28.0	26.9	28.7	14.0	18.2	21.5	19.5	25.2	34.3
	$\epsilon^{(1)}$	22.8	31.4	38.3	16.7	24.2	30.5	5.1	8.7	17.4	2.2	3.6	9.5
	$\epsilon^{(2)}$	21.5	29.4	35.6	16.7	24.1	30.5	6.1	11.7	20.9	2.2	3.7	9.9
	Barinel	**21.5**	28.1	34.9	16.7	24.5	30.7	**5.0**	**8.5**	**15.8**	**1.7**	**3.0**	**7.4**
SFL	Ochiai	21.5	29.1	35.5	**15.5**	**22.0**	**27.4**	5.2	9.1	16.5	1.7	3.6	8.6
	Tarantula	23.5	31.4	38.3	16.1	22.8	31.6	6.9	11.4	19.4	3.4	6.5	13.9

Table 5.5 Wasted effort W [%] on combinations of $C = 1 - 5$ faults for the Siemens set and space

equals $O(N \cdot M + M \cdot \log M)$, putting the Bayesian approaches in the same complexity class as the statistical approaches modulo a large factor. The results in Table 5.6 follow the trends predicted by this complexity analysis.

We measure the time efficiency by conducting our experiments on a 2.3 GHz Intel Pentium-6 PC with 4 GB of memory. As most fault localization techniques have been evaluated in the context of single faults, in order to allow us to compare our fault localization approach to related work we limit ourselves to the original, single-fault Siemens benchmark set, which is the common benchmark set to most fault localization approaches. We obtained timings for PPDG and DD from published results [Baah et al., 2008, Zeller, 2002].

Table 5.6 summarizes the results of the study. The columns show the programs, the average CPU time (in seconds) of Barinel, $\epsilon^{(0)}/\epsilon^{(1)}/\epsilon^{(2)}$, Tarantula/Ochiai, PPDG, and DD, respectively. As expected, the less expensive techniques are the statistics-based techniques Tarantula and Ochiai. At the other extreme are PPDG and DD. Barinel costs less than PPDG and DD. For example, Barinel requires less than 10 seconds on average for replace, whereas PPDG needs 6 minutes and DD needs approximately 1 hour to produce the diagnostic report. Note that our implementation of Barinel has not been optimized (the gradient ascent algorithm). This explains the fact that Barinel is more expensive than the other Bayesian approaches. The effect of the gradient ascent costs is clearly noticeable for the first three programs, and is due to a somewhat lower convergence speed as a result of the fact that the h_j are close to 1. Note, that by using a procedure with quadratic convergence this difference would largely disappear (e.g., 100 iterations instead of 10,000, gaining two orders of magnitude). Therefore, the efficiency results should not

Program	Barinel	$\epsilon^{(0,1,2)}$	Tarantula/Ochiai	PPDG	DD
print_tokens	24.3	4.2	0.37	846.7	2590.1
print_tokens2	19.7	4.7	0.38	243.7	6556.5
replace	9.6	6.2	0.51	335.4	3588.9
schedule	4.1	2.5	0.24	77.3	1909.3
schedule2	2.9	2.5	0.25	199.5	7741.2
tcas	1.5	1.4	0.09	1.7	184.8
tot_info	1.5	1.2	0.08	97.7	521.4
space	41.4	7.4	0.15	N/A	N/A

Table 5.6 Diagnosis cost for the single-fault Siemens benchmark set (time in seconds)

be viewed as definitive. Experiments using the extended Siemens benchmark set to accommodate multiple faults also show the same trend. Although not listed in the table, the execution time of Barinel for space (10kLOC) is 48s (no results of related work are available).

With respect to space complexity, statistical techniques need two store the counters $(n_{11}, n_{10}, n_{01}, n_{00})$ for the similarity computation for all M components. Hence, the space complexity is $O(M)$. $\epsilon^{(0)}$, $\epsilon^{(1)}$, and $\epsilon^{(2)}$ also store similar counters but per diagnosis candidate. Assuming that $|D|$ scales with M, these approaches have $O(M)$ space complexity. Barinel is slightly more expensive because for a given diagnosis d_k it stores the number of times a combination of faulty components in d_k is observed in passed runs $(2^{|d_k|} - 1)$ and in failed runs $(2^{|d_k|} - 1)$. Thus, Barinel's space complexity is estimated to be $O(2^C \cdot M)$ - being slightly more complex than SFL. In practice, however, memory consumption is reasonable (e.g., around 3.7 MB for space).

5.5 RELATED WORK

In model-based reasoning approaches to automatic software debugging the model of the program under analysis is typically generated using static analysis. In the work of Mayer and Stumptner [Mayer and Stumptner, 2008] an overview of techniques to automatically generate program models from the source code is given. They conclude that models generated by means of abstract interpretation [Mayer and Stumptner, 2007b] are the most accurate for debugging. In [Wotawa, 2002] the relationship between program slicing [Tip, 1995] and model-based software debugging is described. Model-based approaches include the Δ-slicing and `explain` work of Groce [Groce, 2004], and the work of Wotawa, Stumptner, and Mayer [Wotawa et al., 2002]. Although model-based diagnosis inherently considers multiple faults, thus far the above software debugging approaches only consider single faults. Apart from this, our approach differs in the fact that we use program spectra as dynamic information on component activity, which allows us to exploit execution behavior, unlike static approaches. Besides, our approach does not rely on the approximations required by static techniques (i.e., incompleteness). Most importantly, our approach is less complex, as can also be deduced by the limited set of programs used by the model-based techniques (as an indication, from the Siemens set, these techniques can only handle `tcas` which is the smallest program). In [Wieland, 2001] a similar trace-based dynamic dependency model has been proposed. The main difference to our work is that we do not exploit component (execution) dependencies.

Essentially all of the above work have mainly been studied in the context of single faults (e.g., Siemens set), except for recent work by Jones, Bowring, and Harrold [Jones et al., 2007], Abreu, Zoeteweij, and Van Gemund [Abreu et al., 2008c], and Steimann and Bertchler [Steimann and Bertchler, 2009], who all take an explicit multiple-fault, spectrum-based approach. The work in [Jones et al., 2007] employs clustering techniques to identify traces (rows in A) which refer to the same fault, after which Tarantula is applied to each cluster of rows. In this clustering approach there is a possibility that multiple developers will still be effectively fixing the same bug. As the experimental environment in [Jones et al., 2007] is different from the one in this chapter, no comparison was possible. The significant difference between our previous work in [Abreu et al., 2008c, Abreu et al., 2008d] and our approach in this chapter is (1) the maximum likelihood health estimation algorithm, replacing the previous, approximate approach, and (2) the use of a heuristic reasoning algorithm to bound the number of multiple-fault candidates. In [Steimann and Bertchler, 2009] another ranking mechanism is introduced for diagnosis candidates that are derived using a similar technique as in [Abreu et al., 2008d], which has exponential time complexity. Unlike our work, neither [Jones et al., 2007] nor [Steimann and Bertchler, 2009] present results on the Siemens set in terms of established effort metrics, prohibiting any form of direct comparison. To our knowledge, there is also no implementation available of these techniques, so we cannot evaluate them.

5.6 SUMMARY

In this chapter we have presented a multiple-fault localization technique, coined BARINEL, which is based on the dynamic, spectrum-based approach from statistical fault localization methods, combined with a probabilistic reasoning approach from model-based diagnosis, inspired by our previous work in both separate disciplines [Abreu et al., 2007, Abreu et al., 2008c, Abreu et al., 2008d]. BARINEL employs low-cost, approximate reasoning, employing a novel, maximum likelihood estimation approach to compute the health probabilities per component at a comparable time and space complexity to current SFL approaches.

Apart from a formal proof of BARINEL's optimality in the single-fault case, synthetic experiments with multiple injected faults have confirmed that our approach consistently outperforms other spectrum-based approaches, such as the Tarantula tool and previous Bayesian reasoning approaches. Application to a set of software programs (Siemens set, `space`) also indicates BARINEL's advantage (20 wins out of 24 trials, despite the significant variance), while the exceptions can be pointed to particular program properties in combination with sampling noise. Although being more expensive than statistics-based techniques, our approach is a comparable complexity class, which, after further optimization, makes it quite amenable to large software systems.

A Low-Cost Approximate Minimal Hitting Set Algorithm

6

ABSTRACT

In Chapter 5 we have used a minimal hitting set algorithm to compute a set of high-potential multiple-fault candidates that were subject to a subsequent Bayesian reasoning approach to rank them. Generating minimal hitting sets of a collection of sets is known to be NP-hard, necessitating heuristic approaches to handle large problems. In this chapter a low-cost, approximate minimal hitting set (MHS) algorithm, coined Staccato, is presented. Staccato uses a heuristic function borrowed from SFL to guide the MHS search. Given the nature of the heuristic function, Staccato is specially tailored to model-based diagnosis problems (where each MHS solution is a diagnosis to the problem), although well-suited for other application domains. We apply Staccato in the context of model-based diagnosis and show that even for small problems our approach is orders of magnitude faster than the brute-force approach, while still capturing all important solutions - as the diagnostic accuracy of both approaches is essentially the same. Due to its low cost complexity, we also show that Staccato is amenable to large problems including millions of variables.

—— // ——

Identifying minimal hitting sets (MHS) of a collection of sets is an important problem in many domains, such as in model-based diagnosis (MBD) where the MHS are the solutions for the diagnostic problem. Known to be a NP-hard problem [Garey and Johnson, 1979], the usage of exhaustive algorithms, e.g. [Reiter, 1987, Greiner et al., 1989, Wotawa, 2001], is prohibitive for large-scale problems. To decrease the cost complexity of MHS algorithms, rendering them amenable to large problems, one

1. uses focusing heuristics to increase the search efficiency and/or

2. limits the size of the return set.

Such strategies have the potential to reduce the MHS problem to a polynomial time complexity at the cost of completeness.

In this chapter, we present an algorithm, coined Staccato[1], to derive an approximate collection of MHS that uses an heuristic borrowed from SFL (see Chapter 2 for a detailed definition). SFL uses sets of component involvement in nominal and failing program executions to yield a ranking of components in order of likelihood to be at fault. We show that this ranking heuristic is suitable to derive the MHS solutions in the MBD domain as the search is focused by visiting solutions in best-first order (aiming to capture the most relevant probability mass in the shortest amount of time). Although the heuristic originates from the MBD domain, it can also be used for other problem domains. We also introduce a search pruning parameter λ and a search truncation parameter L. λ specifies the percentage of top components in the ranking that should be considered, using the knowledge that only most probable solutions are visited. Taking advantage of the fact that most relevant solutions are visited first, the search can be truncated after L solutions are found, avoiding the generation of a myriad of solutions.

In particular, this chapter makes the following contributions:

- We present a new algorithm Staccato, and derive its time and space complexity;
- We compare Staccato with a brute-force approach using synthetic data as well as data collected from a real software program;
- We investigate the impact of λ and L on Staccato's cost/completeness trade-off.

To the best of knowledge this heuristic approach has not been presented before and has proven to have a significant positive effect on MBD complexity in practice (see Chapter 5).

The reminder of this chapter is organized as follows. We start by introducing the MHS problem. Subsequently, Staccato is described, followed by a derivation of the time/space complexity. The experimental results are then presented, followed by a discussion of related work. Finally, a summary of this chapter is given.

6.1 MINIMAL HITTING SET PROBLEM

In this section we describe the minimal hitting set (MHS) problem, and the concepts used throughout this chapter.

Let S be a collection of N non-empty sets $S = \{s_1, \ldots s_N\}$. Each set $s_i \in S$ is a finite set of elements (components from now on), where each of the M elements is represented by a number $j \in \{1, \ldots, M\}$. A minimal hitting set of S is a set d such that

$$\forall s_i \in S, s_i \cap d \neq \emptyset \;\wedge \nexists d' \subset d : s_i \cap d' \neq \emptyset$$

[1]Staccato is an acronym for STAtistiCs-direCted minimAl hiTting set algOrithm.

i.e., each member of S has at least one component of d as a member., and no proper subset of d is a hitting set. There may be several minimal hitting sets for S, which constitutes a collection of minimal hitting sets $D = \{d_1, \ldots, d_k, \ldots, d_{|D|}\}$. The computation of this collection D is known to be a NP-hard problem [Garey and Johnson, 1979].

In the remainder of this chapter, the collection of sets S is encoded into a $N \times M$ (binary) matrix A (which is similar to the activity matrix in SFL). An element a_{ij} is equal to 1 if component j is a member of set i, and 0 otherwise. For $j \leq M$, the row A_{i*} indicates whether a component is a member of set i, whereas the column A_{*j} indicates which sets component j is a member. As an example, consider the set $S = \{\{1,3\},\{2,3\}\}$ for $M = 3$, represented by the matrix

1	2	3	
1	0	1	first set
0	1	1	second set

A naïve, brute-force approach to compute the collection D of minimal hitting sets for S would be to iterate through all possible component combinations to (1) check whether it is a hitting set, and (2) and (if it is a hitting set) whether it is minimal, i.e., not subsumed by any other set of lower cardinality (cardinality of a set d_k, $|d_k|$, is the number of elements in the set). As all possible combinations are checked, the complexity of such an approach is $O(2^M)$. For the example above, the following sets would be checked: $\{1\},\{2\},\{3\},\{1,2\},\{1,3\},\{2,3\},\{1,2,3\}$ to find out that only {3} and {1,2} are minimal hitting sets of S.

6.2 STACCATO

As explained in the previous section, brute-force algorithms have a cost that is exponential in the number of components. Since many of the potential solution candidates turn out to be no minimal hitting set, a heuristic that *focuses* the search towards high-potentials will yield significant efficiency gains. In addition, many of the computed minimal hitting sets may potentially be of little value for the problem that is being solved. Therefore, one would like the solutions to be *ordered* in terms of relevance, possibly terminating the search once a particular number of minimal hitting sets have been found, again boosting efficiency. In this section we present our approximate, statistics-directed minimal hitting set algorithm, coined STACCATO, aimed to increase search efficiency.

The key idea behind our approach is the fact, that components that are members of more sets than other components, may be an indication that there is a minimal hitting set containing such component. The trivial case are those components that are *involved* in all sets, which constitute a minimal hitting set of cardinality 1. A simple search heuristic is to exploit a ranking based on the number of set involvements such as

$$\mathcal{H}(j) = \sum_{i=1}^{N} a_{ij}$$

To illustrate, consider again the example above. Using the heuristic function $\mathcal{H}(j)$, it follows that $\mathcal{H}(1) = 1$, $\mathcal{H}(2) = 1$, and $\mathcal{H}(3) = 2$, yielding the ranking $< 3, 1, 2 >$. This ranking is exploited to guide the search. Starting with component 3, it appears that it is involved in the two sets, and therefore is a minimal hitting set of minimal cardinality. Next in the ranking comes component 1. As it is not involved in all sets, it is combined with those components that are involved in all sets except the ones already covered by 1 (note, that combinations involving 3 are no longer considered due to subsumption). This would lead us to find {1,2} as a second minimal hitting set.

Although this heuristic avoids having to iterate through all possible component combinations ($O(2^M)$), it may still be the case that many combinations have to be considered. For instance, using the heuristic one has to check 3 sets, whereas the brute-force approach iterates over 8 sets. Consequently, we introduce a parameter λ that contains the fraction of the ranking that will be considered. The reasoning behind this parameter is that the components that are involved in most sets (ranked high by $\mathcal{H}$) are more likely to be a minimal hitting set. Clearly, λ cannot be too small. In the previous example, if λ would be set to $\lambda = 1/3$, only element 3 would be considered, and therefore we would miss the solution set {1,2}. Hence, such a parameter trades efficiency for completeness.

6.2.1 *Approximation*

While the above heuristic increases search efficiency, the number of minimal hitting sets can be prohibitive, while often only a subset need be considered that are most relevant with respect to the application context. Typically, approaches to compute the minimal hitting set are applied in the context of (cost) optimization problems. In such case, one is often interested in finding the minimal hitting set of minimal cardinality. For example, suppose one is responsible for assigning courses to teachers. Due to lack of funds, one proposes to minimize the number of teachers that need to be hired. Hence, one would like to find the minimal number of teachers that can teach all courses, which can be solved by formulating the problem as a minimal hitting set problem. For this example, solutions with low cardinality (i.e., number of teachers) are more attractive than those with higher cardinality. The brute-force approach, as well as the above heuristic approach are examples of approaches that find minimal hitting sets with lower cardinality first.

In many situations, however, obtaining MHS solutions in order of just cardinality does not suffice. An example is model-based diagnosis (MBD) where the minimal hitting sets represent fault diagnosis candidates, each of which has a certain probability of being the actual diagnosis. The most cost-efficient approach is to generate the MHS solutions in decreasing order of probability (minimizing average fault localization cost). Although probability typically decreases with increasing MHS cardinality, cardinality is not sufficient, as, e.g., there may be a significant probability difference between diagnosis (MHS) solutions of equal cardinality (of which there may be many). Consequently, a heuristic that predicts probability rather than just cardinality makes the difference. The fact that the MHS solutions are now generated in decreasing order of probability allows us to truncate the number of solutions, where, e.g., one only considers the MHS subset of L solutions that covers .99 probability mass, ignoring the (many) improbable solutions. This *approximation* trades limited cost penalty (completeness) for significant efficiency gains.

6.2.2 *Model-Based Diagnosis*

In this section we extend our above heuristic for use in MBD, as outlined in Chapter 5, where conflicts are deduced from the activity matrix A and are input to the MHS algorithm to deduce the multiple-fault diagnosis candidates. As mentioned in Chapter 5, the MHS solutions d_k are ranked in order of probability of being the true fault explanation $\Pr(d_k)$, which is computed using Bayes' update according to

$$\Pr(d_k|obs_i) = \frac{\Pr(obs_i|d_k)}{\Pr(obs_i)} \cdot \Pr(d_k|obs_{i-1}) \tag{6.1}$$

where obs_i denotes observation i. In the context of this chapter, an observation obs_i stands for a conflict set s_i that results from a particular observation. The denominator $\Pr(obs_i)$ is a normalizing term that is identical for all d_k and thus needs not be computed directly. $\Pr(d_k|obs_{i-1})$ is the prior probability of d_k, before incorporating the new evidence obs_i. For $i = 1$ $\Pr(d_k)$ is defined in a way such that it ranks components of lower cardinality higher in absence of any observation. $\Pr(obs_i|d_k)$ is defined as

$$\Pr(obs_i|d_k) = \begin{cases} 0 & \text{if } obs_i \wedge d_k \models \perp \\ 1 & \text{if } d_k \rightarrow obs_i \\ \varepsilon & \text{otherwise} \end{cases}$$

In MBD, many policies exist for ε based on the chosen modeling strategy. In the context of this chapter we use the $\epsilon^{(2)}$ strategy defined in Chapter 5.

Given a sequence of observations (conflicts), the MHS solutions should be ordered in terms of Eq. (6.1). However, using Eq. (6.1) as heuristic is computationally prohibitive as it has exponential complexity (e.g., in M). Clearly, a low-cost heuristic that still provides a good prediction of Eq. (6.1) is crucial if Staccato is to be useful in MBD.

$$\begin{array}{cc}
 & M \text{ components} \qquad\qquad \text{conflict} \\
N \text{ sets} & \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1M} \\ a_{21} & a_{22} & \cdots & a_{2M} \\ \vdots & \vdots & \ddots & \vdots \\ a_{N1} & a_{M2} & \cdots & a_{NM} \end{bmatrix} \quad \begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_N \end{bmatrix}
\end{array}$$

Figure 6.1 Encoding for a collection of sets

6.2.3 *An MBD Heuristic*

A low-cost, statistics-based technique that is known to be a good predictor for ranking (software) faults in order of likelihood is SFL, as demonstrated in Chapters 2 and 3. To comply with SFL, we extend A into a pair (A, e) (see Figure 6.1), where e is a binary array which indicates whether the A_{i*} corresponds to erroneous system behavior ($e = 1$) or nominal behavior ($e = 0$). Many similarity coefficients exist for SFL, the best one currently being the Ochiai coefficient known from molecular biology and introduced to SFL in Chapter 2 (see Eq. (2.3) in Section 2.1.2 of Chapter 2). In Chapter 2, it has been shown that similarity coefficients provide an ordering of components that yields good diagnostic accuracy, i.e., components that rank high are usually faulty. This diagnostic performance, combined with the very low complexity of $s(j)$ is the key motivation to use the Ochiai coefficient $s(j)$ for $\mathcal{H}$. Thus,

$$s(j) = \frac{n_{11}(j)}{\sqrt{(n_{11}(j) + n_{01}(j)) * (n_{11}(j) + n_{10}(j))}} \tag{6.2}$$

where

$$n_{pq}(j) = |\{i \mid a_{ij} = p \wedge e_i = q\}|$$

If (A, e) only contains conflicts (i.e., $\nexists e_i = 0$), the ranking returned by this heuristic function reduces to the original one

$$\mathcal{H}(j) = \sum_{i=1}^{N} a_{ij} = n_{11}(j) \tag{6.3}$$

and, therefore, classic MHS problems are also adequately handled by this MBD heuristic.

