Cost Optimizations in Runtime Testing and Diagnosis

Cost Optimizations in Runtime Testing and Diagnosis

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof. ir. K.C.A.M. Luyben,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen
op maandag 19 september 2011 om 12:30 uur door

Alberto GONZÁLEZ SÁNCHEZ

Ingeniero en Informática - Universidad de Valladolid
geboren te Valladolid, Spanje.

Dit proefschrift is goedgekeurd door de promotor:
Prof. dr. ir. A.J.C. van Gemund

Copromotor Dr. phil. H.G. Gross

Samenstelling promotiecomissie:

Rector Magnificus	voorzitter
Prof. dr. ir. A.J.C. van Gemund	Delft University of Technology, promotor
Dr. phil. H.G. Gross	Delft University of Technology, copromotor
Prof. dr. F. Wotawa	Graz University of Technology, Austria
Prof. dr. C. Atkinson	University of Mannheim, Germany
Prof. dr. C. Witteveen	Delft University of Technology
Dr. ir. G.J. Tretmans	Radboud University, The Netherlands
Dr. R.F. Lima Maranhão de Abreu	University of Porto, Portugal
Prof. dr. K.G. Langendoen	Delft University of Technology, reservelid







This work was carried out as part of the POSEIDON 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.

Copyright © 2011 by Alberto González Sánchez

Cover: Wulingyuan National Forest Park, Zhangjiajie, China.
Printed and bound in The Netherlands by CPI Wörmann Print Service.

ISBN: 90–XXXX–XXX–X

To Papa Bear

While a book has got to be worthwhile from the point of view of the reader it's got to be worthwhile from the point of view of the writer as well.

– Terry Pratchett

Acknowledgments

The last four years have been an unforgettable exploration journey that is about to reach the end. Destination: the summit of 'Mount Doctorate'. Reaching the summit has its advantages, for example, selecting 'Treatment: Dr.' next time I book a flight, or replying 'Yes! Me!' next time someone screams 'Is there a doctor in the room?', although I'm not sure I'll be the kind of doctor they are interested in. Even though these little perks make the destination worthwhile, what makes it really worthwhile is the journey itself, and all the new experiences on the way. It may sound cliché, but there is definitely some truth to it. During four years I have discovered so many places, learnt about and experienced so many cultures, that I'm glad the book you have in your hands was required to contain *exclusively* my scientific findings. In my journey I have also met many people that have left a big imprint in me, and, directly or indirectly, helped to shape the content of this Thesis. The following lines are dedicated to all of them.

On every mountaineering journey you need a local guide. In my case my guide has been my supervisor and promotor prof.dr.ir. Arjan J.C. van Gemund. I will never be able to fully express my gratitude to Arjan. His dedication was total and went well beyond the duties of mere supervision. I definitely would not have reached this point without his advice, his support, his insistence, and his hard work. He not only pointed me to the right mountains to climb. When I didn't know how to continue, he showed me the way. When the load of my backpack felt too heavy, he carried it, while others would have just sat comfortably and watched me trip and fall over again and again. He has been a real *sherpa*.

In my journey I was accompanied by my co-supervisor and co-promotor Gerd Gross, who trusted me and offered me the position I have occupied the last four years, and gave me invaluable advice, and by Éric Piel, my officemate and friend in the moments of joy and sorrow, success and failure, happiness and unhappiness, during all these years.

I also don't want to forget about all the people in the tents at the base camp of the Software Engineering Research Group (SERG). The chiefs: Arie van Deursen, Rini van Solingen, Eelco Visser, Andy Zaidman, and Martin Pinzger. The warriors: Ali Mesbah, Bas Cornelissen (bring on the *lomo*!), Marius Marin, Cathal Boogerd, Cor-Paul Bezemer, and Rui Abreu (my 'research brother', my first friend in Delft, the one who welcomed me, and with whom I've gone all around the globe to the most fantastic places). The amazons: Michaela Greiler, Anja Guzzi and Felienne Hermans (a great Zwarte Piet!). And finally, the new recruits, in charge of perpetuating the good atmosphere: Cuiting *Den-Den* Chen (the smallest person, with the biggest heart), Tiago Espinha, and Daniele Romano (pay attention to this guy!).