6.2.4 *Algorithm*

Staccato uses the SFL heuristic Eq. (6.2) to focus the search of the minimal hitting set computation (see Algorithm 3). To illustrate how Staccato works, consider the following (A, e), comprising two (conflict) sets originating from erroneous system behavior and one set corresponding to component involvement in nominal system behavior.

1	2	3	e_i	
1	0	1	1	first set (error)
0	1	1	1	second set (error)
1	0	1	0	third set (nominal)

From (A, e) it follows $\mathcal{H}(1) = 0.5$, $\mathcal{H}(2) = 0.7$, and $\mathcal{H}(3) = 1$, yielding the following ranking $< 3, 2, 1 >$. As component 3 is involved in all failed sets, it is added to the minimal hitting set and removed from A using function Strip_Component, avoiding solutions subsumed by $\{3\}$ to be considered (lines 5–12). After this phase, the (A, e) is as follows

1	2	e_i
1	0	1
0	1	1
1	0	0

Next component to be checked is component 2, which is not involved in one failed set. Thus, the column for that component as well as all conflict sets in which it is involved are removed from (A, e), using the Strip function, yielding the following

1	e_i
1	1
1	0

Running Staccato with the newly generated (A, e) yields a ranking with component 1 only (line 17), which is a MHS for the current (A, e). For each MHS d returned by this invocation of Staccato, the union of d and component 2 is checked ($\{1, 2\}$), and because this set is involved in all failed sets, and is minimal, it is also added to the list of solutions D (lines 18–24). The same would be done for component 1, the last in the ranking, but no minimal set would be found. Thus, Staccato would return the following minimal hitting sets $\{\{3\}, \{1, 2\}\}$. Note that this heuristic ranks component 2 on top of component 1, whereas the previous heuristic ranked component 1 and 2 at the same place (because they both explained the same number of conflicts).

Staccato calls SFL C times to find a diagnosis candidate with cardinality C. For instance, Figure 6.2 depicts the workflow for generating a candidate with cardinality 3, the only solution for the following (A, e):

1	2	3	e_i
1	0	0	1
0	1	0	1
0	0	1	1
0	0	1	1
1	0	0	0

In the figure, $(A, e)^{(1)}$ and $(A, e)^{(2)}$ represent the intermediate matrices.

In summary, Staccato comprises the following steps

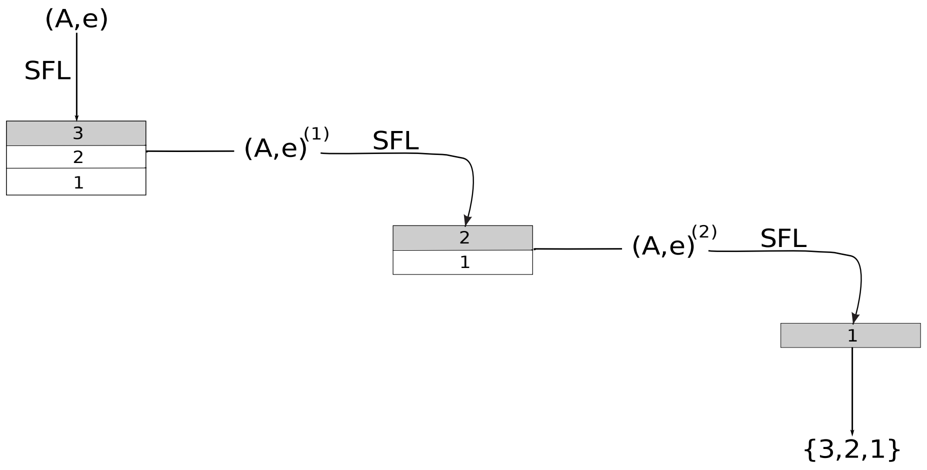


Figure 6.2 Example workflow to generate a candidate with cardinality 3

- Initialization phase, where a ranking of components using the heuristic function borrowed from SFL is computed (lines 1–4 in Algorithm 3);
- Components that are involved in all failed sets are added to D (lines 5–12);
- While $|D| < L$, for the first top λ components in the ranking (including also the ones added to D, lines 13-25) do the following:
  - remove the component j and all A_{i*} for which $e_i = 1 \wedge a_{ij} = 1$ holds from (A, e) (line 17);
  - run Staccato with the new (A, e), and
  - combine the solutions returned with the component and verify whether it is a minimal hitting set (lines 17–24).

6.2.5 *Complexity Analysis*

To find a minimal hitting set of cardinality C Staccato has to be (recursively) invoked C times. Each time it (1) updates the four counters per component ($O(N \cdot M)$), (2) ranks components in fault likelihood ($O(M \cdot \log M)$), (3) traverses λ components in the ranking ($O(M)$), and (4) checks whether it covers all failed sets ($O(N)$). Hence, the overall time complexity of Staccato is merely $O((M \cdot (N + \log M))^C)$. In practice, however, due to the search focusing heuristic the time complexity is merely $O(C \cdot M \cdot (N + \log M))$ (confirmed by measurements in Section 6.3).

With respect to space complexity, for each invocation of Staccato, it has to store four counters per component to create the SFL-based ranking

Algorithm 3 Staccato

```
Input: Matrix $(A, e)$, number of components $M$, stop criteria $\lambda$, $L$
Output: Minimal Hitting set $D$
1  $T_F \leftarrow \{A_{i*} | e_i = 1\}$        ▷ Collection of conflict sets
2  $R \leftarrow$ RANK$(\mathcal{H}, A, e)$
3  $D \leftarrow \emptyset$
4  $seen \leftarrow 0$
5  for all $j \in \{1..M\}$ do
6      if $n_{11}(j) = |T_F|$ then
7          PUSH$(D, \{j\})$
8          $A \leftarrow$ STRIP_COMPONENT$(A, j)$        ▷ Remove $c_j$ from $A$
9          $R \leftarrow R \backslash \{j\}$
10         $seen \leftarrow seen + \frac{1}{M}$
11     end if
12 end for
13 while $R \neq \emptyset \wedge seen \leq \lambda \wedge |D| \leq L$ do
14     $j \leftarrow$ POP$(R)$
15     $seen \leftarrow seen + \frac{1}{M}$
16     $(A', e') \leftarrow$ STRIP$(A, e, j)$
17     $D' \leftarrow$ STACCATO $(A', e', \lambda, L)$
18     while $D' \neq \emptyset$ do
19         $j' \leftarrow$ POP$(D')$
20         $j' \leftarrow \{j\} \cup j'$
21         if IS_NOT_SUBSUMED$(D, j')$ then
22             PUSH$(D, j')$
23         end if
24     end while
25 end while
26 return $D$
```

$(n_{11}, n_{10}, n_{01}, n_{00})$. As the recursion depth is C to find a solution of the same cardinality, Staccato has a space complexity of $O(C \cdot M)$.

6.3 EXPERIMENTAL RESULTS

In this section we present our experimental results using synthetic data and data collected from a real software program.

6.3.1 *Synthetic Diagnosis Experiments*

In order to assess the performance of our algorithm we use the synthetic (A, e) sets generated for the diagnostic algorithm research described in Chapter 5, generated for various values of N, M, and the number of injected faults C

(cardinality). Component activity a_{ij} is sampled from a Bernoulli distribution with parameter r, i.e., the probability a component is involved in a row of A equals r. For the C faulty components c_j (without loss of generality we select the first C components, i.e., $c_1, \ldots, c_C$ are faulty). We also set the probability a faulty component behaves as expected h_j. Thus the probability of a component j being involved *and* generating a failure equals $r \cdot (1 - h_j)$. A row i in A generates an error ($e_i = 1$) if at least 1 of the C components generates a failure (or-model). Measurements for a specific scenario are averaged over 500 sample matrices.

Table 6.1 summarizes the results of our study for $r = 0.6$ (typical value for software), $M = 20$ and $N = 300$, which is the limit for which a brute-force approach is feasible. Per scenario, we measure the number of MHS solutions ($|D|$), the computation CPU time (T), the completeness ρ per C (the ratio of solutions with cardinality C found using Staccato and the brute-force approach, indication of the heuristic's search focusing ability), and the diagnostic performance (W) for the brute-force approach (B-F) and Staccato with several λ parameters. The completeness ρ values presented are per cardinality - separated by a '/' - where the last value is the percentage of solutions missed with $C \geq 6$. Diagnostic performance is measured in terms of a diagnostic performance metric W that measures the percentage of excess *work* incurred in finding the actual components at fault, a typical metric in software debugging [Abreu et al., 2007], after ranking the MHS solutions using the Bayesian policy described in [Abreu et al., 2008c]. For instance, consider a $M = 5$ component program with the following diagnostic report $D =< \{4, 5\}, \{1, 2\} >$, while components 1 and 2 are actually faulty. The first diagnosis candidate leads the developer to inspect components 4 and 5. As both components are healthy, W is increased with $\frac{2}{5}$. The next components to be inspected are components 1 and 2. As they are both faulty, no more wasted effort is incurred. After repairing these two components, the program would be re-run to verify that all test cases pass. Otherwise, the debugging process would start again until no more test cases fail.

As expected, $|D|$ and the time needed to compute D decreases with λ. Although some solutions are missed for low values of λ, they are not important for the diagnostic problem as W does not increase. This suggest that our heuristic function captures the most probable solutions to be faulty. An important observation is that for $\lambda = 1$, the results are essentially the same as an exhaustive search but with several orders of magnitude speed-up. Furthermore, we also truncated $|D|$ to 100 to investigate the impact of this parameter in the diagnostic accuracy for $\lambda = 1$. Although it has a small negative impact on W, it reduces the time needed to compute W by more than half. For instance, for $C = 5$ and $h = 0.1$ it takes 0.008 s to generate D, demanding the developer to waste more effort to find the faulty components, $W = 10\%$.

We have not presented results for other settings of M, N because the brute-force approach does not scale. However, we observed the same trends with Staccato as the ones presented. As an example, for $M = 1{,}000{,}000$, $N =$

		h	0.1		0.9	
		C	1	5	1	5
B-F		$\|D\|$	355	508	115	286
		T (s)	25.5	54.3	0.27	5.72
		W (%)	0.0	13	14	21
Staccato	$\lambda = 0.1$	$\|D\|$	63	127	10	46
		T (s)	0.006	0.007	0.001	0.003
		ρ (%)	0/0/0/65/87/0	0/0/41/95/60/82	0/44/100/0/0/0	0/0/42/88/0/0
		W (%)	0.0	10.7	0.0	12.9
	$\lambda = 0.2$	$\|D\|$	86	181	16	63
		T (s)	0.008	0.009	0.02	0.003
		ρ (%)	0/0/0/54/87/0	0/0/30/87/59/70	0/31/100/0/0/0	0/0/36/88/0/0
		W (%)	0.0	9.2	0.0	13.9
	$\lambda = 0.3$	$\|D\|$	112	232	26	75
		T (s)	0.009	0.010	0.003	0.004
		ρ (%)	0/0/0/30/74/0	0/0/21/87/53/65	0/25/73/0/0/0	0/0/26/75/0/0
		W (%)	0.0	9.1	0.0	14.4
	$\lambda = 0.4$	$\|D\|$	175	276	47	83
		T (s)	0.011	0.012	0.004	0.004
		ρ (%)	0/0/0/26/75/0	0/0/21/87/18/64	0/6/72/0/0/0	0/0/10/63/0/0
		W (%)	0.0	9.2	0.0	13.8
	$\lambda = 0.5$	$\|D\|$	218	300	67	146
		T (s)	0.013	0.018	0.004	0.007
		ρ (%)	0/0/0/23/66/0	0/0/10/87/6/65	0/0/64/0/0/0	0/0/5/56/0/0
		W (%)	0.0	9.0	0.0	14.3
	$\lambda = 0.6$	$\|D\|$	253	372	83	180
		T (s)	0.015	0.019	0.004	0.008
		ρ (%)	0/0/0/20/61/0	0/0/0.08/65/0/65	0/0/39/0/0/0	0/0/0/46/0/0
		W (%)	0.0	8.8	0.0	14.7
	$\lambda = 0.7$	$\|D\|$	293	425	87	199
		T (s)	0.019	0.025	0.005	0.008
		ρ (%)	0/0/0/11/50/0	0/0/0.06/54/0/55	0/0/39/0/0/0	0/0/0/44/0/0
		W (%)	0.0	8.6	0.0	14.4
	$\lambda = 0.8$	$\|D\|$	343	449	109	228
		T (s)	0.023	0.028	0.06	0.009
		ρ (%)	0/0/0/7/26/0	0/0/0.02/38/0/24	0/0/24/0/0/0	0/0/0/32/0/0
		W (%)	0.0	8.8	0.0	14.8
	$\lambda = 0.9$	$\|D\|$	355	508	115	270
		T (s)	0.024	0.034	0.08	0.012
		ρ (%)	0/0/0/0/0/0	0/0/0/15/0/10	0/0/0/0/0/0	0/0/0/13/0/0
		W (%)	0.0	9.0	0.0	14.8
	$\lambda = 1$	$\|D\|$	355	508	115	286
		T (s)	0.025	0.041	0.010	0.016
		ρ (%)	0/0/0/0/0/0	0/0/0/0/0/0	0/0/0/0/0/0	0/0/0/0/0/0
		W (%)	0.0	9.0	0.0	14.9

Table 6.1 Results for the synthetic matrices

$1,000$, and $C = 1,000$, the candidate generation rate with Staccato is 4.1 ms on average (2.1 ms for $C = 100$).

6.3.2 *Real Software Analysis for Diagnosis*

In this section we apply the Staccato algorithm in the context of model-based software fault diagnosis, namely to derive the set of valid diagnoses given a set of observations (test cases). We use the `tcas` program which can be obtained from the software infrastructure repository (SIR, [Do et al., 2005]). The other programs in SIR were not used because the brute-force approach cannot handle them. TCAS (Traffic Alert and Collision Avoidance System) is an aircraft conflict detection and resolution system used by all US commercial aircraft. The SIR version of `tcas` includes 41 faulty versions of ANSI-C code for the resolution advisory component of the TCAS system. In addition, it also provides a correct version of the program and a pool containing $N = 1,608$ test cases. `tcas` has $M = 178$ lines of code, which, in the context of the following experiments, are the number of components. In our experiments, we randomly injected C faults in one program. All measurements are averages over 100 versions, except for the single fault programs which are averages over the 41 available faults. The activity matrices are obtained using the GNU `gcov`[2] profiling tool and a script to translate its output into a matrix. As each program suite includes a correct version, we use the output of the correct version as reference. We characterize a run/computation as failed if its output differs from the corresponding output of the correct version, and as passed otherwise.

Table 6.2 presents a summary of the results obtained using a brute-force approach (B-F) and Staccato with different λ parameters. Again, we report the size of the minimal hitting set ($|D|$), the time T required to generate D, and the diagnostic performance incurred by the different settings. As expected, the brute-force approach is the most expensive of them all. The best trade-off between complexity and the diagnostic cost W is for $\lambda \approx 0.5$, since Staccato does not miss important candidates - judging by the fact that W is essentially the same as the brute-force approach - and it is faster than for other, higher λ.

Although Staccato was applied to other, bigger software programs (see Chapter 5), no comparison is given as the brute-force algorithm does not scale. As an indication, for a given program with $M = 10,000$ lines of code and $N = 132$ test cases, Staccato required roughly 1 s to compute the relevant MHS solutions (for $\lambda = 0.5$ and $L = 100$). In addition, in these experiments, using the well-known Siemens benchmark set of software faults and the `space` program, $L = 100$ was proven to already yield comparable results to those obtained for $L = \infty$ (i.e., generating all solutions).

[2]http://gcc.gnu.org/onlinedocs/gcc/Gcov.html

	tcas				
	C		1	2	5
	#matrices		41	100	100
B-F	$\lvert D \rvert$		76	59	68
	T (s)		0.98	2.1	11.2
	W (%)		16.7	23.7	29.7
Staccato	$\lambda = 0.1$	$\lvert D \rvert$	30	35	61
		T (s)	0.11	0.16	0.22
		W	15.2	29.3	37.1
	$\lambda = 0.2$	$\lvert D \rvert$	34	39	62
		T (s)	0.15	0.17	0.25
		W	15.2	29.3	37.0
	$\lambda = 0.3$	$\lvert D \rvert$	50	44	63
		T (s)	0.18	0.18	0.26
		W	16.2	28.8	37.1
	$\lambda = 0.4$	$\lvert D \rvert$	51	58	65
		T (s)	0.19	0.19	0.27
		W	16.3	23.7	32.1
	$\lambda = 0.5$	$\lvert D \rvert$	76	58	66
		T (s)	0.20	0.20	0.30
		W	16.7	23.7	30.1
	$\lambda = 0.6$	$\lvert D \rvert$	76	59	67
		T (s)	0.22	0.22	0.31
		W	16.7	23.7	29.7
	$\lambda = 0.7$	$\lvert D \rvert$	76	59	68
		T (s)	0.23	0.25	0.34
		W	16.7	23.7	29.7
	$\lambda = 0.8$	$\lvert D \rvert$	76	59	68
		T (s)	0.24	0.26	0.35
		W	16.7	23.7	29.7
	$\lambda = 0.9$	$\lvert D \rvert$	76	59	68
		T (s)	0.27	0.27	0.37
		W	16.7	23.7	29.7
	$\lambda = 1$	$\lvert D \rvert$	76	59	68
		T (s)	0.30	0.28	0.48
		W	16.7	23.7	29.7

Table 6.2 Results for `tcas`

6.4 RELATED WORK

Several algorithms have been presented to solve the MHS problem. Exhaustive approaches, which are in general independent of the application domain, include the following works. Since Reiter [Reiter, 1987] showed that diagnoses are MHSs of conflict sets, many approaches to solve this problem in this context have been presented. In [Greiner et al., 1989, de Kleer and Williams, 1987, Reiter, 1987, Wotawa, 2001] the hitting set problem is solved using so-called hit-set trees. In [Fijany and Vatan, 2004, Fijany and Vatan, 2005] the MHS problem is mapped onto an 1/0-integer programming problem. Contrary to our work, their approach, which also targets the model-based diagnosis problem, does not use any other information but the conflict sets. The integer programming approach has the potential so solve problems with thousands of variables but no complexity results are presented. In contrast, our low-cost approach can easily handle much larger problems. In [Zhao and Ouyang, 2007] a method using set-enumeration trees to derive all minimal conflict sets in the context of model-based diagnosis is presented. The authors conclude that this method has a exponential time complexity in the number of elements in the sets (components). The Quine-McCluskey algorithm [Quine, 1955], originating from logic optimization, is a method for deriving the prime implicants of a monotone boolean function (which is a dual problem of the MHS problem). This algorithm is, however, of limited use due to its exponential complexity, which has prompted the development of heuristics such as Espresso (discussed later on).

Many heuristic approaches have been proposed to render MHS computation amenable to large systems. In [Lin and Jiang, 2002, Lin and Jiang, 2003] an approximate method to compute MHSs using genetic algorithms is described. The fitness function used aims at finding solutions of minimal cardinality, which is less suitable for MBD as even solutions with similar cardinality have different probabilities of being the true fault explanation. Their paper does not present a time complexity analysis, but we suspect the cost/completeness trade-off to be worse than for Staccato. Stochastic algorithms, as discussed in the framework of constraint satisfaction [Freuder et al., 1995] and propositional satisfiability [Qasem and Prügel-Bennett, 2008], are examples of domain independent approaches to compute the MHS. Stochastic algorithms are more efficient than exhaustive methods. The Espresso algorithm [Rudell, 1986], primarily used to minimize logic circuits, uses a heuristics to guide the final result. Originating from logic circuits, it uses an heuristic to guide the circuit minimization that is specific for this domain. Due to its efficiency, this algorithm still forms the basis of every logic synthesis tool. Dual to the MHS problem, no prime implicants cost/completeness data is available to allow comparison with Staccato.

To our knowledge the statistics-based heuristic to guide the search for computing MHS solutions has not been presented before. Although the heuristic function used in our approach comes from a fault diagnosis approach, there is no reason to believe that Staccato will not work well in other domains.

6.5 SUMMARY

In this chapter we presented a low-cost approximate hitting set algorithm, coined Staccato, which uses an heuristic borrowed from a low-cost, statistics fault diagnosis tool, making it especially suitable to the model-based diagnosis domain.

Synthetic experiments have demonstrated that even for small problems our heuristic approach is orders of magnitude faster than exhaustive approaches, even when the algorithm is set to be complete. Our experiments have shown that search can be further focused using a parameter to prune the search space λ where completeness is hardly sacrificed for $\lambda \approx 0.5$. Compared to λ, the impact of truncating the number of solutions L in the set on cost is much greater. As most relevant solutions are visited first, the number of solutions returned to the user can be suitably truncated (e.g., only returning 100 candidates in the context of model-based diagnosis). Hence, a very attractive cost/completeness trade-off is reached by setting $\lambda = 1$ while limiting L.

7 Using SFL to Focus Model-based Software Debugging

ABSTRACT

SFL is a statistical technique that aims at helping software developers to find faults quickly by analyzing abstractions of program traces to create a ranking of most probable faulty components (e.g., program statements). Although spectrum-based fault localization has been shown to be effective, its diagnostic accuracy is inherently limited, since the semantics of components are not considered. In particular, components that exhibit identical execution patterns cannot be distinguished. To enhance its diagnostic quality, in this chapter, we combine spectrum-based fault localization with a model-based debugging approach based on abstract interpretation within a framework coined Deputo. The model-based approach is used to refine the ranking obtained from the spectrum-based method by filtering out those components that do not explain the observed failures when the program's semantics is considered. We show that this combined approach outperforms the individual approaches and other state-of-the-art automated debugging techniques.

//

Considerable costs are attached to locating and eliminating problems in software systems during development as well as after deployment [RTI, 2002]. Hence, numerous approaches have been proposed to automate parts of the testing and debugging process to help detect more defects earlier in the development cycle and to guide software engineers towards possible faults.

Statistical techniques are rather dependent on the availability of a suitable test suite. Better results can often be achieved if a model of the correct program behavior is available. Model-based software debugging (MBSD) techniques have been advocated as powerful debugging aid that isolate faults in complex programs [Mayer and Stumptner, 2008, Mayer, 2007]. By comparing the state and behavior of a program to what is anticipated by its programmer, model-based reasoning techniques separate those parts of a program that may contain a fault from those that cannot be responsible for observed symptoms. Although being competitive with other state of the art automated debugging approaches [Mayer and Stumptner, 2008], MBSD is computationally much more demanding than SFL and may still produce a large output that lacks ranking information.

In this chapter, we present a new framework, coined Deputo[1], that integrates SFL with MBSD to focus the search by filtering ranked results. Our approach first uses SFL to compute the ranked list of likely faulty components, and, subsequently, applies MBSD to refine the ranking by removing components that do not explain observed failures. Our algorithm combines the low computational complexity from SFL and the significantly improved diagnostic accuracy from MBSD. While MBSD is general enough to be combined with almost any debugging tool that can expose its findings in terms of the original program's source code and a set of fault assumptions, the combination of semantic and trace-based analysis (SFL) is particularly appealing, since the approaches contribute complementary information: MBSD injects and analyses specific modifications to the semantics of a program, while dynamic analysis exploits fault correlation to focus the search.

In this chapter, similar to the previous one, SFL is used as a focusing mechanism. In the previous chapter, SFL was used to derive the set of valid diagnosis candidates given the matrix. However, no program component topology is taken into account, many of the solutions generated can, in fact, be discarded. In this chapter, MBSD is used to find "mere coincidence" to definite explanations.

In particular, this chapter makes the following contributions

- We present a new algorithm, Deputo, that integrates MBSD with SFL to focus search and rank results.
- We show that, as a result, fewer program fragments are being implicated, leading to considerably increased accuracy.
- We show that our algorithm has low complexity, specially compared with MBSD, making it amenable to large programs.

The chapter is organized as follows. The principles of model-based debugging are outlined in Section 7.1. The combined framework is discussed in Section 7.2. Empirical validation of our approach and our findings are given in Section 7.3. Section 7.4 discusses relevant related works, followed by the summary.

7.1 MBSD

Statistics-based techniques, such as the one described on Chapter 2, are rather dependent on the availability of a suitable test harness. Better results can often be achieved if a model of the correct program behavior is available to guide debugging efforts, for example, a partial specification expressed in some formal language. Unfortunately, building such models is error-prone and prohibitively expensive for many software development scenarios. Attempts to

[1]Latin for pruning.

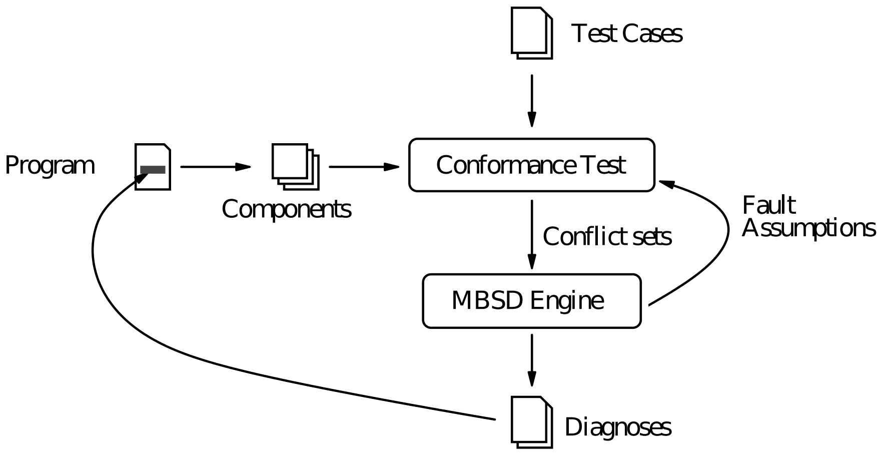


Figure 7.1 Model-based Software Debugging

devise formal specifications for non-trivial systems has shown that constructing a model that captures an abstraction of the semantics of a system can be as difficult and fallible as building a concrete implementation [Musuvathi and Engler, 2003].

Model-based diagnosis has been proven successful in aiding developers in locating the root cause of failures in physical systems by using a model of the systems' intended behavior [Reiter, 1987]. For software programs, however, creating such a model can be as difficult and error-prone as building the actual implementation [Musuvathi and Engler, 2003]. Model-based Software debugging [Mayer and Stumptner, 2008] (see Figure 7.1, which was taken from the website of the MBSD project [MBSD, 2008]) aims to close the gap between powerful formal analysis techniques and execution-based strategies in a way that does not require the end-user to possess knowledge of the underlying reasoning mechanisms. Here, an adaptation of the classic "reasoning from first principles" [Reiter, 1987] (that is, information directly available from program execution and source code) paradigm borrowed from diagnosis of physical systems is particularly appealing, since much of the complexity of the formal underpinnings of program analysis can be hidden behind an interface that resembles the end-user's traditional view of software development. In contrast to statistics-based approaches, MBSD is less dependent on large test suites as it exploits a model of normal behavior.

In contrast to classical model-based diagnosis, where a correct model is furnished and compared to symptoms exhibited by an actual faulty physical artifact, debugging software reverses the roles of model and observations. Instead of relying on the user to formally specify the desired program behavior, the (faulty) program is taken as its own model and is compared to examples representing correct and incorrect executions. Hence, the model in MBSD reflects the faults present in the program, while the observations indicate program inputs and correct and incorrect aspects of a program's execution. Observations can either be introduced interactively or can be sourced from existing test suites.

In the following, we briefly outline the model construction. More detailed discussion can be found in [Mayer and Stumptner, 2008]. Similar to SFL, a program is partitioned into components, each representing a particular fragment in the program's source code. The behavior of each component is automatically derived from the effects of individual expressions the component comprises. Connections between components are based on control- and data-dependencies between the program fragments represented by each component.

Assume a model at statement granularity is to be created from the program in Figure 7.2. For each statement s, a separate component is created that is comprised of the expressions and sub-expressions in s. The inputs and outputs of the components correspond to the used and modified variables, respectively. Connections between the components are created to reflect data dependencies between statements in the program (as determined by a simple data flow analysis). Additional variables and components may be introduced to correctly capture data flow at points where control flow paths may split or merge. The component c_7 corresponding to statement 7 in Figure 7.2 is represented as a component with input i_2 and output i_7. Here, i_7 represents the result value of statement 7, and i_2 denotes the previous value of variable i that is implicitly defined at the loop head in line 2.

Similar to classical model-based diagnosis, the model also provides operating modes for each component, where the "correct" (*healthy*) mode h_j of component c_j corresponds to the case where c_j is not to blame for a program's misbehavior. In this case, c_j is defined to function as specified in the program. Conversely, when component j is assumed "not healthy" ($\neg h_j$), c_j may deviate from the program's behavior.

For example, the behavior of c_7 can be expressed as the logical sentence

$$h_7 \Rightarrow i_7 = i_2 + 1. \tag{7.1}$$

In the case where c_7 is considered faulty ($\neg h_7$ is true), the effect on i_7 is left unspecified.

The main difference between the original program and its model is that the model represents the program in a form that is suitable for automated consistency checking and prediction of values in program states in the presence of fault assumptions. This includes program simulation on partially defined program states, using abstract interpretation [Cousot and Cousot, 1977], and backward propagation of values or constraints, which would not occur in a regular (forward) program execution.

```
function FindIndex($tbl, n, k$)
    ▷ Find the index of key $k$ in the hash table $tbl[0,\ldots,n-1]$, or $-1$ if not found.
      Assumes that $tbl$ contains a free slot.
 1      $i \leftarrow$ Hash(k)                ▷ Hash key
 2      while $tbl[i] \neq 0$ do              ▷ Empty slot?
 3          if $tbl[i] = k$ then
 4              return $i$                    ▷ Found match
 5          end if
 6          if $i < n - 1$ then               ▷ At end?
 7              $i \leftarrow i + 1$          ▷ Try next
 8          else
 9              $i \leftarrow 1$              ▷ Wrap around (Fault)
10          end if
11      end while
12      return $-1$                           ▷ Not found
end function
```

Figure 7.2 Algorithm to search in a hash table

Since the resulting model includes the same faults as the program, means to compensate for incorrect structure and behavior of components must be introduced. While heuristics to diagnose structural deficiencies in physical systems can be based on invariants and spatial proximity [Böttcher, 1995], in software, the model must be adapted and restructured once a defect in its structure has become a likely explanation. Here, detection and model adaptation must be guided by using abstract assertions that capture simple "structural invariants" [Mayer and Stumptner, 2008]. Also, since different fault assumptions may alter the control and data flow in a program, models may be created lazily rather than in the initial setup stage.

A trade-off between computational complexity and accuracy can be achieved by selecting different abstractions and models [Mayer and Stumptner, 2008], both in terms of model granularity and representation of program states and executed transitions. In Eq. (7.1) the representation of program state has been left unspecified. Using an interval abstraction to approximate a set of values, sentence Eq. (7.1) becomes a constraint over interval-valued variables i_2 and i_7 [Mayer and Stumptner, 2008]. Another possible abstraction is to encode the operation as logical sentences over the variables' bit representations [Mayer and Stumptner, 2008]. In this chapter, we use the interval abstraction, since it provides good accuracy but avoids the computational complexity of the bit-wise representation. In this chapter, we use a combination of both approaches, where interval abstraction is applied first, followed by bit-wise representation to gain precision but avoid the computational complexity required by the bit-wise model.

Similar to consistency-based diagnosis of physical systems [Reiter, 1987], from discrepancies between the behavior predicted by the model and the behavior anticipated by the user, sets of fault assumptions are isolated that render the model consistent with the observations. Formally, the MBSD framework is based on extensions to Reiter's consistency-based framework, where a diagnosis is a set of faulty components that together explain all observed failures. Diagnoses are obtained by mapping the implicated components into the program's source code [Mayer and Stumptner, 2008].

Diagnosis Let $\mathcal{P}$ denote a program and $\mathcal{T}$ a set of test cases, where each $T \in \mathcal{T}$ is a pair $\langle I, O \rangle$ where I specifies $\mathcal{P}$'s inputs and A is a set of assertions over variables in $\mathcal{P}$ that (partially) specify the correct behavior of $\mathcal{P}$ *with respect to* T. Let $\mathcal{C}$ denote a partition of the statements in $\mathcal{P}$. A *diagnosis* of $\mathcal{P}$ with respect to $\mathcal{T}$ is a set of components $\mathcal{D}$ such that $\forall \langle I, A \rangle \in \mathcal{T}$:

$$\mathcal{P}(I) \wedge \{\neg h_j | c_j \in D\} \wedge \{h_j | c_j \in \mathcal{C} \setminus D\} \not\models \neg O.$$

As an example, suppose the program in Figure 7.2, which contains a defect in line 9 – instead of assigning 0 to variable i, it assigns 1. An observation for this program consists of program inputs, i.e., values for variables *tbl*, n and k, together with the anticipated result value returned by the algorithm. For example, the following assignments

- $tbl \leftarrow [90, 21, 15, 0, 0, 0, 8, 23, 0, 0, 0, 0, 50, 60, 59]$,
- $n \leftarrow 16$, and
- $k \leftarrow 90$

and the assertion $result = 0$ could be an "observation" specifying the inputs and the desired result of a particular program execution. Since the result (-1) obtained by running the program on the given inputs contradicts the anticipated result (0), it has been shown that the program is incorrect (assuming that the test harness is correct).

When MBSD is used with the program in Figure 7.2 and the test case above, a contradiction is detected when the assertion checking the expected result fails. It is derived that the (cardinality-) minimal fault assumptions that are consistent with our test specification are: $\{\neg h_1\}$, $\{\neg h_7\}$, $\{\neg h_9\}$, and $\{\neg h_{12}\}$. Hence, the statements in lines 1, 7, 9 and 12 are considered the possible root causes of the symptoms. Any other statement cannot alone explain the incorrect result, since the result remains incorrect even if a statement is altered.

Internally, MBSD compiles the program into a notation that facilitates symbolic execution of the program. This representation is used to reason over observed failures. To illustrate how MBSD works internally, suppose the simple program in Figure 7.3 (we decided to use this program instead of the working example using thusfar for simplicity). When the program is run with $x = 2$ and $n = 3$, the expected result is 8, but instead a 0 is returned. As an example, and using the test case, MBSD computes whether c_3 is a valid diagnosis candidate by checking if the following formula is consistent

$$\begin{aligned}
&\neg h_3 \wedge \\
&x, n, p \in \{[k, l] \mid x, n \in \mathbb{R}, k \leq l\} \wedge \\
&x = 2 \wedge n = 3 \wedge \\
\neg h_1 \vee\ & p = 1 \wedge \\
(\neg h_2 \vee\ & n > 0 \wedge \\
\neg h_3 \vee\ & p = p \div x \wedge \\
\neg h_4 \vee\ & n = n - 1)^n \wedge \\
&p = 8
\end{aligned}$$

```
function Pow($x, n$)
    ▷ Compute $x^n$
1       $p \leftarrow 1$
2       while $n > 0$ do
3           $p \leftarrow p \div x$                ▷ Fault: used $\div$ instead of $\times$.
4           $n \leftarrow n - 1$
5       end while
        return $p$
end function
```

Figure 7.3 Faulty power function

Because there is a solution for the formula above, c_3 is a valid diagnosis and it is added to the diagnostic report. This is repeated to all components in the program to obtain the list of all possible diagnosis candidates. For this example $D = \{\{1\}, \{3\}, \{5\}\}$.

Conversely to SFL (see Chapter 2), the model-based technique captures the semantics of programming constructs, but does not assign ranking information to candidate explanations. Hence, in this respect the techniques complement each other.

7.1.1 *Issues in MBSD*

While the pure MBSD framework is well-suited to carry out complex inferences, its application in practice is limited due to the following factors:

Result interpretation: If many explanations are returned, MBSD alone provides little information to discriminate between the different explanations. Here, a mechanism to rank results would be desired.

In contrast to electronic circuits, where long sequences of e.g. inverters are uncommon, program executions frequently contain long chains of control- and data dependencies, leading to a number of explanations that cannot be distinguished without further observations. For example, the value of the conditional test in line 2 of the program in Figure 7.2 depends on all statements executed in previous iterations. Interactive measurement selection techniques are difficult to apply, since program states in different executions may be incomparable, rendering entropy-based solutions ineffective.Returning a "super component" as explanation is also not viable in general, since the involved statements can span many different program fragments. Therefore, an approach that works with little or no user involvement is desired.

Scalability: The application of MBSD has been limited to small programs, since the computational effort exceeds what is considered reasonable for interactive scenarios. Hence, inference processes must be applied selectively to remain efficient.

External interfaces: MBSD requires that effects of program fragments can be simulated even if only partial information is available. Programs interacting with external components, such as I/O, files and GUIs, must be modified to either remove these interactions or provide placeholder implementations.

The first two issues can be addressed by introducing a mechanism to estimate, for each component c in the model, how likely it is that c contains a fault. The third issue is common to most program analysis techniques and is beyond the scope of this chapter.

Assuming a suitable measure is available, ranking of results based on fault probability and investigating different explanations in best-first order rather than computing all explanations at once are straightforward. Since a priori probabilities are typically not directly available, other means to determine a suitable likelihood value must be used.

As demonstrated in Chapter 2, SFL is a light-weight technique that analyzes abstraction of program traces to yield a ranking of likely fault locations. To illustrate how SFL works, consider the faulty program in Figure 7.2 and the inputs/outputs used to illustrate the MBSD approach. Executing the program in Figure 7.2 using that observation results in the first row vector in the activity matrix A in Figure 7.4. The vector contains a single 0 entry, indicating that all components but c_4 are executed. Since the returned value does not match the anticipated result, the entry in the error vector is set to 1. Assume that further tests are executed to yield the other 5 rows in the activity matrix.

For each component c_j the Ochiai similarity s_j is given below the matrix. For c_3, the similarity coefficient s_3 is 0.63: as can be seen from the third column in the matrix, there are two failing test runs where c_3 is executed ($n_{11}(3) = 2$), no failing run where c_3 does not participate ($n_{01}(3) = 0$), and three successful executions where c_3 is involved ($n_{10}(3) = 3$). c_6, c_7 and c_9 are considered to be most closely correlated with failing tests and should be examined first. Conversely, c_4 is not considered relevant at all.

Since SFL abstracts a program's behavior into a model that is not suitable for reasoning about the semantics of individual components, results may suffer from the following phenomena:

$$\begin{array}{cccccccc} c_1 & c_2 & c_3 & c_4 & c_6 & c_7 & c_9 & c_{12} \\ \left[\begin{array}{cccccccc} 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 \end{array}\right] \end{array} \begin{array}{c} e \\ \left[\begin{array}{c} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{array}\right] \end{array}$$

$$0.58 \quad 0.58 \quad 0.63 \quad 0.00 \quad 0.71 \quad 0.71 \quad 0.71 \quad 0.58$$

Figure 7.4 Activity Matrix

- If a fault lies in a component that participates in all runs (for example, an initialization component), the component is likely not to be ranked in the first places;
- SFL cannot distinguish between components that exhibit identical execution patterns (such as components c_6, c_7, and c_9);
- Nested components executed after the fault is hit are likely to outrank the faulty component. For example, branches of a conditional statement are likely to outrank those components preceding it.
- Many components may be included in the ranking. In our example, seven out of eight components are included in the report.

The aim of this chapter is to show that correlation between the execution patterns of statements with correct and failed executions can significantly improve diagnosis results. The following section outlines our approach to assessing the similarity between different program executions and test outcomes. Since MBSD does not usually exploit correct program executions in any way, this approach can contribute valuable information to guide the model-based framework.

7.2 DEPUTO

In this section we describe how we combine the spectrum-based and the model-based approaches described earlier, in order to capture the best characteristics of both techniques.

Algorithm 4 outlines our combined approach. The algorithm executes in three stages, with the similarity-based approach used in the setup stage (steps 1 to 6), feeding into the subsequent model-based filtering stage (steps 7 to 16), followed by an optional best-first search stage (lines 17 to 24). This combination has significantly lower resource requirements than applying MBSD on the whole program and using SFL only to rank results as proposed in [Mayer et al., 2008]. We start by partitioning the program $\mathcal{P}$ into a set of components $\mathcal{C}$ and execute $\mathcal{P}$ on the available test cases $\mathcal{T}$ to obtain the activity matrix $\mathcal{M}$.

Algorithm 4 DEPUTO Algorithm

```
Input: Program $\mathcal{P}$, set of test cases $\mathcal{T}$
Output: Fault assumptions explaining failed test runs
 1 $\mathcal{C} \leftarrow$ CREATECOMPONENTS($\mathcal{P}$)
 2 $\mathcal{M} \leftarrow$ GETCOMPONENTMATRIX($\mathcal{C}, \mathcal{P}, \mathcal{T}$)
 3 $\langle \mathcal{T}_P, \mathcal{T}_F \rangle \leftarrow$ PARTITION($\mathcal{M}, \mathcal{T}$)
 4 $\mathcal{R} \leftarrow$ SFL($\mathcal{M}$)                    ▷ Apply SFL
 5 $\mathcal{S} \leftarrow \emptyset$                          ▷ Skipped components
 6 $I \leftarrow \emptyset$                          ▷ Inspected components
 7 repeat
 8     $\widehat{\mathcal{C}} \leftarrow$ RANKING_POP($\mathcal{R}$)
 9     $D \leftarrow$ MBSD($\widehat{\mathcal{C}}, \mathcal{T}_F$)        ▷ Apply MBSD
10     $I \leftarrow I \cup D$
11     if $D_{bug} \in D$ is confirmed faulty then
12         return $D_{bug}$
13     else
14         $\mathcal{S} \leftarrow \mathcal{S} \cup (\widehat{\mathcal{C}} \setminus D)$
15     end if
16 until $\mathcal{R} = \emptyset$
17 while $\mathcal{S} \neq \emptyset$ do
18     $\widehat{\mathcal{C}} \leftarrow$ PDG_RANKING_POP($\mathcal{S}, I$)
19     $I \leftarrow I \cup \widehat{\mathcal{C}}$
20     if $c_{bug} \in \widehat{\mathcal{C}}$ is confirmed faulty then
21         return $\{\neg h_{bug}\}$
22     end if
23 end while
24 return $\emptyset$                          ▷ No explanation found
```

Using $\mathcal{M}$, we partition $\mathcal{T}$ into passing tests ($\mathcal{T}_P$) and failing ones ($\mathcal{T}_F$). From $\mathcal{M}$, a sorted list of components $\mathcal{R}$ in order of likelihood to be at fault is obtained as described in Chapter 2 (line 4).