Outside base camp, in the village further down the valley, my housemates in Koornmarkt 42: Aurélie, Justin and Ilhan, will always have a special place in my memories for all the crazy parties and trips, and especially Mathieu, who was always a good friend and was always there to listen, and with whom I have shared wonderful Belgian beers and conversation. Layla, my dear 'sister', for taking care of me better than anyone else in Delft. Inez, for translating my summary to perfect *vlaams*. David J., and Anne (and also Éric, Inez, Cathal and Layla, but those had their moment of glory already), for the craziest ski holiday ever. David H., for the coolest bike in my 'collection', and Théocléa, for taking me ice-skating at times of the day when sane people sleep. César, for all the fun in conferences, and the *soju!* Tânia, Gabriel and Emilie, for that night at *Paard van Troje*, that t-shirt, and so many other things. Egbert, a great guitar master and teacher (learning to play guitar may be the best spent money of my life). Qiaole, a friend with a cool head but a warm soul, someone who will never cease to amaze me, and from whom I learn something new every day. My fellows in the board of Promood, for more than two years of great experiences, and letting me be their Sinterklaas, twice.

A friend is a loving companion at all times. But a paranymph is not just a friend. Paranymphs are the friends you would trust to risk their own lives to defend yours, should the defense of your ideals go beyond verbal. Mine is Juan Carlos, the first friend I can remember, the most positive and optimistic, and the most loyal and understanding of them all. It is my honor to have him by my side for my defence.

Finalmente, agradecer a mi familia y amigos en España: a mi madre María, a mi padre Alfonso, a mi hermano Enrique (y al Gato). A mis amigos de siempre: Iñaki, Javi, Lucía, Padi, Irene, Pablo, Susana. A los compañeros de mesa y aventuras en la universidad: Juan, Jorge, Rubén, Luis, Diego, Lorena, Carmen, Graf y a todos los demás. Gracias por los años vividos, la (buena) educación recibida y todo lo que habéis tenido que soportar por mi culpa, y por mi ausencia. Me temo que tendremos que seguir echándonos de menos...

Alberto

Contents

Acknowledgements **vii**

1 Introduction **1**
1.1 Concepts and Definitions 2
1.2 Problem Statement 5
1.3 Contributions 8
1.4 Thesis Outline 10
1.5 Origin of Chapters 11

2 Using Testability Information in Software Fault Localization **13**
2.1 Fault Diagnosis 15
2.2 Estimating False Negative Rates 18
2.3 Experiment Setup 20
2.4 Results 22
2.5 Related Work 28
2.6 Summary 28

3 On-line Diagnostic Prioritization **29**
3.1 Test Case Prioritization 32
3.2 Prioritization and Diagnosis 34
3.3 Diagnostic Prioritization 36
3.4 Evaluation 43
3.5 Single Fault Results (IG) 45
3.6 Multiple Faults (SEQUOIA) 49

3.7 Related Work 50
3.8 Summary 53

4 Off-line Diagnostic Prioritization **55**
4.1 Raptor 56
4.2 Empirical Evaluation 61
4.3 Discussion 66
4.4 Related Work 70
4.5 Summary 71

5 Performance Modeling of Prioritization Algorithms **73**
5.1 Preliminaries 75
5.2 Performance Prediction 75
5.3 Effectiveness Prediction Using Ambiguity 82
5.4 Experiments 84
5.5 Related Work 87
5.6 Summary 90

6 Runtime Testability Measurement **91**
6.1 Runtime Testability 93
6.2 Runtime Testability Measurement 96
6.3 Theoretical Validation 99
6.4 Empirical Validation 103
6.5 Related Work 107
6.6 Summary 108

7 Optimization of Runtime Testability **109**
7.1 A Method to Improve Runtime Testability 110
7.2 Application Example 115
7.3 Near-optimal Optimization 121
7.4 Related Work 124
7.5 Summary 125

8 A Diagnostic View on Runtime Testability **127**
8.1 Efficiency: Taking Testing Cost into Account 129
8.2 Experiments 131
8.3 Related Work 135

8.4 Summary 136

9 Conclusions **137**
9.1 Summary of Contributions 138
9.2 Recommendations for Future Work 139

Bibliography **143**

Summary **153**

Samenvatting **155**

Curriculum Vitae **159**

List of Acronyms

ADDST Additional Statement coverage

APFD Average Percentage of Detected Faults

ART Adaptive Random Testing

CIG Component Interaction Graph

FEP Fault Exposing Potential Test Prioritization

FNR False Negative Rate

IG Information Gain

MSS Maritime Safety and Security

PIE Propagation-Infection-Execution

Raptor gReedy diAgnostic Prioritization by ambiguiTy grOup Reduction

RiTMO RuntIme Testability Measurement and Optimization

RTM Runtime Testability Metric

SC Similarity Coefficient

Sequoia SEQUencing fOr dIAgnosis

SFL Spectrum-based Fault Localization

SIR Software Infrastructure Repository

SOA Service Oriented Architecture

SoS System of Systems

List of Symbols

C	–	Set of components in the system.
M	–	Number of components of the system. $M = \|C\|$.
c_j	–	Component number j, $c_j \in C$.
T	–	Set of test cases representing the program's test suite.
N	–	Number of test cases in the test suite. $N = \|T\|$.
t_i	–	Test number i, $t_i \in T$.
C_i	–	Cost of test t_i.
A	–	Activity matrix of size $N \times M$.
a_{ij}	–	Involvement of component c_j in test t_i.
O	–	Test outcome vector.
o_i	–	Outcome of test t_i.
M_f	–	Number of faulty components in the system.
ρ	–	Coverage density of matrix A.
C_d	–	Diagnostic effort.
C_t	–	Testing effort.
D	–	Diagnostic report (list of diagnostic candidates).
d_k	–	Diagnostic candidate, $d_k \in D$.
$\Pr(d_k)$	–	Probability of candidate d_k being the true diagnosis.
d_*	–	True diagnosis, $d_* \in D$.
p_j	–	Prior probability of component c_j being faulty.
h_j	–	False negative rate of component c_j.
$n_{pq}(j)$	–	Number of times component c_j is involved ($p = 1$) or not ($p = 0$) in passed ($q = 0$) or failed ($q = 1$) runs.
l_j	–	Likelihood of component c_j being faulty.
$H(D)$	–	Information entropy of diagnosis D.
AG	–	Set of ambiguity groups generated by an activity matrix.
g_l	–	Ambiguity group, $g_l \in AG$.
$G(AG)$	–	Average diagnostic ambiguity of AG.
S	–	Diagnostic effectiveness.
V	–	Set of vertices in the CIG graph.
V_0	–	Set of input operations. $V_0 \subseteq V$.
E	–	Edges in the CIG graph.
$c(v_i)$	–	Individual preparation cost of vertex v_i.
$C(V_i)$	–	Total preparation cost of vertex v_i.

1 Introduction

The commission of the European communities has recently pushed for the establishment of a European Network for Maritime Surveillance [E. U. Commission, Maritime Affairs, 2007]. Such a network will provide safe and secure usage of the seas around Europe, integrated and coordinated maritime planning, research, climate control, and sustainable development. This network will also require the cooperation of the Member States' security agencies, and an efficient usage and integration of not only existing navigation, monitoring and tracking systems, but also the systems in the respective operations and control centers.

This kind of large-scale heterogeneous system, in which the components have an operational entity of their own, and usually a managerial entity as well, is known as System of Systems (SoS) [Maier, 1998]. SoS present considerable engineering challenges that were acknowledged by the Dutch Embedded Systems Institute and Thales Nederland. They set up a research project [Embedded Systems Institute, 2007], committed to devising engineering best practices for developing, integrating, deploying, and continuous quality assurance of such Maritime Safety and Security (MSS) systems, which provide much of the context of this thesis work.