In the subsequent loop, MBSD (line 9) is used to eliminate the top-ranked candidate explanations that are not considered valid explanations by the model-based approach. Instead of applying MBSD once to compute *all* explanations and present the ranked candidates to the user, an incremental strategy allows for early termination once a fault has been identified. First, the set of components $\widehat{\mathcal{C}}$ with the highest similarity coefficient in $\mathcal{R}$ are obtained using the RANKING_POP($\mathcal{R}$) function, which also removes from $\mathcal{R}$ all elements in $\widehat{\mathcal{C}}$. Second, function MBSD($\widehat{\mathcal{C}}, \mathcal{T}_F$) returns a set of candidate explanations $D \subseteq \widehat{\mathcal{C}}$ that explain observed failures $\mathcal{T}_F$. Finally, if the fault is in the returned set, the algorithm stops; otherwise none of the candidates represent valid explanations and other must be generated. The algorithm stops once no more explanations could be found or if none of the remaining components was executed for a failing test. $\mathcal{S}$ is the set of components that are implicated by SFL but not by MBSD.

If no explanation is found after all components implicated by MBSD have been explored, we employ a best-first search procedure that traverses the program along dependencies between components with decreasing fault similarity. No explanation may be found if the fault affects component interdependencies such that the fault assumptions and model abstraction can no longer represent the fault. In line 18, the set of components with maximum fault similarity that are connected to the previously explored components is returned. Function PDG_RANKING_POP$(\mathcal{S}, I)$ returns the set of components in S with highest similarity that are directly connected to the previously inspected set of component I. If the component is confirmed to be (part of) a valid explanation, the search stops and the diagnosis is returned. Note that the explanation may only cover part of the true fault. Line 24 in Algorithm 4 can only be reached if the faulty program fragment is not covered by any component, or if the user oracle that decides whether an explanation is indeed a satisfying explanation is imperfect and may miss a fault.

Applying Algorithm 4 using the test suite from the example in Section 7.1, $\{\neg h_7\}$ and $\{\neg h_9\}$ are obtained as candidate explanations. Both candidates are associated with the highest similarity coefficient 0.71.

Notably, this result improves upon both individual fault localization procedures. Different from pure SFL, $\{\neg h_6\}$ is no longer considered an explanation. Conversely, candidates $\{\neg h_1\}$ and $\{\neg h_{12}\}$ obtained using pure MBSD are low-ranking in SFL and hence omitted at this stage. ($\{\neg h_{12}\}$ is already eliminated by pure MBSD when using the second failing test case in the example.)

Without further information, neither approach can discriminate between the two remaining candidate explanations. Since it is assumed that the user acts as oracle that can reliably recognize true faults, the algorithm stops in the first iteration (in line 12), once the statement in Figure 7.2 corresponding to $\{\neg h_9\}$ has been confirmed to be incorrect.

7.3 EMPIRICAL EVALUATION

To gain a better understanding of the combined approach, in this section, we empirically evaluate its efficiency. First, we introduce the program under analysis and the evaluation metric.

7.3.1 *Experimental Setup*

Program under analysis In our study we use the *TCAS* program as taken from the *Siemens Test Suite*[Do et al., 2005]. *TCAS* simulates the resolution-advisory component of a collision avoidance system similar to those found in commercial aircraft. It consists of 138 lines of C code and takes twelve parameters as input; the numeric result value encodes one out of three possible resolution advisories. The program comes with 1608 test cases and 41 different variants

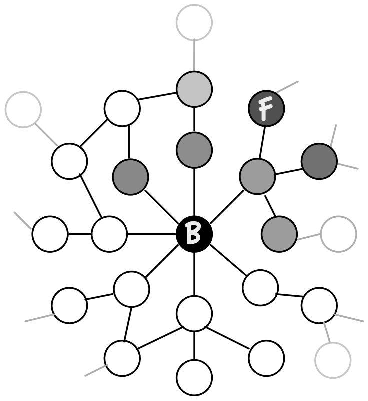


Figure 7.5 SCORE: Traversing the PDG

with known faults. For each variant, an average of forty test cases reveal a fault. In our experiments, all available test cases were used.

Evaluation Metric In the fault diagnosis research community rank- [Abreu et al., 2007, Jones and Harrold, 2005, Wong et al., 2008] and dependency-based [Liu et al., 2005, Renieris and Reiss, 2003] metrics have often been used. The former quantify the quality of a result based on the ranking position of the faulty component relative to all components, and is mainly used with techniques that rank components in a program. In contrast, dependency-based measures typically operate on the program dependence graph (PDG) and are mainly applied to evaluate techniques that either do not rank components (for example MBSD) or do not rank all components of a program (such as SOBER [Liu et al., 2005]). Essentially, starting with the set of blamed components, dependencies between components are traversed in breadth-first order until the fault has been reached. The quality of a diagnostic report is measured as the fraction of the PDG that is traversed. Both metrics quantify the percentage of a program that needs to be inspected in order to find the fault. We refer to them as SCORE.

To assess the accuracy of DEPUTO, we use both metrics. First, if a fault is found in the refining phase (lines 7 to 16 in algorithm 4), the SCORE is given by

$$\text{SCORE} = \frac{|I|}{M} \cdot 100\%,$$

where $|I|$ denotes the number of inspected components. However, if no fault is found in this phase then we use the PDG-based metric by traversing the ranking starting with the previous inspected set of components I (lines 17 to 23). Figure 7.5 depicts how the PDG is used to compute the SCORE: consider B (for blamed) to be the only component in I, SCORE is computed by computing the number of nodes that we need to inspect to reach node F (for faulty).

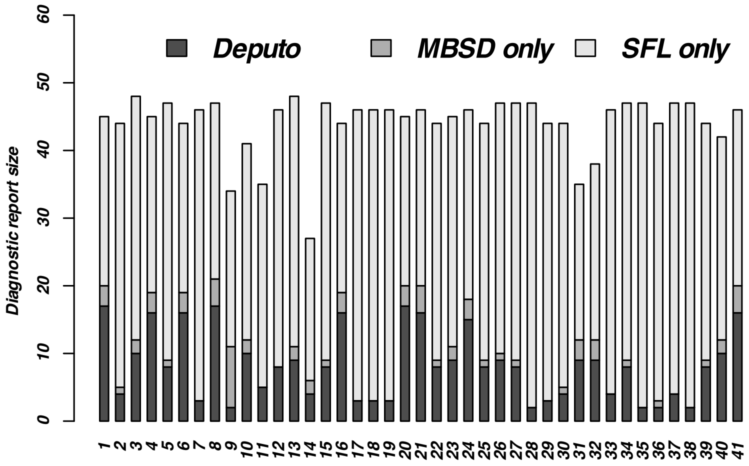


Figure 7.6 Components implicated by SFL, MBSD, and Deputo for the 41 variants of *TCAS*

7.3.2 *Experimental Results*

Results for the individual approaches have already been published elsewhere. MBSD using an interval abstraction requires 11 statements to be inspected on average [Mayer and Stumptner, 2008]. The median Score is 13% (14% on average) for *TCAS*. Note that when MBSD fails to implicate the faulty component, more statements than just those in the diagnostic report must be inspected. The results obtained with SFL are discussed in Chapter 2. Following the generated ranking would lead to the fault after inspecting 20 statements on average, resulting in a median (and average) Score of 14%.

Deputo combines SFL with MBSD to refine the ranking of implicated components. To understand how well MBSD filters statements from the ranking, we first study the number of implicated components. Figure 7.6 contrasts the components implicated by either approach with those blamed by both. It can be seen that MBSD significantly reduces the number of components when compared with SFL. Furthermore, neither approach subsumes the other. Restricting the debugging process to those statements that are implicated by both approaches, the average number of statements reduces from 36 (20 if considering only until the fault is hit) to 8. Hence, the total number of relevant statements reduces considerably. Note, however, that Figure 7.6 does not imply that SFL's contribution is negligible; although it implicates more components, it also builds a ranking that more quickly leads to the fault.

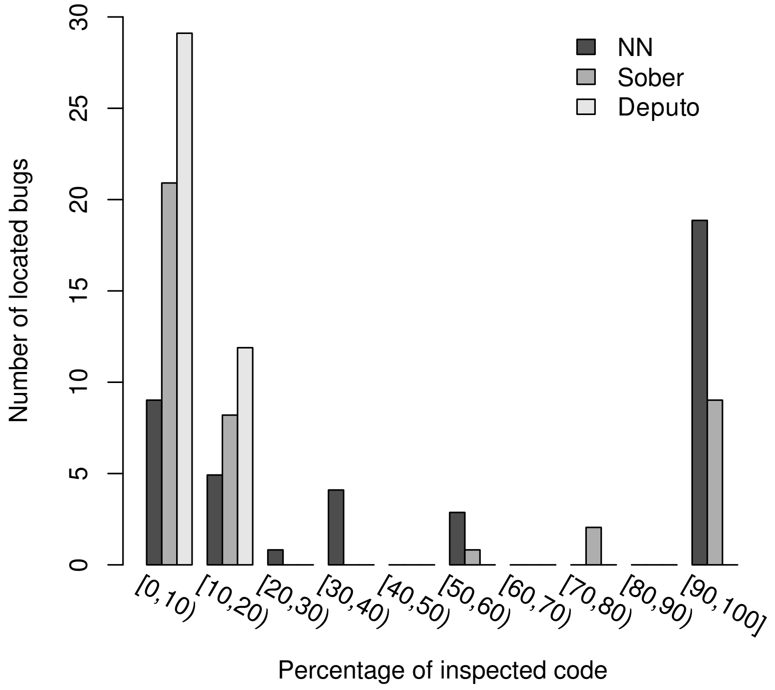


Figure 7.7 Debugging efficiency

Similar improvements can be observed in the ranking of components. Using the pure SFL approach one hits the true fault after inspecting twenty statements on average, but many unrelated statements must be examined. When using MBSD as filtering mechanism, the true fault is located after *seven* statements on average. Hence, the model-based filtering mechanism seems well-suited to discard irrelevant components from the SFL fault profiles.

The improved accuracy of the combined approach also reflects in much improved quality indicators. It is observed that the combined approach largely outperforms the individual techniques. In some cases, SFL outperforms the combined approach, suggesting that the model used in MBSD may not be able to accurately reflect the fault; so far, we have not been able to devise heuristics that can consistently predict such discrepancy from the a-priori component probabilities and diagnoses to further improve accuracy. Overall, the fraction of the program that must be inspected reduces from 13% and 30% for SFL and MBSD, respectively, to 8%. Although MBSD alone is not able to locate faults for 9 of the 41 programs (due to limitations on faults in global vari-

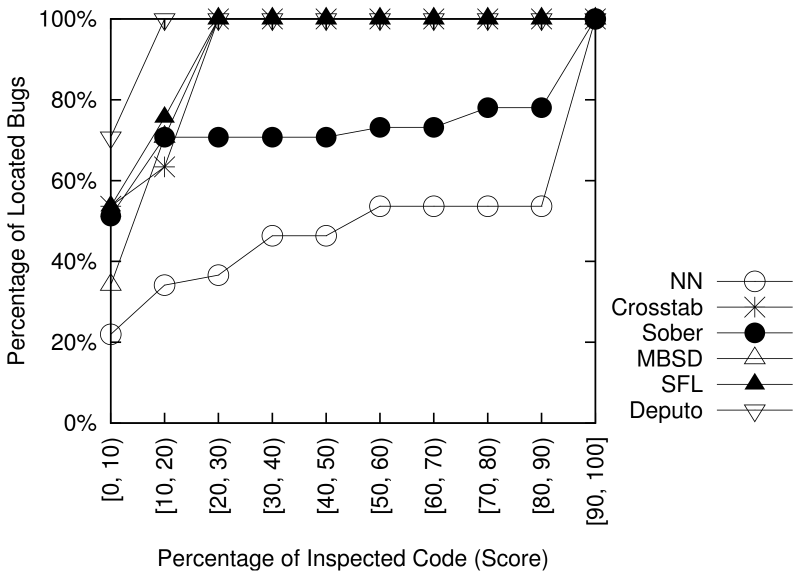


Figure 7.8 Cummulative Debugging efficiency

able initialization in our current implementation), the overall performance of the combined approach does not seem to be adversely affected in most cases. This can be explained by two observations: (i) the number of diagnoses that are implicated in those cases is small (4 on average), and (ii) the suspect program fragments are close to the actual faults when navigating the program structure.

Since the diagnostic report obtained from Deputo is a ranked list of likely faulty components, its size alone is not a good indicator for its quality. Instead, we employ the Score metric as defined in the previous section to evaluate our results. Figures 7.7 and 7.8 visualize the percentage of located bugs and cumulative percentage of located bugs, respectively, for different fractions of inspected code. Our approach outperforms the individual approaches as well as the simple statistics-based fault localization technique proposed in [Renieris and Reiss, 2003], where different combinations of union and intersection of "similar" passing and failing test runs are computed. This can be attributed to the improved ranking mechanisms built into our algorithm that is more robust with respect to overlapping passing and failing executions. Our combined approach also improves on SOBER [Liu et al., 2005] and CROSSTAB [Wong et al., 2008], which are statistical approaches based on hypothesis testing that have been shown to dominate other recent bug detectors. For instance, if up to 10% of the program would have been inspected, Deputo would locate 71% of the faults, whereas SOBER and CROSSTAB would yield only 51% and 53%, respectively.

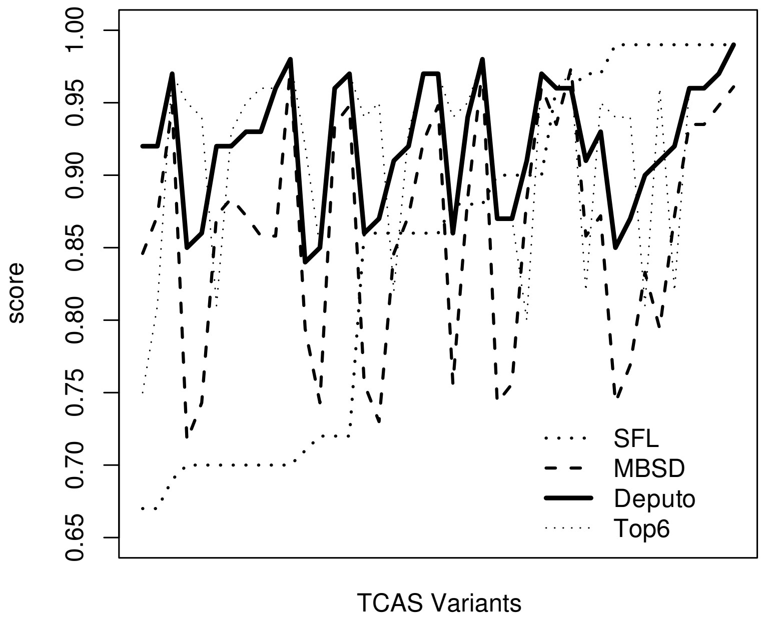


Figure 7.9 Individual report quality

We also evaluated a modified version of Algorithm 4, where the MBSD section is stopped after the six[2] most highly ranked components have been explored; the remaining components were subsequently explored using the best-first part of our algorithm. The resulting quality indicators are labeled *Top6* in Figure 7.9. The results indicate that the components implicated by the combined approach sometimes narrowly miss the true faults; in these cases, the score measure improves compared to the combined approach. In other cases, following the original algorithm is more successful. Overall, the quality indicators do not differ significantly between the two models. Investigating whether heuristics can be developed that choose a cutoff to improve accuracy remains for future work.

Δ-slicing and explain [Groce et al., 2006] are two techniques for fault localization that exploit differences between passing and failing abstract program executions traces found by a model checker. Table 7.1 compares our

[2]This cutoff seemed to have the best overall effect for an extended test suite used in [Liu et al., 2005].

	explain	Δ-slicing	Deputo
v1	49	9	7
v11	64	7	7
v31	24	7	7
v40	25	–	16
v41	32	12	6

Table 7.1 Individual Scores

results to the published individual results for all five versions of *TCAS* reported in [Groce et al., 2006]. We conclude that Deputo is far superior to explain (which requires to explore 24–64% of a program) and performs competitive with respect to Delta slicing (within 5%), yet at reduced complexity. However, to understand whether our approach is indeed much better than the ones presented in [Groce et al., 2006] more experimentation is needed.

Our combined framework also reduces the time required by MBSD from 185*s* [Mayer and Stumptner, 2008] to 73.2*s* on average, representing an average speed-up of 2.5 times. The time required by the SFL part of our algorithm is negligible.

7.4 RELATED WORK

The most similar work to the one presented in this chapter is described in [Mayer et al., 2008] where a statistics-based approach is used to rank the set of candidates given by MBSD. The authors conclude that the combination of the two approaches reduces the effort to track down faulty components. However the computational complexity of this approach is determined by MBSD. As a result of its high computational complexity, the approach proves prohibitive for large programs. While the underlying principle is similar to [Mayer et al., 2008], we achieved an 2.5 times (on average) speed-up while maintaining the accuracy of the previous approach.

In model-based reasoning, the program model is typically generated from the source code, as opposed to the traditional application of model based diagnosis where the model is obtained from a formal specification of the (physical) system [Reiter, 1987]. In [Mayer and Stumptner, 2008] an overview of different MBSD techniques concludes that the Abstract Interpretation model used in this work leads to good results while not suffering from the computational complexity inherent to more precise analysis techniques [Mayer and Stumptner, 2008]. Recently, model-based techniques have also been proposed to isolate specific faults stemming from incorrect implementation of high-level conceptual models [Yilmaz and Williams, 2007], where mutations are applied to state machine models to detect conceptual errors (see Figure 7.10), such as incorrect control flow and missing or additional features found in the implementation. Other approaches that fit into this category include `explain` [Groce et al., 2006] and Δ-slicing [Groce et al., 2006], which are based on comparing

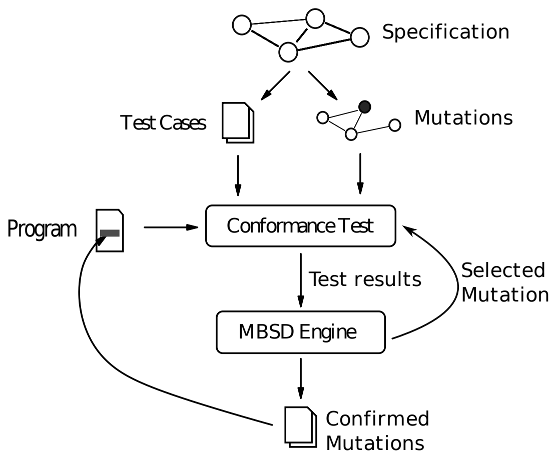


Figure 7.10 Using Conceptual Models to Enhance MBSD

execution traces of correct and failed runs using model checkers. Model-based test generation [Esser and Struss, 2007] from abstract specifications of systems employs a similar idea where possible faults manifested as differences in abstract state machines are analyzed to generate tests. Our work differs in that we are concerned with program representations that more closely reflect the actual program artifact to locate faults at a more detailed level [Mayer and Stumptner, 2007a].

Machine learning techniques have been applied to programs [Xie and Engler, 2003] and their executions [Nimmer and Ernst, 2001] to infer likely invariants that must hold at particular locations in a program. Violations can subsequently be used to detect potential errors. Model-based approaches have been shown to provide more reliable behavior than [Nimmer and Ernst, 2001], since success of the trace analysis depends much on the test runs and type of invariants to be inferred [Köb and Wotawa, 2005]. The static program analysis approach requires that similar patterns appear repeatedly in a program, but is not applicable when common patterns are not easily identified.

Combining program execution and symbolic evaluation has been proposed to infer possible errors [Engler and Dunbar, 2007]. Similar to MBSD, a symbolic, under-constrained representation of a program execution and memory structures are built. Instead of using fault probabilities to guide diagnosis, only those candidate explanations that definitively imply a test failure are flagged.

7.5 SUMMARY

We have shown that the accuracy of spectrum-based fault localization increases significantly when combined with approaches that make use of a model to yield valid explanations for observed failures. Our unique combination of semantics-based analysis as undertaken in model-based software debugging and dynamic aspects obtained from program execution spectra has proved to greatly focus debugging efforts to relevant parts of a program. Overall, a reduction of suspect program fragments to less than 8% of the complete program has been achieved on our test suite, outperforming both individual techniques and most other state of the art techniques. Furthermore, an average speed-up of 2.5 compared to the model-based approach has been observed. We have further shown that our approach is among the state of the art automated debugging tools.

Conclusions

8

"All truths are easy to understand once they are discovered; the point is to discover them."

– Galileo Galilei

Automatic fault localization techniques aid developers/testers to pinpoint the root cause of software faults, thereby reducing the debugging effort. Depending on the amount of knowledge that is required about the system's internal component structure and behavior, current, predominant approaches to automatic software fault localization can be classified as (1) statistics-based approaches, and (2) reasoning approaches. Statistics-based fault localization techniques such as SFL use program spectra to find a statistical relationship with observed failures. While modeling costs and computational complexity are minimal, SFL's diagnostic accuracy is inherently limited as no reasoning is used. In contrast to SFL, model-based reasoning approaches use prior knowledge of the system, such as component interconnection and statement semantics, to build a model of the correct behavior of the system. While delivering higher diagnostic accuracy, they suffer from high computation complexity.

In this thesis, we endeavored to capture the best of both worlds in one single approach by investigating a spectrum-based reasoning approach to fault localization. By abstracting from program topology and dependencies, the spectrum-based modeling approach, in conjunction with a low-cost heuristic minimal hitting set algorithm, allows the reasoning to be relatively cheap, while accuracy is increased compared to SFL. In particular, we investigated (1) the inherent performance limitation of SFL, and (2) the benefits of a reasoning approach to spectrum-based fault localization.

In addition, we also conducted the following studies. First, aimed at total automation of the fault localization process, we studied the capabilities of simple, generic program invariants to replace test oracles. Second, we investigated the possibility of using an SFL-based heuristic to focus the computation of valid diagnosis candidates, rendering our reasoning approach amenable to large programs. Finally, we studied whether SFL can be integrated with existing model-based software debugging approaches to reduce their high time complexity, while improving their diagnostic quality.

8.1 SUMMARY OF CONTRIBUTIONS

The main contributions of this thesis are twofold:

- From our study on SFL we found that near-maximal diagnostic accuracy is already obtained for low-quality error observations and limited numbers of test cases (approximately 20% of the Siemens programs needs to be inspected, on average, to find the fault). In particular, we found that a new similarity coefficient, known from the molecular biology community, the Ochiai similarity coefficient, consistently outperforms all coefficients investigated, independent of the experimental environment.
- We present a novel, low-cost, Bayesian reasoning approach to spectrum-based multiple fault localization, coined Barinel. A central feature of our contribution is the use of a generic, intermittent component failure model. Whereas previous approaches have used approximations instead, in Barinel component intermittency rate is computed as part of the posterior candidate probability computation, using a maximum likelihood estimation procedure. This procedure optimally exploits all information contained by the program spectra. Our synthetic and real software experiments show that Barinel outperforms SFL. Furthermore, we have both empirically and theoretically established that Barinel's diagnostic accuracy is optimal in the context of single-fault programs. In perspective, our results also show that the diagnostic accuracy of the previous, approximation-based approaches is only marginally worse, while allowing an attractive, low-cost, incremental computation scheme similar to coefficient computation in SFL.

Furthermore, this thesis also makes the following five additional contributions. First, as SFL is amenable to resource-constrained software systems, such as embedded software, we applied SFL to industrial (real-time) embedded television software. From our experiments, we conclude that not only SFL is well-suited to resource-constrained (embedded) systems, but SFL also has the potential to reduce debugging effort from several weeks to a couple of hours.

Second, we showed that simple fault screeners have the capabilities to replace test oracles with respect to the diagnostic performance of SFL at limited overhead. In particular, our empirical experiments suggest that SFL is robust to false positives and negatives, which are a fact of life when using generic fault screeners as error detectors. Our analytical screeners' performance model confirms the empirical experiments in that the training effort required by near-"ideal" screeners increases with the variable domain size, whereas simple screeners, such as range screeners, only require limited (constant) training effort. From preliminary experiments, we conclude that not all program variables need to be screened to yield comparable diagnostic results as when all program variables are screened. This result paves the way for automatic fault localization at modest run-time cost.

Third, we present an SFL-based heuristic algorithm, coined Staccato, to focus the search of minimal hitting sets, decreasing the time complexity by orders of magnitude while capturing all important diagnosis candidates. Staccato is extremely important as Barinel critically depends on low-cost minimal hitting set algorithms to be amenable to large software systems. For the programs considered, only $O(100)$ candidates need to be generated to capture all candidates needed to find the actual faults.

Fourth, we combined SFL with model-based software debugging to render the latter amenable to larger programs, as well as to introduce ranking of its diagnostic set. We showed that the combination of semantics-based analysis as performed in model-based diagnosis and the dynamic aspects obtained from program execution spectra yields better diagnostic reports. In our experiments, an average reduction of suspect program fragments to less than 8% of the complete program has been achieved, outperforming both individual techniques and most other state-of-the-art techniques. We have also shown that our combined approach yields an average speed-up of 2.5 compared to the model-based approach.

Finally, on a more practical note, we released a toolset providing most of the algorithms proposed in this thesis (`http://www.fdir.org/zoltar`) [Janssen et al., 2009]. The toolset is composed of several instances of SFL (i.e., several similarity coefficients), Barinel, and Staccato. Error detection input can either be manually given by the developer, provided using test oracles, or using the output from fault screeners.

8.2 RECOMMENDATIONS FOR FUTURE WORK

Inspired by the contributions made in this thesis, a multitude of interesting open issues are worth investigating. In the following we suggest several recommendations for future work:

- We have mainly investigated the fundamental limitations of SFL empirically. Although such experiments are valuable to understand how SFL works in detail, creating an analytic performance model to predict the impact of several parameters on W would give us a complete overview of SFL's limitations/trends. The model could potentially aid, e.g., in creating an optimal similarity coefficient. Another application would be to prove that the optimal grain size for componentization is the basic block size, something that is intuitive but not based on a formal proof.

- Aimed at total automation of the detection-diagnose process, we investigated the capabilities of generic program invariants to replace test oracles at the operational phase. More time-consuming and program-specific screeners, such as relationship invariants between variables, or components' state machine-based program invariants [Lorenzoli et al., 2008], may lead to better diagnostic performance, and are therefore worth investigating. In addition, given the fact that only a limited number of (so-called collar) variables are primarily responsible for program behavior [Menzies et al., 2007, Pattabiraman et al., 2005], a study of the impact of (judiciously) reducing the amount of screened program points is required to minimize overhead.

- As for Barinel, recommendation for future work includes (1) extending the activity matrix from binary to integer, allowing us to also exploit component involvement frequency (e.g., program loops), (2) reducing the cost of gradient ascent by implementing quadratic convergence techniques.

- The SFL-based heuristic function used by the Staccato algorithm is specifically tailored to fault localization problems, as it visits solutions in best-first order (aiming to capture the maximum posterior probability mass in the shortest amount of time). Staccato is interesting due to its low complexity, and may be well-suited to other application domains, such as test case minimization and prioritization [Harrold et al., 1993, Tallam and Gupta, 2005, Li et al., 2007, Smith and Kapfhammer, 2009, Santelices et al., 2008]. Given the different domain, a prerequisite is to devise a prospect heuristic focusing function, e.g., based on information gain, which efficiently emits test cases in best-first order. Such heuristic function could be plugged into Staccato to perform test case minimization and prioritization for, e.g., sequential diagnosis.

- With regard to model-based software debugging, connecting the lower-level models that reflect most details of a program to high-level conceptual models to detect a more diverse set of faults seems promising to broaden the scope of applicability. Moreover, as the number of possible multiple-fault diagnosis candidates is exponential in the number of components, model-based software debugging has only been applied in the context of single faults. Integrating Staccato within the model-based software debugging framework could potentially allow it to be applied in the context of multiple faults, as well as making it amenable to large software systems.

- Our current diagnostic performance metric W assumes a debugging context, and does not take into account that the cost of operating on a wrong diagnosis can be quite different from offline component testing, such as in situations where a critical fault necessitates a high-cost shutdown. In particular, a cost-reward metric to judge the utility of the diagnosis candidates should be investigated, which would be used by the system to either ignore the candidate or, e.g., start a recovery process. An interesting issue is to compare this utility metric with currently used metrics, such as (wasted) debugging effort W.

- Thusfar, SFL has been applied exclusively in the software engineering domain. Although the hardware domain is much more amenable to a model-based diagnosis approach, even in this domain (except for, e.g., boolean logic) modeling can be an inhibiting factor. Despite the inherently superior precision of model-based diagnosis, SFL's greater ability to handle large time series of observation data can partly compensate for its inherently limited precision. Furthermore, further investigation is required to ascertain whether SFL's spectral information input can be enriched with additional information (conflict sets [de Kleer and Williams, 1987]) up to the point where SFL's precision starts approaching that of model-based diagnosis, yet at the attractive, low time complexity of the former method. Consequently, the performance loss may well be outweighed by the extent of automation that can be achieved.

- Last but not least, although the industrial case studies presented on Chapter 3 already indicate the added value of using automatic fault localization approaches, a more elaborate field test would have to be performed to assess whether automatic fault localization techniques have the real potential to reduce software debugging efforts.

Appendices

Bloom Filter Hash Functions

A

This appendix gives further information on the two hash functions used on the implementation of the Bloom filter screener described on Section 4. The two hash functions used are the 32 bit Mix Functions by Thomas Wong (see Figure A.1) and the 32 bit integer hash function and Robert Jekins (see Figure A.2).

```
function 32Mix($a$)
    ▷ Compute hash value for a.
    $a = \neg a + (a * 2^{15})$
    $a = a \wedge (a * 2^{12 \wedge (32-1)})$
    $a = a + (a * 2^{2})$
    $a = a \wedge (a * 2^{4 \wedge (32-1)})$
    $a = a * 2057$
    $a = a \wedge (a * 2^{16 \wedge (32-1)})$
    return a
end function
```

Figure A.1 32-bit Mix Function

```
function 32Int($a$)
    ▷ Compute hash value for a.
    $a = (a + 0x7ed55d16) + (a * 2^{12})$
    $a = (a \wedge 0xc761c23c) \wedge (a \div 2^{19})$
    $a = (a + 0x165667b1) + (a * 2^{5})$
    $a = (a + 0xd3a2646c) \wedge (a * 2^{9})$
    $a = (a + 0xfd7046c5) + (a * 2^{3})$
    $a = (a \wedge 0xb55a4f09) \wedge (a \div 2^{16})$
    return a
end function
```

Figure A.2 32-bit Integer Hash Function

Gradient Ascent Procedure

B

The Barinel algorithm, described in Chapter 5, computes the individual health probabilities h_j of faulty components c_j using a maximum likelihood estimation method. Underlying this method is a gradient ascent procedure for the computation of the individual h_j. This appendix outlines the computational aspects of the Barinel algorithm for $C = 2$ for ease of exposition. However, the explanation as well as the approach easily generalizes for $C > 2$.

As an example, consider the following activity matrix and error vector (A, e)

c_1	c_2	e
1	0	0
1	1	0
0	1	1
1	0	1

Let the minimal hitting set algorithm (e.g., Staccato) yield the set of diagnosis D comprising the valid candidates $\{d_1, \ldots, d_k, \ldots, d_K\}$, where $K = |D|$ is the number of diagnosis candidates in D. For example, $d = (1, 1, 0, \ldots, 0)$ means that component 1 and 2 are considered to be faulty. For simplicity, d is also represented only in terms of the indices of faulty components only. Hence, $d = (1, 1, 0, \ldots, 0)$ is mapped into $\{1, 2\}$. In the following, $d^{(j)}$ indexes bit j of d, where $j \leq M$. The minimal hitting set for the (A, e) example given above is the following list containing one diagnosis candidates $D = \{(1, 1)\}$.

For each $d \in D$, the probability diagnosis candidate d is correct $\Pr(d)$ is defined. Compiled in order of the rows of A and e, it follows

$$\Pr((1,1)) \quad = \quad (1 - h_1) \cdot (1 - h_1 \cdot h_2) \cdot h_2 \cdot h_1$$

Generalized per row i we obtain

$$\Pr(d, i) = h_1^{(d^{(1)} \cdot a_{i,1})} \cdot \ldots \cdot h_M^{(d^{(M)} \cdot a_{i,M})}$$

for $e_i = 0$, or

$$\Pr(d, i) = 1 - h_1^{(d^{(1)} \cdot a_{i,1})} \cdot \ldots \cdot h_M^{(d^{(M)} \cdot a_{i,M})}$$

for $e_i = 1$.

Returning back to the example, it follows

$$\begin{aligned}
\Pr((1,1),1) &= 1 - h_1^{(1 \cdot 1)} \cdot h_2^{(1 \cdot 0)} = 1 - h_1 \\
\Pr((1,1),2) &= 1 - h_1^{(1 \cdot 1)} \cdot h_2^{(1 \cdot 1)} = 1 - h_1 \cdot h_2 \\
\Pr((1,1),3) &= h_1^{(1 \cdot 0)} \cdot h_2^{(1 \cdot 1)} = h_2 \\
\Pr((1,1),4) &= h_1^{(1 \cdot 1)} \cdot h_2^{(1 \cdot 0)} = h_1
\end{aligned}$$

(note that there are no more h_j involved than h_1,h_2, i.e., only faulty components in d contribute to the computation of the probability).

In general, the expression for $\Pr(d)$ has a complex form with many terms. For instance, for $C = 2$ the $\Pr(d)$ expression is as follows

$$h_1^{n_{10}}, h_2^{n_{01}}, (h_1 \cdot h_2)^{n_{11}}, (1-h_1)^{n'_{10}}, (1-h_2)^{n'_{01}}, (1-h_1 \cdot h_2)^{n'_{11}}$$

where n, n' are counters for passed and failed runs, respectively, coded as bit strings that encode the involvement of individual components (like d), accumulated while scanning the rows like above. As an example, n_{10} counts the number of times a passed run was observed while only component one was involved, whereas n'_{10} counts the number of times a failed run was observed while only component one was involved. Note that for simplicity the running example only contains two components ($M = 2$), but the definitions for n and n' generalize for larger systems. In the above example for candidate $(1,1)$ it follows

$$\begin{aligned} n_{10} &= 1 \\ n_{01} &= 1 \\ n_{11} &= 0 \\ n'_{10} &= 1 \\ n'_{01} &= 0 \\ n'_{11} &= 1 \end{aligned}$$

The key idea underlying Barinel is that for each candidate d we compute the h_j for the candidate's faulty components that maximizes the probability $Pr(e|d)$ of the outcome e occurring, conditioned on that candidate d (maximum likelihood estimation for naive Bayes classifier d). In order to compute those h_j, a gradient ascent procedure is used, which is a general framework for solving optimization problems where we want to maximize functions of continuous (differentiable) parameters. Essentially, starting at a point $h_j^{(0)}$, this method takes the form of iterating until a fixed point is reached, i.e., maximum point of $\Pr(e|d)$ (for detailed information on this procedure, refer to [Avriel, 2003]):

$$\begin{aligned} h_1 &= h_1^{(0)} + \nabla \Pr(e|d) \\ &\cdot \\ &\cdot \\ h_M &= h_M^{(0)} + \nabla \Pr(e|d) \end{aligned}$$

Note, however, that, in practice, only the healthy variables for non-faulty components in d need to be calculated. We consider $\log \Pr(e|d)$ instead of $\Pr(e|d)$, as maximizing $\log \Pr(e|d)$ gives us the same h_j and computing the gradient ascent ∇, i.e., $\frac{\partial(\log \Pr(e|d))}{\partial h_j}$ for each faulty component j in d, from the log is much more convenient.

$\frac{\partial(\log \Pr(e|d))}{\partial h_1}$ has the form (for $C = 2$, per term):

$$(\frac{1}{h_1})^{\mathrm{n}_{10}} \cdot \mathrm{n}_{10} \cdot h_1^{(\mathrm{n}_{10}-1)} \cdot 1 + 0 + (\frac{1}{h_1 \cdot h_2})^{\mathrm{n}_{11}} \cdot \mathrm{n}_{11} \cdot (h_1 \cdot h_2)^{(\mathrm{n}_{11}-1)} \cdot h_2 +$$
$$(\frac{1}{1-g1})^{\mathrm{n}'_{10}} \cdot \mathrm{n}'_{10}(1-h_1)^{(\mathrm{n}'_{10}-1)} \cdot (-1) + 0 +$$
$$(\frac{1}{1-h_1 \cdot h_2})^{\mathrm{n}'_{11}} \cdot \mathrm{n}'_{11}(1-h_1 \cdot h_2)^{(\mathrm{n}'_{11}-1)} \cdot (-h_2)$$

which reduces to

$$\begin{aligned} & \frac{\mathrm{n}_{10}}{h_1} + 0 + \frac{\mathrm{n}_{11}}{h_1} - \frac{\mathrm{n}'_{10}}{(1-h_1)} - 0 - \mathrm{n}'_{11}\frac{h_2}{(1-h_1 \cdot h_2)} \\ = \quad & \frac{\mathrm{n}_{10}}{h_1} + \frac{\mathrm{n}_{11}}{h_1} - \frac{\mathrm{n}'_{10}}{(1-h_1)} - \mathrm{n}'_{11}\frac{h_2}{(1-h_1 \cdot h_2)} \end{aligned}$$

Similarly, $\frac{\partial(\log \Pr(e|d))}{\partial h_2}$ equals

$$\begin{aligned} & 0 + \frac{\mathrm{n}_{01}}{h_2} + \frac{\mathrm{n}_{11}}{h_2} - 0 - \frac{\mathrm{n}'_{01}}{(1-h_2)} - \mathrm{n}'_{11}\frac{h_1}{(1-h_1 \cdot h_2)} \\ = \quad & \frac{\mathrm{n}_{01}}{h_2} + \frac{\mathrm{n}_{11}}{h_2} - \frac{\mathrm{n}'_{01}}{(1-h_2)} - \mathrm{n}'_{11}\frac{h_1}{(1-h_1 \cdot h_2)} \end{aligned}$$

Consequently $\frac{\partial(\log \Pr(e|d))}{\partial h_j}$ contains all terms which involve h_j. It follows

$$\frac{\partial \log \Pr(e|d)}{\partial h_j} = \sum_{\text{all n involving j}} \frac{n}{h_j} + \sum_{\text{all n}' \text{ involving j}} -n' \frac{f'(n')}{(1-f(\mathrm{n}'))}$$

where $f(\mathrm{n}) = h_1^{\mathrm{n}_1} \ldots h_M^{\mathrm{n}_M}$ and $f'(\mathrm{n}) = \frac{f(\mathrm{n})}{h_j}$.

In the above example for diagnosis candidate $(1,1)$

$$\begin{aligned} \mathrm{n}_{10} &= 1 \\ \mathrm{n}_{01} &= 1 \\ \mathrm{n}_{11} &= 0 \\ \mathrm{n}'_{10} &= 1 \\ \mathrm{n}'_{01} &= 0 \\ \mathrm{n}'_{11} &= 1 \end{aligned}$$

As

$$\begin{aligned} \Pr((1,1)) &= h_1^1 \cdot h_2^1 \cdot (h_1 \cdot h_2)^0 \cdot (1-h_1)^1 \cdot (1-h_2)^0 \cdot (1-h_1 \cdot h_2)^1 \\ &= h_1 \cdot h_2 \cdot (1-h_1) \cdot (1-h_1 \cdot h_2) \end{aligned}$$

it follows

$$\begin{aligned}
\frac{\partial \log \Pr(e|d)}{\partial h_1} &= \sum_{x\in\{10,11\}} \frac{n_x}{h_1} - \sum_{x\in\{10,11\}} n'_x \frac{f'(n')}{(1-f(n'))} \\
&= \frac{(n_{10}+n_{11})}{h_1} - n'_{10}\frac{f'(n_{10})}{(1-f(n_{10}))} - n'_{11}\frac{f(n_{11})}{1-f(n_{11}))} \\
&= \frac{(n_{10}+n_{11})}{h_1} - n'_{10}\frac{1}{(1-h_1)} - n'_{11}\frac{h_2}{(1-h_1\cdot h_2)}
\end{aligned}$$

and

$$\begin{aligned}
\frac{\partial \log \Pr(e|d)}{\partial h_2} &= \sum_{x\in\{10,11\}} \frac{n_x}{h_2} - \sum_{x\in\{10,11\}} n'_x \frac{f'(n')}{(1-f(n'))} \\
&= \frac{(n_{01}+n_{11})}{h_2} - n'_{01}\frac{f'(n_{01})}{(1-f(n_{01}))} - n'_{11}\frac{f(n_{11})}{(1-f(n_{11}))} \\
&= \frac{(n_{01}+n_{11})}{h_2} - n'_{01}\frac{1}{(1-h_2)} - n'_{11}\frac{h_1}{(1-h_1\cdot h_2)}
\end{aligned}$$

When substituting the n and n' counters for the above example, the proper expressions are derived. The above expressions are the terms used for ∇. The use of ∇ in the gradient ascent iteration is straightforward.

Bibliography

[Abreu et al., 2008a] Abreu, R., González, A., Zoeteweij, P., and van Gemund, A. J. C. (2008a). Automatic software fault localization using generic program invariants. In Wainwright, R. L. and Haddad, H., editors, *Proceedings of the 23rd Annual ACM Symposium on Applied Computing (SAC'08)*, pages 712–717, Fortaleza, Ceará, Brazil. ACM Press. (Cited on pages 11, 71, and 73.)

[Abreu et al., 2008b] Abreu, R., González, A., Zoeteweij, P., and van Gemund, A. J. C. (2008b). On the performance of fault screeners in software development and deployment. In Gonzalez-Perez, C. and Jablonski, S., editors, *Proceedings of the 3rd International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE'08)*, pages 123–130. INSTICC Press. (Cited on page 11.)

[Abreu et al., 2009a] Abreu, R., González, A., Zoeteweij, P., and van Gemund, A. J. C. (2009a). Using fault screeners for software error detection. In *WEBIST / ENASE 2008 Revised Best Papers*, Lecture Notes in Communications in Computer and Information Science (LNCCIS). Springer-Verlag. (Cited on page 11.)

[Abreu et al., 2009b] Abreu, R., Mayer, W., Stumptner, M., and van Gemund, A. J. C. (2009b). Refining spectrum-based fault localization rankings. In Wainwright, R. L. and Haddad, H., editors, *Proceedings of the 24th Annual ACM Symposium on Applied Computing (SAC'09)*, pages 409–414, Honolulu, Hawaii, USA. ACM Press. (Cited on page 11.)

[Abreu and van Gemund, 2009a] Abreu, R. and van Gemund, A. J. C. (2009a). A low-cost approximate minimal hitting set algorithm and its application to model-based diagnosis. In Bulitko, V. and Beck, J. C., editors, *Proceedings of the 8th Symposium on Abstraction, Reformulation and Approximation (SARA'09)*, Lake Arrowhead, California, USA. AAAI Press. (Cited on page 11.)

[Abreu and van Gemund, 2009b] Abreu, R. and van Gemund, A. J. C. (2009b). Statistics-directed minimal hitting set algorithm. In Nyberg, M., Frisk, E., Krysander, M., and Aslund, J., editors, *Proceedings of the 20th International Workshop on Principles of Diagnosis (DX'09)*, pages 51 – 58, Stockholm, Sweden. (Cited on page 11.)

[Abreu et al., 2009c] Abreu, R., Zoeteweij, P., Golsteijn, R., and van Gemund, A. J. C. (2009c). A practical evaluation of spectrum-based fault localization. *Journal of Systems & Software (JSS)*. (Cited on page 11.)

[Abreu et al., 2006a] Abreu, R., Zoeteweij, P., and van Gemund, A. J. C. (2006a). An evaluation of similarity coefficients for software fault localization. In Jeske, D., Ciardo, G., and Dai, Y.-S., editors, *Proceedings of the 12th*

Pacific Rim International Symposium on Dependable Computing (PRDC'06), pages 39–46, Riverside, California, USA. IEEE Computer Society. (Cited on pages 14, 22, 24, 38, and 71.)

[Abreu et al., 2006b] Abreu, R., Zoeteweij, P., and van Gemund, A. J. C. (2006b). Program spectra analysis in embedded software: A case study. In Lelieveldt, B., Haverkort, B., de Laat, C., and Heijnsdijk, J., editors, *Proceedings of the 12th Annual Conference of the Advanced School for Computing and Imaging (ASCI'06)*, pages 263–269, Lommel, Belgium. Advanced School for Computing & Imaging (ASCI). (Cited on page 43.)

[Abreu et al., 2007] Abreu, R., Zoeteweij, P., and van Gemund, A. J. C. (2007). On the accuracy of spectrum-based fault localization. In McMinn, P., editor, *Proceedings of the Testing: Academia and Industry Conference - Practice And Research Techniques (TAIC PART'07)*, pages 89–98, Windsor, United Kingdom. IEEE Computer Society. (Cited on pages 11, 77, 91, 92, 99, 105, 116, and 134.)

[Abreu et al., 2008c] Abreu, R., Zoeteweij, P., and van Gemund, A. J. C. (2008c). A dynamic modeling approach to software multiple-fault localization. In Grastien, A. and Stumptner, M., editors, *Proceedings of the 19th International Workshop on Principles of Diagnosis (DX'08)*, pages 7–14, Blue Mountains, NSW, Australia. (Cited on pages 78, 86, 89, 90, 105, and 116.)

[Abreu et al., 2008d] Abreu, R., Zoeteweij, P., and van Gemund, A. J. C. (2008d). An observation-based model for fault localization. In Liblit, B. and Rountev, A., editors, *Proceedings of the 6th Workshop on Dynamic Analysis (WODA'08)*, pages 64–70. ACM Press. (Cited on pages 11, 78, 86, 102, and 105.)

[Abreu et al., 2009d] Abreu, R., Zoeteweij, P., and van Gemund, A. J. C. (2009d). A new Bayesian approach to multiple intermittent fault diagnosis. In Boutilier, C., editor, *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI'09)*, pages 653 – 658, Pasadena, California, USA. AAAI Press. (Cited on page 11.)

[Abreu et al., 2009e] Abreu, R., Zoeteweij, P., and van Gemund, A. J. C. (2009e). Spectrum-based multiple fault localization. In Taentzer, G. and Heimdahl, M., editors, *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE'09)*, Auckland, New Zealand. IEEE Computer Society, to appear. (Cited on page 11.)

[Agrawal et al., 1991] Agrawal, H., de Millo, R., and Spafford, E. (1991). An execution backtracking approach to program debugging. *IEEE Software*, 8:21 – 26. (Cited on page 3.)

[Agrawal et al., 1993] Agrawal, H., DeMillo, R. A., and Spafford, E. H. (1993). Debugging with dynamic slicing and backtracking. *Software - Practice and Experience*, 23(6):589–616. (Cited on page 33.)

[Augusteijn, 2002] Augusteijn, L. (2002). Front: A front-end generator for Lex, Yacc and C, release 1.0. `http://front.sourceforge.net/`. (Cited on pages 22 and 43.)

[Avižienis et al., 2004] Avižienis, A., Laprie, J.-C., Randell, B., and Landwehr, C. E. (2004). Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable Secure Computing*, 1(1):11–33. (Cited on page 4.)

[Avriel, 2003] Avriel, M. (2003). *Nonlinear Programming: Analysis and Methods*. Dover Publishing, Mineola, New York, USA. (Cited on pages 88 and 154.)

[Baah et al., 2008] Baah, G. K., Podgurski, A., and Harrold, M. J. (2008). The probabilistic program dependence graph and its application to fault diagnosis. In Ryder, B. G. and Zeller, A., editors, *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'08)*, pages 189 – 200, Seattle, Washington, USA. ACM Press. (Cited on pages 6, 33, 99, and 103.)

[Balzer, 1969] Balzer, R. M. (1969). EXDAMS: Extendible debugging and monitoring system. In *Proceedings of the AFIPS Spring Joint Conference*, volume 34, pages 567 – 580, Montvale, New Jersey, USA. AFIPS Press. (Cited on page 3.)

[Baudry et al., 2006] Baudry, B., Fleurey, F., and Traon, Y. L. (2006). Improving test suites for efficient fault localization. In Osterweil, L. J., Rombach, H. D., and Soffa, M. L., editors, *Proceedings of the 28th International Conference on Software Engineering (ICSE'06)*, pages 82–91, Shanghai, China. ACM Press. (Cited on page 34.)

[Bensalem and Havelund, 2005] Bensalem, S. and Havelund, K. (2005). Dynamic deadlock analysis of multi-threaded programs. In Ur, S., Bin, E., and Wolfsthal, Y., editors, *Proceedings of the 1st International Haifa Hardware and Software Verification and Testing Conference, Revised Selected Papers*, volume 3875 of *Lecture Notes in Computer Science*, pages 208–223, Haifa, Israel. Springer-Verlag. (Cited on page 75.)

[Bloom, 1970] Bloom, B. (1970). Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426. (Cited on page 60.)

[Bolton, 1991] Bolton, H. C. (1991). On the mathematical significance of the similarity index of ochiai as a measure for biogeographical habitats. *Australian Journal of Zoology*, 39(2):143–156. (Cited on page 34.)

[Böttcher, 1995] Böttcher, C. (1995). No faults in structure? How to diagnose hidden interaction. In Mellish, C. S., editor, *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI'95)*, pages 1728–1735, Montreal, Quebec, Canada. Morgan Kaufmann Publishers. (Cited on page 127.)

[Carey et al., 1999] Carey, J., Gross, N., Stepanek, M., and Port, O. (1999). Software hell. *Business Week*, pages 391–411. (Cited on page 1.)

[Chen et al., 2002] Chen, M. Y., Kiciman, E., Fratkin, E., Fox, A., and Brewer, E. A. (2002). Pinpoint: Problem determination in large, dynamic internet services. In Lala, J. H., editor, *Proceedings of the International Conference on Dependable Systems and Networks (DSN'02)*, pages 595–604, Bethesda, Maryland, USA. IEEE Computer Society. (Cited on pages 6, 14, 18, 33, 34, 37, and 38.)

[Console and Torasso, 1991] Console, L. and Torasso, P. (1991). A spectrum of logical definitions of model-based diagnosis. *Computational Intelligence*, 7:133–141. (Cited on page 8.)

[Cousot and Cousot, 1977] Cousot, P. and Cousot, R. (1977). Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the 4th ACM Symposium on Principles of Programming Languages (POPL'77)*, pages 238–252, Los Angeles, California, USA. ACM Press. (Cited on page 126.)

[da Silva Meyer et al., 2004] da Silva Meyer, A., Franco Farcia, A. A., and Pereira de Souza, A. (2004). Comparison of similarity coefficients used for cluster analysis with dominant markers in maize (*Zea mays* L). *Genetics and Molecular Biology*, 27(1):83–91. (Cited on pages 14, 18, 23, and 34.)

[Dallmeier et al., 2005] Dallmeier, V., Lindig, C., and Zeller, A. (2005). Lightweight defect localization for java. In Black, A. P., editor, *Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP'05)*, volume 3586 of *Lecture Notes in Computer Science*, pages 528–550, Glasgow, UK. Springer-Verlag. (Cited on pages 6, 14, 33, and 38.)

[David and Nagaraja, 1970] David, H. A. and Nagaraja, H. N. (1970). *Order Statistics*. John Wiley & Sons. (Cited on page 68.)

[DBX, 1990] DBX (1990). Debugging tools – DBX, SunOS 4.1.1 ed. SUN MICROSYSTEMS, INC. (Cited on page 3.)

[de Kleer, 2007] de Kleer, J. (2007). Diagnosing intermittent faults. In Biswas, G., Koutsoukos, X., and Abdelwahed, S., editors, *Proceedings of the 18th International Workshop on Principles of Diagnosis (DX'07)*, pages 45 – 51, Nashville, Tennessee, USA. (Cited on pages 78, 82, 89, and 90.)

[de Kleer, 2009] de Kleer, J. (2009). Diagnosing multiple persistent and intermittent faults. In Boutilier, C., editor, *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI'09)*, Pasadena, California, USA. AAAI Press. (Cited on pages 78 and 90.)

[De Kleer et al., 1992] De Kleer, J., Mackworth, A. K., and Reiter, R. (1992). Characterizing diagnoses and systems. *Artificial Intelligence*, 56:197–222. (Cited on pages 8 and 81.)

[De Kleer et al., 2008] De Kleer, J., Price, B., Kuhn, L., Do, M., and Zhou, R. (2008). A framework for continuously estimating persistent and intermittent failure probabilities. In Grastien, A. and Stumptner, M., editors, *Proceedings of the 19th International Workshop on Principles of Diagnosis (DX'08)*, pages 22–24, Blue Mountains, NSW, Australia. (Cited on page 90.)

[de Kleer and Williams, 1987] de Kleer, J. and Williams, B. C. (1987). Diagnosing multiple faults. *Artificial Intelligence*, 32(1):97–130. (Cited on pages 7, 8, 33, 77, 81, 119, and 147.)

[de Visser, 2008] de Visser, I. (2008). *Analyzing User Perceived Failure Severity in Consumer Electronics Products*. PhD thesis, Eindhoven University of Technology. (Cited on page 2.)

[Do et al., 2005] Do, H., Elbaum, S. G., and Rothermel, G. (2005). Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal*, 10(4):405–435. (Cited on pages 15, 20, 98, 118, and 133.)

[Dolby et al., 2007] Dolby, J., Vaziri, M., and Tip, F. (2007). Finding bugs efficiently with a sat solver. In Crnkovic, I. and Bertolino, A., editors, *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/SIGSOFT FSE'07)*, pages 195–204. ACM Press. (Cited on page 7.)

[Dowson, 1997] Dowson, M. (1997). The Ariane 5 software failure. *SIGSOFT Software Engineering Notes*, 22(2):84. (Cited on page 2.)

[Engler and Dunbar, 2007] Engler, D. R. and Dunbar, D. (2007). Under-constrained execution: making automatic code destruction easy and scalable. In Rosenblum, D. S. and Elbaum, S. G., editors, *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'07)*, pages 1–4, London, UK. ACM Press. (Cited on page 140.)

[Ernst et al., 2001] Ernst, M. D., Cockrell, J., Griswold, W. G., and Notkin, D. (2001). Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering (TSE)*, 27(2):99–123. (Cited on pages 56, 57, and 73.)

[Ernst et al., 2007] Ernst, M. D., Perkins, J. H., Guo, P. J., McCamant, S., Pacheco, C., Tschantz, M. S., and Xiao, C. (2007). The daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1-3):35–45. (Cited on pages 57 and 73.)

[Esser and Struss, 2007] Esser, M. and Struss, P. (2007). Automated test generation from models based on functional software specifications. In Veloso, M. M., editor, *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI'07)*, pages 2255–2268, Hyderabad, India. AAAI Press. (Cited on page 140.)

[Feldman et al., 2008] Feldman, A., Provan, G., and van Gemund, A. J. C. (2008). Computing minimal diagnoses by greedy stochastic search. In Fox, D. and Gomes, C. P., editors, *Proceedings of the 23rd National Conference on Artificial Intelligence (AAAI'08)*, pages 919–924, Chicago, Illinos, USA. AAAI Press. (Cited on pages 77 and 81.)

[Feldman and van Gemund, 2006] Feldman, A. and van Gemund, A. J. C. (2006). A two-step hierarchical algorithm for model-based diagnosis. In Gil, Y. and Mooney, R., editors, *Proceedings of the 21st National Conference on Artificial Intelligence and the 18th Innovative Applications of Artificial Intelligence Conference (AAAI'06)*, Boston, MA, USA. AAAI Press. (Cited on page 77.)

[Fijany and Vatan, 2004] Fijany, A. and Vatan, F. (2004). New approaches for efficient solution of hitting set problem. In *Proceedings of the Winter International Symposium on Information and communication technologies (WISICT'04)*, volume 58 of *ACM International Conference Proceeding Series*, Cancun, Mexico. Trinity College Dublin. (Cited on page 120.)

[Fijany and Vatan, 2005] Fijany, A. and Vatan, F. (2005). New high performance algorithmic solution for diagnosis problem. In Profet, K., Mattingly, R., and Bryan, E., editors, *Proceedings of the 2005 IEEE Aerospace Conference (IEEEAC'05)*. IEEE Computer Society. (Cited on page 120.)

[Freuder et al., 1995] Freuder, E. C., Dechter, R., Ginsberg, M. L., Selman, B., and Tsang, E. P. K. (1995). Systematic versus stochastic constraint satisfaction. In Mellish, C. S., editor, *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI'95)*, pages 2027–2032, Montréal, Québec, Canada. (Cited on page 120.)

[Friedrich et al., 1996] Friedrich, G., Stumptner, M., and Wotawa, F. (1996). Model-based diagnosis of hardware designs. In Wahlster, W., editor, *Proceedings of the 12th European Conference on Artificial Intelligence (ECAI'96)*, pages 491–495, Budapest, Hungary. John Wiley and Sons, Chichester. (Cited on page 7.)

[Friedrich et al., 1999] Friedrich, G., Stumptner, M., and Wotawa, F. (1999). Model-based diagnosis of hardware designs. *Artificial Intelligence*, 111(1-2):3–39. (Cited on page 7.)

[Garey and Johnson, 1979] Garey, M. R. and Johnson, D. S. (1979). *Computers and Intractability — A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, New York. (Cited on pages 107 and 109.)

[Garfinkel, 2005] Garfinkel, S. (2005). History's worst software bugs. `http://www.wired.com/software/coolapps/news/2005/11/69355/`. (Cited on page 1.)

[Gautama and van Gemund, 2006] Gautama, H. and van Gemund, A. J. C. (2006). Low-cost static performance prediction of parallel stochastic task compositions. *IEEE Transactions on Parallel Distributed Systems*, 17(1):78–91. (Cited on page 69.)

[González, 2007] González, A. (2007). Automatic error detection techniques based on dynamic invariants. Master's thesis, Delft University of Technology and Universidad de Valladolid. (Cited on page 63.)

[Greiner et al., 1989] Greiner, R., Smith, B. A., and Wilkerson, R. W. (1989). A correction to the algorithm in Reiter's theory of diagnosis. *Artificial Intelligence*, 41(1):79–88. (Cited on pages 107 and 119.)

[Groce, 2004] Groce, A. (2004). Error explanation with distance metrics. In Jensen, K. and Podelski, A., editors, *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04), Held as Part of the Joint European Conferences on Theory and Practice of Software (ETAPS'04)*, volume 2988 of *Lecture Notes in Computer Science*, pages 108 – 122, Barcelona, Spain. Springer-Verlag. (Cited on pages 7 and 104.)

[Groce et al., 2006] Groce, A., Chaki, S., Kroening, D., and Strichman, O. (2006). Error explanation with distance metrics. *International Journal on Software Tools for Technology Transfer (STTT)*, 8(3):229–247. (Cited on pages 7, 138, and 139.)

[Gumbel, 1962] Gumbel, E. (1962). Statistical theory of extreme values (main results). In Sarhan, A. and Greenberg, B., editors, *Contributions to Order Statistics*. John Wiley & Sons. (Cited on page 69.)

[Gupta et al., 2005] Gupta, N., He, H., Zhang, X., and Gupta, R. (2005). Locating faulty code using failure-inducing chops. In Redmiles, D. F., Ellman, T., and Zisman, A., editors, *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE'05)*, pages 263 – 272, Long Beach, California, USA. IEEE Computer Society. (Cited on pages 33 and 77.)

[Hailpern and Santhanam, 2002] Hailpern, B. and Santhanam, P. (2002). Software debugging, testing, and verification. *IBM Systems Journal*, 41(1):4–12. (Cited on pages 2 and 13.)

[Halperin et al., 2002] Halperin, D., Heydt-Benjamin, T. S., Ransford, B., Clark, S. S., Defend, B., Morgan, W., Fu, K., Kohno, T., and Maisel, W. H. (2002). Pacemakers and implantable cardiac defibrillators: Software radio attacks and zero-power defenses. In Abadi, M. and Bellovin, S., editors, *Proceedings of the IEEE Symposium on Security and Privacy*, pages 129 – 142, Oakland, California, USA. IEEE Computer Society. (Cited on page 1.)

[Hangal et al., 2005] Hangal, S., Chandra, N., Narayanan, S., and Chakravorty, S. (2005). IODINE: a tool to automatically infer dynamic invariants for hardware designs. In Jr., W. H. J., Martin, G., and Kahng, A. B., editors, *Proceedings of the 42nd Design Automation Conference (DAC'05)*, pages 775–778, San Diego, California, USA. ACM Press. (Cited on page 74.)

[Hangal and Lam, 2002] Hangal, S. and Lam, M. S. (2002). Tracking down software bugs using automatic anomaly detection. In Young, M. and Magee, J., editors, *Proceedings of the 22nd International Conference on Software Engineering (ICSE'02)*, pages 291–301, Orlando, Florida, USA. ACM Press. (Cited on pages 56, 58, 73, and 74.)

[Harrold et al., 1998] Harrold, M., Rothermel, G., Wu, R., and Yi, L. (1998). An empirical investigation of program spectra. *ACM SIGPLAN Notices*, 33(7). (Cited on page 70.)

[Harrold et al., 1993] Harrold, M. J., Gupta, R., and Soffa, M. L. (1993). A methodology for controlling the size of a test suite. *ACM Transactions on Software Engineering and Methodology (ACM TOSEM)*, 2(3):270–285. (Cited on page 146.)

[Harrold et al., 2000] Harrold, M. J., Rothermel, G., Sayre, K., Wu, R., and Yi, L. (2000). An empirical investigation of the relationship between spectra differences and regression faults. *Software Testing, Verification and Reliability*, 10(3):171–194. (Cited on pages 6, 14, 15, and 32.)

[Holzmann, 1997] Holzmann, G. J. (1997). The model checker SPIN. *IEEE Transactions on Software Engineering (TSE)*, 23:279–295. (Cited on page 7.)

[Hutchins et al., 1994] Hutchins, M., Foster, H., Goradia, T., and Ostrand, T. (1994). Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In Taylor, R. N. and Coutaz, J., editors, *Proceedings of the International Conference on Software Engineering (ICSE'94)*. IEEE CS. (Cited on pages 15, 20, and 62.)

[Iverson, 1962] Iverson, K. E. (1962). *A programming language*. John Wiley & Sons, New York, NY, USA. (Cited on page 90.)

[Jain and Dubes, 1988] Jain, A. and Dubes, R. (1988). *Algorithms for clustering data*. Prentice-Hall, Inc. (Cited on pages 17 and 70.)

[Janssen et al., 2009] Janssen, T., Abreu, R., and van Gemund, A. J. C. (2009). Zoltar: A toolset for automatic fault localization. In van der Hoek, A. and Menzies, T., editors, *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE'09) - Tools Track*, Auckland, New Zealand. IEEE Computer Society, to appear. (Cited on page 145.)

[Jeffrey et al., 2008] Jeffrey, D., Gupta, N., and Gupta, R. (2008). Fault localization using value replacement. In Ryder, B. G. and Zeller, A., editors, *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'08)*, pages 119–120, Seattle, Washington, USA. ACM Press. (Cited on page 33.)

[Jones and Harrold, 2005] Jones, J. A. and Harrold, M. J. (2005). Empirical evaluation of the tarantula automatic fault-localization technique. In Redmiles, D. F., Ellman, T., and Zisman, A., editors, *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE'05)*, pages 273–282, Long Beach, California, USA. IEEE Computer Society. (Cited on pages 7, 18, 23, 34, and 134.)

[Jones et al., 2007] Jones, J. A., Harrold, M. J., and Bowring, J. F. (2007). Debugging in parallel. In Rosenblum, D. S. and Elbaum, S. G., editors, *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'07)*, pages 16–26, London, UK. ACM Press. (Cited on page 105.)

[Jones et al., 2002] Jones, J. A., Harrold, M. J., and Stasko, J. T. (2002). Visualization of test information to assist fault localization. In Young, M. and Magee, J., editors, *Proceedings of the 22rd International Conference on Software Engineering (ICSE'02)*, pages 467–477, Orlando, Florida, USA. ACM Press. (Cited on pages 6, 14, 33, 38, and 77.)

[Jones and Kelly, 1997] Jones, R. W. M. and Kelly, P. H. J. (1997). Backwards-compatible bounds checking for arrays and pointers in c programs. In Kamkar, M., editor, *Proceedings of the 3rd International Workshop on Automatic Debugging (AADEBUG'97)*, pages 13–26, Linköping, Sweden. ACM Press. (Cited on pages 5 and 75.)

[Keijzers et al., 2008] Keijzers, J., den Ouden, E., and Lu, Y. (2008). Usability benchmark study of commercially available smart phones: cell phone type platform, pda type platform and pc type platform. In ter Hofte, G. H., Mulder, I., and de Ruyter, B. E. R., editors, *Proceedings of the 10th Conference on Human-Computer Interaction with Mobile Devices and Services, Mobile HCI*, pages 265–272. ACM Press. (Cited on page 2.)

[Kephart and Chess, 2003] Kephart, J. and Chess, D. (2003). The vision of autonomic computing. *Computer*, 36(1). (Cited on page 55.)

[Knuth, 1997] Knuth, D. E. (1997). *The art of computer programming, volume 2 (3rd ed.): seminumerical algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. (Cited on page 61.)

[Köb and Wotawa, 2005] Köb, D. and Wotawa, F. (2005). A comparison of fault explanation and localization. In Dearden, R. and Narasimhan, S., editors, *Proceedings of the 16th International Workshop on Principles of Diagnosis (DX'05)*, pages 157–162, Monterey, California, USA. (Cited on page 140.)

[Korel and Laski, 1988] Korel, B. and Laski, J. (1988). Dynamic program slicing. *Information Processing Letters*, 29:155–163. (Cited on page 15.)

[Kuhn et al., 2008] Kuhn, L., Price, B., de Kleer, J., Do, M., and Zhou, R. (2008). Pervasive diagnosis: Integration of active diagnosis into production plans. In Fox, D. and Gomes, C. P., editors, *Proceedings of the 23rd National*

Conference on Artificial Intelligence (AAAI'08), pages 1306–1312, Chicago, Illinois, USA. AAAI Press. (Cited on page 90.)

[Lattner and Adve, 2004] Lattner, C. and Adve, V. S. (2004). LLVM: A compilation framework for lifelong program analysis & transformation. In Dupré, M., Drach, N., and Temam, O., editors, *Proceedings of the 2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO'04),*, pages 75–88, San Jose, California, USA. IEEE Computer Society. (Cited on page 63.)

[Li et al., 2007] Li, Z., Harman, M., and Hierons, R. M. (2007). Search algorithms for regression test case prioritization. *IEEE Transactions on Software Engineering (TSE)*, 33(4):225–237. (Cited on page 146.)

[Liblit, 2008] Liblit, B. (2008). Cooperative debugging with five hundred million test cases. In Ryder, B. G. and Zeller, A., editors, *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'08)*, pages 119–120, Seattle, Washington, USA. ACM Press. (Cited on page 6.)

[Liblit et al., 2005] Liblit, B., Naik, M., Zheng, A. X., Aiken, A., and Jordan, M. I. (2005). Scalable statistical bug isolation. In Sarkar, V. and Hall, M. W., editors, *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI'05)*, pages 15–26, Chicago, Illinois, USA. ACM Press. (Cited on page 6.)

[Lin and Jiang, 2002] Lin, L. and Jiang, Y. (2002). Computing minimal hitting sets with genetic algorithms. In *Proceedings of the 13rd International Workshop on Principles of Diagnosis (DX'02)*, Semmering, Austria. (Cited on page 120.)

[Lin and Jiang, 2003] Lin, L. and Jiang, Y. (2003). The computation of hitting sets: review and new algorithms. *Information Processing Letters*, 86(4):177–184. (Cited on page 120.)

[Lions, 1996] Lions, J. L. (1996). Ariane 5: Flight 501 failure. Report, European Space Agency. (Cited on page 2.)

[Liu et al., 2006] Liu, C., Fei, L., Yan, X., Han, J., and Midkiff, S. (2006). Statistical debugging: A hypothesis testing-based approach. *IEEE Transactions on Software Engineering (TSE)*, 32(10):831–848. (Cited on pages 6, 7, 33, 77, 99, and 100.)

[Liu and Han, 2006] Liu, C. and Han, J. (2006). Failure proximity: a fault localization-based approach. In Young, M. and Devanbu, P. T., editors, *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT FSE'06)*, pages 46–56, Portland, Oregon, USA. ACM Press. (Cited on page 6.)

[Liu et al., 2005] Liu, C., Yan, X., Fei, L., Han, J., and Midkiff, S. P. (2005). SOBER: statistical model-based bug localization. In Wermelinger, M. and

Gall, H., editors, *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/SIGSOFT FSE)*, pages 286–295, Lisbon, Portugal. ACM Press. (Cited on pages 134, 137, and 138.)

[Lorenzoli et al., 2008] Lorenzoli, D., Mariani, L., and Pezzè, M. (2008). Automatic generation of software behavioral models. In Schäfer, W., Dwyer, M. B., and Gruhn, V., editors, *Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*, pages 501–510, Leipzig, Germany. ACM Press. (Cited on page 145.)

[Mayer, 2007] Mayer, W. (2007). *Static and Hybrid Analysis in Model-based Debugging*. PhD thesis, School of Computer and Information Science, University of South Australia. (Cited on pages 8 and 123.)

[Mayer et al., 2008] Mayer, W., Abreu, R., Stumptner, M., and van Gemund, A. J. C. (2008). Prioritizing model-based debugging diagnostic reports. In Grastien, A., Stumptner, M., and Mayer, W., editors, *Proceedings of the 19th International Workshop on Principles of Diagnosis (DX'08)*, pages 127–134, Blue Mountains, New South Wales, Australia. (Cited on pages 11, 131, and 139.)

[Mayer and Stumptner, 2007a] Mayer, W. and Stumptner, M. (2007a). Model-based debugging - state of the art and future challenges. *Electronic Notes on Theoretical Computer Science*, 174(4):61–82. (Cited on page 140.)

[Mayer and Stumptner, 2007b] Mayer, W. and Stumptner, M. (2007b). Models and tradeoffs in model-based debugging. In Biswas, G., Koutsoukos, X., and Abdelwahed, S., editors, *Proceedings of the 18th International Workshop on Principles of Diagnosis (DX'07)*, pages 138 – 145, Nashville, Tennessee, USA. (Cited on page 104.)

[Mayer and Stumptner, 2008] Mayer, W. and Stumptner, M. (2008). Evaluating models for model-based debugging. In Ireland, A. and Visser, W., editors, *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE'08)*, pages 128–137, L'Aquila, Italy. ACM Press. (Cited on pages 7, 77, 78, 83, 104, 123, 125, 126, 127, 128, 135, and 139.)

[MBSD, 2008] MBSD (2008). Model-based software debugging website, School of Computer and Information Science, University of South Australia. `http://www.acrc.unisa.edu.au/groups/kse/mbsd/index.html`. (Cited on page 125.)

[Menzies et al., 2007] Menzies, T., Owen, D., and Richardson, J. (2007). The strangest thing about software. *Computer*, 40(1):54–60. (Cited on page 146.)

[Morell, 1990] Morell, L. J. (1990). A theory of fault-based testing. *IEEE Transactions on Software Engineering (TSE)*, 16(8):844–857. (Cited on page 26.)

[Musuvathi and Engler, 2003] Musuvathi, M. and Engler, D. R. (2003). Some lessons from using static analysis and software model checking for bug finding. *Electronic Notes Theoretical Computer Science*, 89(3). (Cited on page 125.)

[Nainar et al., 2007] Nainar, P. A., Chen, T., Rosin, J., and Liblit, B. (2007). Statistical debugging using compound boolean predicates. In Rosenblum, D. S. and Elbaum, S. G., editors, *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'07)*, pages 5–15, London, UK, July. ACM Press. (Cited on page 6.)

[Nica and Wotawa, 2008] Nica, M. and Wotawa, F. (2008). From constraint representations of sequential code and program annotations to their use in debugging. In Ghallab, M., Spyropoulos, C. D., Fakotakis, N., and Avouris, N. M., editors, *Proceedings of the 18th European Conference on Artificial Intelligence (ECAI'08)*, volume 178 of *Frontiers in Artificial Intelligence and Applications*, pages 797–798, Patras, Greece. IOS Press. (Cited on page 7.)

[Nimmer and Ernst, 2001] Nimmer, J. W. and Ernst, M. D. (2001). Static verification of dynamically detected program invariants: Integrating daikon and ESC/Java. *Electronic Notes Theoretical Computer Science*, 55(2). (Cited on page 140.)

[NXP, 2009] NXP (2009). NXP Semiconductors website, NXP Semiconductors. `http://www.nxp.com`. (Cited on pages 9, 22, and 45.)

[Pattabiraman et al., 2005] Pattabiraman, K., Kalbarczyk, Z., and Iyer, R. K. (2005). Application-based metrics for strategic placement of detectors. In Cao, J. and Zhang, D., editors, *Proceedings of the 11th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC'05)*, pages 75–82, Changsha, Hunan, China. IEEE Computer Society. (Cited on page 146.)

[Patterson et al., 2002] Patterson, D., Brown, A., Broadwell, P., Candea, G., Chen, M., Cutler, J., Enriquez, P., Fox, A., Kiciman, E., Merzbacher, M., Oppenheimer, D., Sastry, N., Tetzlaff, W., Traupman, J., and Treuhaft, N. (2002). Recovery Oriented Computing (ROC): Motivation, definition, techniques, and case studies. Technical Report UCB/CSD-02-1175, University of California at Berkeley. (Cited on pages 5 and 55.)

[Pietersma and van Gemund, 2006] Pietersma, J. and van Gemund, A. J. C. (2006). Temporal versus spatial observability in model-based diagnosis. In Lin, C.-T., editor, *Proceedings of 2006 IEEE International Conference on Systems, Man, and Cybernetics (SMC'06)*, pages 5325–5331, Taipei, Taiwan. IEEE Computer Society. (Cited on pages 77 and 80.)

[Pytlik et al., 2003] Pytlik, B., Renieris, M., Krishnamurthi, S., and Reiss, S. (2003). Automated fault localization using potential invariants. In Ronsse, M., editor, *Proceedings of the International Workshop on Automated and Analysis-Driven Debugging (AADEBUG'03)*. ACM Press. (Cited on pages 56, 57, 73, and 74.)

[Qasem and Prügel-Bennett, 2008] Qasem, M. and Prügel-Bennett, A. (2008). Complexity of max-sat using stochastic algorithms. In Ryan, C. and Keijzer, M., editors, *Proceedings of the 10th annual conference on Genetic and evolutionary computation (GECCO'08)*, pages 615–616, Atlanta, Georgia, USA. ACM Press. (Cited on page 120.)

[Quine, 1955] Quine, W. (1955). A way to simplify truth functions. *American Mathematical Monthly*, 62:627 – 631. (Cited on page 120.)

[Racunas et al., 2007] Racunas, P., Constantinides, K., Manne, S., and Mukherjee, S. S. (2007). Perturbation-based fault screening. In Mudge, T., editor, *Proceedings of the 13rd International Conference on High-Performance Computer Architecture (HPCA-13)*, pages 169–180, Phoenix, Arizona, USA. IEEE Computer Society. (Cited on pages 56, 57, 58, 59, 60, 73, and 74.)

[Reiter, 1987] Reiter, R. (1987). A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95. (Cited on pages 81, 107, 119, 125, 127, and 139.)

[Renieris and Reiss, 2003] Renieris, M. and Reiss, S. P. (2003). Fault localization with nearest neighbor queries. In Grundy, J. and Penix, J., editors, *Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE'03)*, pages 30–39, Montreal, Canada. IEEE Computer Society. (Cited on pages 6, 33, 35, 77, 91, 92, 99, 134, and 137.)

[Reps et al., 1997] Reps, T. W., Ball, T., Das, M., and Larus, J. R. (1997). The use of program profiling for software maintenance with applications to the year 2000 problem. In Jazayeri, M. and Schauer, H., editors, *Proceedings of the 6th European Software Engineering Conference Held Jointly with the 5th ACM SIGSOFT Symposium on Foundations of Software Engineering (ESEC / SIGSOFT FSE)*, volume 1301 of *Lecture Notes in Computer Science*, pages 432–449, Zurich, Switzerland. Springer-Verlag. (Cited on pages 14, 15, 32, and 37.)

[Rogerson, 2002] Rogerson, S. (2002). The chinook helicopter disaster. *The Institute for the Management of Information Systems (IMIS)*, 12(2). (Cited on page 2.)

[RTI, 2002] RTI (2002). Planning report 02-3: The economic impacts of inadequate infrastructure for software testing. Planning Report 02-3, National Institute of Standards and Technology. (Cited on pages 4 and 123.)

[Rudell, 1986] Rudell, R. L. (1986). Multiple-valued logic minimization for pla synthesis. Technical Report UCB/ERL M86/65, EECS Department, University of California, Berkeley. (Cited on page 120.)

[Ruwase and Lam, 2004] Ruwase, O. and Lam, M. S. (2004). A practical dynamic buffer overflow detector. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'04)*, San Diego, California, USA. The Internet Society. (Cited on page 75.)

[Santelices et al., 2008] Santelices, R. A., Chittimalli, P. K., Apiwattanapong, T., Orso, A., and Harrold, M. J. (2008). Test-suite augmentation for evolving software. In Ireland, A. and Visser, W., editors, *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE'08)*, pages 218–227, L'Aquila, Italy. ACM Press. (Cited on page 146.)

[Savage et al., 1997] Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., and Anderson, T. E. (1997). Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411. (Cited on page 75.)

[Seward and Nethercote, 2005] Seward, J. and Nethercote, N. (2005). Using Valgrind to detect undefined value errors with bit-precision. In *Proceedings of the 2005 USENIX Annual Technical Conference, General Track*, pages 17–30, Anaheim, California, USA. (Cited on page 75.)

[Sharygina et al., 2009] Sharygina, N., Tonetta, S., and Tsitovich, A. (2009). The synergy of precise and fast abstractions for program verification. In Shin, S. Y. and Ossowski, S., editors, *Proceedings of the 2009 ACM Symposium on Applied Computing (SAC'09)*, pages 566–573, Honolulu, Hawaii, USA. (Cited on page 7.)

[Smith and Kapfhammer, 2009] Smith, A. M. and Kapfhammer, G. M. (2009). An empirical study of incorporating cost into test suite reduction and prioritization. In Wainwright, R. L. and Haddad, H., editors, *Proceedings of the 24th Annual ACM Symposium on Applied Computing (SAC'09)*, pages 461–467, Honolulu, Hawaii, USA. ACM Press. (Cited on page 146.)

[Sözer, 2009] Sözer, H. (2009). *Architecting Fault-Tolerant Software Systems*. PhD thesis, University of Twente. (Cited on pages 5, 53, and 55.)

[Stallman, 1994] Stallman, R. (1994). Debugging with GDB – The GNU source level debugger. Free Software Foundation. (Cited on page 3.)

[Steimann and Bertchler, 2009] Steimann, F. and Bertchler, M. (2009). A simple coverage-based locator for multiple faults. In Offutt, J. and Runeson, P., editors, *Proceedings of the 2nd International Conference on Software Testing, Verification, and Validation (ICST'09)*, Denver, CO, USA. IEEE Computer Society. (Cited on page 105.)

[Steimann et al., 2008] Steimann, F., Eichstädt-Engelen, T., and Schaaf, M. (2008). Towards raising the failure of unit tests to the level of compiler-reported errors. In Paige, R. F. and Meyer, B., editors, *Proceedings of the 46th International Conference on Objects, Components, Models and Patterns (TOOLS EUROPE'08)*, volume 11 of *Lecture Notes in Business Information Processing*, pages 60–79. Springer-Verlag. (Cited on page 6.)

[Struss and Dressler, 1989] Struss, P. and Dressler, O. (1989). Physical Negation: Integrating fault models into the general diagnostic engine. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI'89)*, pages 1318–1323. (Cited on page 8.)

[Sweetman, 2006] Sweetman, D. (2006). *See MIPS Run, Second Edition*. Morgan Kaufmann Publishers Inc., San Francisco, California, USA. (Cited on page 40.)

[Tallam and Gupta, 2005] Tallam, S. and Gupta, N. (2005). A concept analysis inspired greedy algorithm for test suite minimization. In Ernst, M. D. and Jensen, T. P., editors, *Proceedings of the 2005 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering (PASTE'05)*, pages 35–42, Lisbon, Portugal. ACM Press. (Cited on page 146.)

[Tip, 1995] Tip, F. (1995). A survey of program slicing techniques. *Journal of Programming Languages*, 3(3). (Cited on page 104.)

[Trader, 2009] Trader (2005 – 2009). Trader project website, Embedded Systems Institute. `http://www.esi.nl/trader/`. (Cited on pages 4 and 40.)

[van Ommering et al., 2000] van Ommering, R. C., van der Linden, F., Kramer, J., and Magee, J. (2000). The Koala component model for consumer electronics software. *IEEE Computer*, 33(3):78–85. (Cited on page 40.)

[Vayani, 2007] Vayani, R. (2007). Improving automatic software fault localization. Master's thesis, Delft University of Technology. Best thesis award, Faculty of Mathematics and Computer Science. (Cited on page 101.)

[Visser et al., 2003] Visser, W., Havelund, K., Brat, G. P., Park, S., and Lerda, F. (2003). Model checking programs. *Automated Software Engineering*, 10(2):203–232. (Cited on page 7.)

[Visser and Mehlitz, 2005] Visser, W. and Mehlitz, P. C. (2005). Model checking programs with java pathfinder. In Godefroid, P., editor, *Proceedings of the 12th International SPIN Workshop on Model Checking Software, San Francisco, CA, USA, August 22-24, 2005*, volume 3639 of *Lecture Notes in Computer Science*, page 27, San Francisco, CA, USA. Springer. (Cited on page 7.)

[Voas, 1992] Voas, J. (1992). PIE: A dynamic failure based technique. *IEEE Transactions on Software Engineering (TSE)*, 18(8):717–727. (Cited on page 26.)

[Wieland, 2001] Wieland, D. (2001). *Model-based Debugging of Java Programs Using Dependencies*. PhD thesis, Vienna University of Technology. (Cited on page 105.)

[Williams and Ragno, 2007] Williams, B. and Ragno, R. (2007). Conflict-directed A* and its role in model-based embedded systems. *Discrete Applied Mathematics*, 155(12). (Cited on page 81.)

[Wong et al., 2008] Wong, E., Wei, T., Qi, Y., and Zhao, L. (2008). A crosstab-based statistical method for effective fault localization. In Hierons, R. and Mathur, A., editors, *Proceedings of the 1st International Conference on Software Testing, Verification, and Validation (ICST'08)*, pages 42–51, Lillehammer, Norway. IEEE Computer Society. (Cited on pages 6, 7, 33, 99, 134, and 137.)

[Wotawa, 2001] Wotawa, F. (2001). A variant of Reiter's hitting-set algorithm. *Information Processing Letters*, 79(1):45–51. (Cited on pages 107 and 119.)

[Wotawa, 2002] Wotawa, F. (2002). On the relationship between model-based debugging and program slicing. *Artificial Intelligence*, 135(1-2):125–143. (Cited on page 104.)

[Wotawa et al., 2002] Wotawa, F., Stumptner, M., and Mayer, W. (2002). Model-based debugging or how to diagnose programs automatically. In Hendtlass, T. and Ali, M., editors, *Proceedings of IAE/AIE 2002*, volume 2358 of *LNCS*, pages 746–757, Cairns, Australia. Springer-Verlag. (Cited on pages 7, 33, 77, and 104.)

[Xie and Notkin, 2005] Xie, T. and Notkin, D. (2005). Checking inside the black box: Regression testing by comparing value spectra. *IEEE Transactions on Software Engineering (TSE)*, 31(10):869–883. (Cited on page 33.)

[Xie and Engler, 2003] Xie, Y. and Engler, D. R. (2003). Using redundancies to find errors. *IEEE Transactions on Software Engineering (TSE)*, 29(10):915–928. (Cited on page 140.)

[Yang and Evans, 2004] Yang, J. and Evans, D. (2004). Automatically inferring temporal properties for program evolution. In Briand, L. and Voas, J., editors, *Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE'04)*, pages 340–351, Saint-Malo, Bretagne, France. IEEE Computer Society. (Cited on page 73.)

[Yilmaz et al., 2008] Yilmaz, C., Paradkar, A. M., and Williams, C. (2008). Time will tell: fault localization using time spectra. In Schäfer, W., Dwyer, M. B., and Gruhn, V., editors, *Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*, pages 81–90, Leipzig, Germany. ACM Press. (Cited on pages 6 and 33.)

[Yilmaz and Williams, 2007] Yilmaz, C. and Williams, C. (2007). An automated model-based debugging approach. In Stirewalt, R. E. K., Egyed, A., and Fischer, B., editors, *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE'07)*, pages 174–183, Atlanta, Georgia, USA. ACM Press. (Cited on page 139.)

[Yu et al., 2008] Yu, Y., Jones, J. A., and Harrold, M. J. (2008). An empirical study of the effects of test-suite reduction on fault localization. In Schäfer, W., Dwyer, M. B., and Gruhn, V., editors, *Proceedings of the 30th International*

Conference on Software Engineering (ICSE'08), pages 201–210, Leipzig, Germany. ACM Press. (Cited on page 34.)

[Zeller, 2002] Zeller, A. (2002). Isolating cause-effect chains from computer programs. In *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE'02)*, pages 1 – 10, Charleston, South Carolina, USA. ACM Press. (Cited on pages 33, 77, 99, and 103.)

[Zeller and Hildebrandt, 2002] Zeller, A. and Hildebrandt, R. (2002). Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200. (Cited on page 33.)

[Zeller and Lütkehaus, 1996] Zeller, A. and Lütkehaus, D. (1996). DDD – A free graphical front-end for UNIX debuggers. *ACM SIGPLAN Notices*, 31(1):22–27. (Cited on page 3.)

[Zhang et al., 2006] Zhang, X., Gupta, N., and Gupta, R. (2006). Locating faults through automated predicate switching. In Osterweil, L. J., Rombach, H. D., and Soffa, M. L., editors, *Proceedings of the 28th International Conference on Software Engineering (ICSE'06)*, pages 272–281, Shanghai, China. (Cited on page 33.)

[Zhao and Ouyang, 2007] Zhao, X. and Ouyang, D. (2007). Improved algorithms for deriving all minimal conflict sets in model-based diagnosis. In Huang, D.-S., Heutte, L., and Loog, M., editors, *Proceedings of the 3rd International Conference on Intelligent Computing (ICIC'07), Advanced Intelligent Computing Theories and Applications. With Aspects of Theoretical and Methodological Issues*, volume 4681 of *Lecture Notes in Computer Science*, pages 157–166, Qingdao, China. Springer-Verlag. (Cited on page 120.)

[Zheng et al., 2006] Zheng, A. X., Jordan, M. I., Liblit, B., Naik, M., and Aiken, A. (2006). Statistical debugging: simultaneous identification of multiple bugs. In Cohen, W. W. and Moore, A., editors, *Proceedings of the 23rd International Conference on Machine Learning (ICML'06)*, volume 148 of *ACM International Conference Proceeding Series*, pages 1105–1112, Pittsburgh, Pennsylvania, USA. ACM Press. (Cited on page 6.)

[Zoeteweij et al., 2007] Zoeteweij, P., Abreu, R., Golsteijn, R., and van Gemund, A. J. C. (2007). Diagnosis of embedded software using program spectra. In Leaney, J., Rozenblit, J., and Peng, J., editors, *Proceedings 14th International Conference on the Engineering of Computer Based Systems (ECBS'07)*, pages 213 – 218. IEEE Computer Society. (Cited on pages 11 and 14.)

Summary

Spectrum-based Fault Localization in Embedded Software

– Rui Abreu –

Locating software components that are responsible for observed failures is a time-intensive and expensive phase in the software development cycle. Automatic fault localization techniques aid developers/testers in pinpointing the root cause of software failures, as such reducing the debugging effort. Automatic fault localization has been an active area of research in the past years.

Current approaches to automatic software fault localization can be classified as either (1) statistics-based approaches, or (2) reasoning approaches. This distinction is based on the required amount of knowledge about the program's internal component structure and behavior. Statistics-based fault localization techniques such as Spectrum-based Fault Localization (SFL) use abstraction of program traces (also known as program spectra) to find a statistical relationship between source code locations and observed failures. Although SFL's modeling costs and computational complexity are minimal, its diagnostic accuracy is inherently limited since no reasoning is used. In contrast to SFL, model-based reasoning approaches use prior knowledge of the program, such as component interconnection and statement semantics, to build a model of the correct behavior of the system. On the one hand, model-based reasoning approaches deliver higher diagnostic accuracy, but on the other hand, they suffer from high computation complexity.

In this thesis, we thoroughly studied the fundamental limitations of SFL. In particular, we studied its diagnostic accuracy as a function of similarity coefficient, quantity of observations, and quality of the error detectors. As a result of this study, we discovered a new similarity coefficient (Ochiai), known from the molecular biology community. Ochiai consistently outperforms all coefficients investigated, including those used by related approaches. Furthermore, we present a novel, low-cost, Bayesian reasoning approach to spectrum-based multiple fault localization, dubbed Barinel. A central feature of our approach is the use of a generic, intermittent component failure model. The novelty of this model lies in the computation of the component intermittency rate as part of the posterior candidate probability computation using a maximum likelihood estimation procedure, rather than using previous approaches' approximations. This procedure optimally exploits all information contained in the program spectra. Our synthetic and real software experiments show that Barinel outperforms previous approaches to fault localization.

Furthermore, this thesis reports on the following additional studies. First, we studied the capabilities of simple, generic program invariants to replace test oracles, so as to achieve total automation of the fault localization process. We verified that, despite the simplicity of the program invariants (and therefore considerable rates of false positives and/or negatives), the diagnostic performance of SFL is similar to the combination of SFL and test oracles.

Second, to scale to large systems, reasoning approaches such as Barinel depends on low-cost algorithms to compute the set of diagnosis candidates. We investigated the possibility of using an SFL-based heuristic to focus the computation of valid diagnosis candidates. We show that the SFL-based heuristic is suitable to derive the set of candidates as the search is focused by visiting candidates in best-first order (aiming to capture the most relevant probability mass in the shortest amount of time). Therefore, our algorithm, Staccato, is order of magnitude faster than, e.g., brute-force approaches, rendering our reasoning approach amenable to large programs.

Finally, we studied whether SFL can be integrated with existing model-based software debugging approaches (MBSD) to reduce their high time complexity, while improving their diagnostic quality. We have shown that the combination of SFL with MBSD focus the debugging process to relevant parts of the program. Specially compared to MBSD, we have shown that our algorithm has lower complexity, making it scale to large programs.

Samenvatting

Spectrum-based Fault Localization in Embedded Software
– Rui Abreu –

Het lokaliseren van software componenten verantwoordelijk voor waargenomen fouten is een intensieve en kostbare fase van het software-ontwikkelingsproces. Automatische foutlokalisatietechnieken reduceren debugging tijd door ontwikkelaars en testers te helpen de hoofdoorzaak van programmastoringen te achterhalen. Automatische foutlokalisatie is gedurende de laatste jaren een actief onderzoeksgebied geweest.

Huidige benaderingen van automatische foutlokalisatie kunnen worden geklassificeerd als (1) gebaseerd op statistiek, of (2) gebaseerd op logisch redeneren. Dit onderscheid is gebaseerd op de vereiste hoeveel informatie over de interne componentenstructuur en het gedrag van een programma. Statistiek-gebaseerde foutlokalisatietechnieken zoals Spectrum-gebaseerde Fout Lokalisatie (SFL) gebruiken geabstraheerde programmatraces (ook wel programmaspectra genoemd) om een statistische relatie te vinden tussen broncodelokaties en waargenomen programmastoringen. Alhoewel SFL weinig modelleertijd vereist en een minimale computationele complexiteit heeft, heeft diens diagnostische precisie inherente beperkingen, aangezien geen formele logica wordt gebruikt. In tegenstelling to SFL gebruiken model-gebaseerde benaderingen wel a priori kennis van een programma, zoals component-interrelaties en taalsemantiek, om correct systeemgedrag te modelleren. Enerzijds bieden model-gebaseerde benaderingen een hogere diagnostische precisie, anderzijds hebben zij last van een hogere computationele complexiteit.

In dit proefschrift hebben we de fundamentele beperkingen van SFL grondig bestudeerd. We hebben met name de diagnostiche precisie bestudeerd, en de invloed daarop van similarity coëfficiënten, aantallen observaties, en kwaliteit van de foutdetectiebenaderingen. Tijdens dit onderzoek hebben we een nieuwe similarity coëfficiënt gevonden (Ochiai), die al eerder bekend was binnen de moleculaire biologie. Ochiai presteert consequent beter dan alle andere bestudeerde coëfficiënten, waaronder die van gerelateerde benaderingen. Verder introduceren we een nieuwe, goedkope, Bayesiaanse benadering voor spectrumgebaseerde lokalisatie van meerdere fouten, genaamd Barinel. Een belangrijk kenmerk van onze benadering is het gebruik van een generiek model voor onregelmatig optredende componentstoringen. Het vernieuwende van dit model is de berekening van de mate waarin componenten falen als onderdeel van de berekening van de diagnostische waarden van kandidaat-lokaties. Hierbij wordt gebruik gemaakt van een meest aannemelijke schatter methode in plaats van de schattingen zoals gebruikt door voorgaande bena-

deringen. Deze procedure benut optimaal alle informatie aanwezig in de programmaspectra. Zowel onze synthetische als echte experimenten tonen aan dat Barinel beter presteert dan voorgaande benaderingen van foutlokalisatie.

Bovendien bevat dit proefschrift nog de volgende additionele studies. Allereerst hebben we de bekeken of het mogelijk is om eenvoudige, generieke programma-invarianten te gebruiken om test-orakels te vervangen, waarmee volledige automatisering van het foutlokalisatieproces mogelijk wordt. Deze studie bevestigde dat, ondanks de eenvoud van de programma-invarianten (en het daardoor relatief hoge aantal fout-positieven en fout-negatieven), de diagnostische kwaliteit van SFL vergelijkbaar blijft met de combinatie van SFL en test-orakels.

Ten tweede, om naar grote systemen te schalen, hebben benaderingen zoals Barinel goedkope algoritmes nodig die de verzameling diagnose-kandidaten kunnen berekenen. Daarom hebben we onderzocht of we een SFL-gebaseerde heuristiek kunnen gebruiken om het aantal diagnostische mogelijkheden te beperken. Ons onderzoek toont aan dat de heuristiek snel een passende verzameling diagnose-kandidaten oplevert, doordat het algoritme eerst de meest waarschijnlijke kandidaten selecteert (waardoor de grootst mogelijke totale waarschijnlijkheid in een zo kort mogelijke tijd wordt behaald). Hierdoor is ons algoritme, Staccato, orden van grootte sneller dan bijvoorbeeld een uitputtende zoekopdracht, en daarmee toepasbaar bij grote programma's.

Als laatste hebben we onderzocht of SFL geïntegreerd kan worden met bestaande model-gebaseerde software debugging benaderingen (MBSD), zodat de computationele complexiteit gereduceerd kan worden, en tegelijkertijd de diagnostische kwaliteit verbetert. We tonen aan dat de combinatie van SFL en MBSD het debugging proces kunnen begeleiden naar relevante delen van het programma. Bovendien heeft deze combinatie een lagere computationele complexiteit dan MBSD, en schaalt het naar grote programma's.

Curriculum Vitæ

PERSONAL DATA

Full Name
Rui Filipe Lima Maranhão de Abreu

Date of birth
August 14th, 1981

Place of birth
Fão, Portugal

EDUCATION

August 2005 – September 2009
PhD Student at Delft University of Technology, the Netherlands, under the supervision of prof.dr.ir. A.J.C. van Gemund;

September 1999 – September 2004
Degree on Systems and Computer Science Engineering, specialization in information technologies, University of Minho, Portugal;

September 2002 –February 2003
Followed courses of the Software Technology Master program as an Erasmus Exchange Student, Utrecht University, the Netherlands;

September 1996 – June 1999
Secondary Education Diploma, specialization in informatics, Escola Secundária de Santa Maria Maior, Viana do Castelo, Portugal.

WORK EXPERIENCE

August 2005 – September 2009
Research Assistant at Delft University of Technology and Embedded Systems Institute, the Netherlands;

May 2007 – May 2008
Guest Software Developer at NXP Semiconductors, Eindhoven, the Netherlands;

October 2004 –June 2005
Research Trainee at Philips Research Labs, Eindhoven, the Netherlands;

March 2004 – September 2004
Software-Developer Trainee at Siemens S.A., Porto, Portugal.

PUBLICATIONS

1. R. Abreu, A. González, P. Zoeteweij, and A.J.C. van Gemund, **Using Fault Screeners to Software Error Detection**. In (J. Cordeiro, L.A. Maciaszek, S. Hammoudi and J. Filipe, eds.) WEBIST / ENASE 2008 Revised Best Papers, Lecture Notes in Communications in Computer and Information Science (LNCCIS). Springer-Verlag. (to appear)

2. R. Abreu, P. Zoeteweij, and A.J.C. van Gemund, **Spectrum-based Multiple Fault Localization**. In Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE'09), Auckland, New Zealand, November 2009. IEEE Computer Science. (to appear)

3. R. Abreu, P. Zoeteweij, and A.J.C. van Gemund, **Zoltar: A Toolset for Automatic Fault Localization**. In Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE'09) - Tools Track, Auckland, New Zealand, November 2009. IEEE Computer Science. (to appear)

4. R. Abreu, P. Zoeteweij, A.J.C. van Gemund, **Fault Localization of Embedded Software**. In (R. Mathijssen, ed.) TRADER: Reliability of High-Volume Consumer Products, pp. 103-112, ISBN 978-90-78679-04-2, September 2009. Embedded Systems Institute, Eindhoven, the Netherlands.

5. P. Zoeteweij, R. Abreu, A.J.C. van Gemund, **Spectrum-Based Fault Localization in Practice**. In (R. Mathijssen, ed.) TRADER: Reliability of High-Volume Consumer Products, pp. 112-124, ISBN 978-90-78679-04-2, September 2009. Embedded Systems Institute, Eindhoven, the Netherlands.

6. R. Abreu, P. Zoeteweij, and A.J.C. van Gemund, **Localizing Software Faults Simultaneously**. In Proceedings of the 9th International Conference on Quality of Software (QSIC'09), Jeju, South Korea, August 2009. IEEE Computer Science.

7. T. Janssen, R. Abreu, and A.J.C. van Gemund, **Zoltar: A Spectrum-based Fault Localization Tool**. In Proceedings of the 1st International Workshop on Software Integration and Evolution @ Runtime (SINTER'09), pp. 23-29, Amsterdam, the Netherlands, August 2009. ACM Press.

8. R. Abreu, P. Zoeteweij, R. Golsteijn, and A.J.C. van Gemund, **A Practical Evaluation of Spectrum-based Fault Localization**. Journal of Systems & Software (JSS), Elsevier, 2009.

9. R. Abreu, P. Zoeteweij, and A.J.C. van Gemund, **A New Bayesian Approach to Multiple Intermittent Fault Diagnosis**. In Proceedings of the 21st

International Joint Conference on Artificial Intelligence (IJCAI'09), pp. 653-658, Pasadena, CA, USA, July 2009. AAAI Press.

10. R. Abreu and A.J.C. van Gemund, **A Low-Cost Approximate Minimal Hitting Set Algorithm and its Application to Model-Based Diagnosis**. In Proceedings of the 8th Symposium on Abstraction, Reformulation and Approximation (SARA'09), Lake Arrowhead, CA, USA, July 2009. AAAI Press.

11. R. Abreu, P. Zoeteweij, and A.J.C. van Gemund, **A Model-based Software Reasoning Approach to Software Debugging**. In Proceedings of the 22nd International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems (IEA-AIE'09), Tainan, Taiwan, June 2009. Studies in Computational Intelligence, vol. 214, pp. 233-239, Springer-Verlag.

12. R. Abreu, P. Zoeteweij, and A.J.C. van Gemund, **A Bayesian Approach to Diagnose Multiple Intermittent Faults**. In Proceedings of the 20th International Workshop on Principles of Diagnosis (DX'09), pp. 27-33, Stockholm, Sweden, June 2009.

13. R. Abreu and A.J.C. van Gemund, **Statistics-directed Minimal Hitting Set Algorithm**. In Proceedings of the 20th International Workshop on Principles of Diagnosis (DX'09), pp. 51-58, Stockholm, Sweden, June 2009.

14. R. Abreu, W. Mayer, M. Stumptner, and A.J.C. van Gemund, **Refining Spectrum-based Fault Localization Rankings**. In Proceedings of the 24th Annual ACM Symposium on Applied Computing (SAC'09) - Software Engineering Track, pp. 409–414, Honolulu, Hawai'i, USA, March 2009. ACM Press.

15. R. Abreu, P. Zoeteweij, and A.J.C. van Gemund, **A Dynamic Modeling Approach to Software Multiple-Fault Localization**. In Proceedings of the 19th International Workshop on Principles of Diagnosis (DX'08), pp. 7-14, Blue Mountains, NSW, Australia, September 2008.

16. W. Mayer, R. Abreu, M. Stumptner, and A.J.C. van Gemund, **Prioritizing Model-Based Debugging Diagnostic Reports**. In Proceedings of the 19th International Workshop on Principles of Diagnosis (DX'08), pp. 127-134, Blue Mountains, NSW, Australia, September 2008.

17. R. Abreu, P. Zoeteweij, and A.J.C. van Gemund, **An Observation-based Model for Fault Localization**. In Proceedings of the 6th Workshop on Dynamic Analysis (WODA'08), colocated with the International Symposium on Software Testing and Analysis (ISSTA'08), pp. 64-70, Seattle, WA, USA, July 2008. ACM Press.

18. P. Zoeteweij, J. Pietersma, R. Abreu, A. Feldman, and A.J.C. van Gemund, **Automated Fault Diagnosis in Embedded Systems**. In Proceedings of the 2nd IEEE International Conference on Secure Systems and Reliability Improvement (SSIRI'08), pp. 103-110, Yokohama, Japan, July 2008. IEEE Computer Society.

19. R. Abreu, A. González, P. Zoeteweij, and A.J.C. van Gemund, **On the Performance of Fault Screeners in Software Development and Deployment**. In Proceedings of the 3rd International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE'08), pp. 123–130, Funchal, Madeira, Portugal, May 2008. INSTICC Press.

20. R. Abreu, A. González, P. Zoeteweij, and A.J.C. van Gemund, **Automatic Software Fault Localization using Generic Program Invariants**. In Proceedings of the 23rd Annual ACM Symposium on Applied Computing (SAC'08) - Software Engineering Track, pp. 712–717, Fortaleza, Ceará, Brazil, March 2008. ACM Press.

21. P. Zoeteweij, J. Pietersma, R. Abreu, A. Feldman, and A.J.C. van Gemund, **Automated Fault Diagnosis in Embedded Software**. In Proceedings of the ESI / Bits & Chips Embedded Systems Conference, Eindhoven, The Netherlands, October 2007.

22. R. Abreu, P. Zoeteweij, and A.J.C. van Gemund, **On the Accuracy of Spectrum-based Fault Localization**. In Proceedings of the Testing: Academia and Industry Conference - Practice And Research Techniques (TAIC PART'07), pp. 89–98, Windsor, United Kingdom, September 2007. IEEE Computer Society.

23. P. Zoeteweij, R. Abreu, and A.J.C. van Gemund, **Software Fault Diagnosis**. Tutorial in the joint tutorial day of the TESTCOM / FATES and FORTE conferences, Tallinn, Estonia, June 2007.

24. P. Zoeteweij, R. Abreu, R. Golsteijn, and A.J.C. van Gemund, **Diagnosis of Embedded Software using Program Spectra**. In Proceedings of the 14th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS'07), pp. 213–218, Tucson, AZ, USA, March 2007. IEEE Computer Society.

25. P. Zoeteweij, R. Abreu, R. Golsteijn, and A.J.C. van Gemund, **Fault Diagnosis of Embedded Software using Program Spectra**. In Proceedings of the 3rd European Symposium on Verification and Validation of Software Systems (VVSS 2007), LaQuSo: Laboratory for Quality Software, Eindhoven, The Netherlands, March 2007, also available as Eindhoven Computing Science Technical Report: CS-Report 07-04

26. R. Abreu, P. Zoeteweij, and A.J.C. van Gemund, **An Evaluation of Similarity Coefficients for Software Fault Localization**. In Proceedings of the

12th International Symposium on Pacific Rim Dependable Computing (PRDC'06), pp. 39–46, Riverside, CA, USA, December 2006. IEEE Computer Society.

27. R. Abreu, P. Zoeteweij, R. Golsteijn, and A.J.C. van Gemund, **Automatic Fault Diagnosis in Embedded Software**. In Proceedings of 10th Philips Software Conference (PSC'06), Veldhoven, The Netherlands, November 2006.

28. R. Abreu, P. Zoeteweij, and A.J.C. van Gemund, **Program Spectra Analysis in Embedded Software: A Case Study**. In Proceedings of the 12th Annual Conference of the Advanced School for Computing and Imaging (ASCI'06), pp. 263 – 269, Lommel, Belgium, June 2006.