Devising appropriate quality assurance strategies are amongst the most obvious challenges in building and evolving large-scale MSS SoS, given the large number of different systems contributing to them. MSS SoS evolve dynamically during operation time (*runtime evolution*). Systems can join or leave the SoS, meaning that offered services may vary in terms of functionality, as well as quality. When a sub-system joins or leaves the SoS, the other sub-systems may have to be reconfigured to take advantage of new services and improved quality of service, or they may have to be notified that services are degraded. This process should be mostly seamless for the system operators and should be executed within a short time, without any major disruption of the rest of the SoS.

By their very nature, SoS are large-scale systems, with a large number of components contained in the sub-systems. After each runtime evolution, the quality assurance of the integrated SoS has to be verified again. It is therefore necessary to devise an appropriate verification strategy that not only achieves this goal, but also *minimizes* the cost of checking after each modification. Re-verification must be as little disruptive as possible for the running configuration and the latency be-

tween the moment a reconfiguration is requested and the moment it is accepted and deployed must be minimal.

In the context of this thesis, the SoS will be formed by software components whose quality assurance will be performed by means of runtime testing and diagnosis. Software testing and diagnosis (debugging) is a time-consuming but rather important task for improving software reliability. Hence, testing and debugging are an integral part of the quality assurance process of these systems.

The ultimate goal of testing is to prove the absence of faults for all possible inputs. Since exhaustive testing is hardly ever possible, testing typically aims at detecting the presence of faults, by trying to produce a failure (the external manifestation of a fault). The longer the program does not produce any failures, the more certain testers are that the program contains no faults. Literature comprises a myriad of techniques to efficiently produce failures, i.e., by improving coverage and/or reducing the testing cost [Jiang et al., 2009a, Li et al., 2007, Rothermel et al., 2001, Smith and Kapfhammer, 2009]. Once failures have been detected, the faults that produced them have to be localized. Fault localization is a whole research area on its own that has recently attracted great interest. The goal of (automated) fault localization is to reduce the effort that the software developers have to invest in manually localizing the fault.

Unfortunately for developers, it has recently been shown that effective fault detection and effective fault localization are contradictory goals [Jiang et al., 2009b, Yu et al., 2008]. In particular, it has been shown that focusing exclusively on fault detection has a detrimental effect on fault localization, canceling the savings in testing cost by an increase in diagnostic cost. In this thesis we explore the causes for this cost increase, and propose a shift in the goal of testing, focusing on techniques that achieve much better fault localization performance, while maintaining fault detection performance.

Furthermore, since runtime tests will be carried out in the production system itself, we dedicate a part of this thesis to exploring further implications of runtime testing and diagnosis. In particular, the system's components have to be prepared for the execution of tests while also remaining functional, i.e., to be made runtime testable. Runtime testability requires an investment in preparation cost to ensure the tests maintain their fault detection and localization performance. In this thesis we will focus on the preparation costs required for runtime testability, and their trade-off with fault detection and localization costs.

1.1 Concepts and Definitions

Throughout this thesis, we use the following software reliability terminology [Avizienis 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 systems, where faults are bugs in one of the components of the system (typically in the program code). In general, these systems transform input data into output data, 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. Note that an error does not lead to a failure if it does not propagate to the output.

Runtime Test Execution

Testing is the most commonly used means for the quality assurance of software systems. The goal of testing is to detect the presence of faults (e.g., programming errors, specification mismatches) in the system under test. The presence of faults is demonstrated by the occurrence of failures during testing.

The testing process of high-availability and mission-critical systems has been traditionally performed either in a separate, identical copy of the system, or by taking the system offline. Due to the huge size of the SoS, the limited access to the system's code or executables, the need to avoid disruption to the SoS, or the fact that some components use resources that cannot be duplicated, (a) testing will have to be executed concurrently with the working configuration, and (b) some component instances will be shared between the tested and the working configurations, as mentioned in [Brenner et al., 2007a, Dumitras et al., 2009, González et al., 2008b]. The main reason is that in a SoS or Service Oriented Architecture (SOA) context many components are external to the organization. Therefore duplicating the system's components and their context is too complex, if possible at all. Furthermore, taking the system off-line is usually not possible given the high availability requirements of such systems.

Runtime testing is emerging as the solution for the validation of SoS and SOA systems. Runtime testing is an on-line testing method that has to be carried out in the final execution environment of the system, when the system or some of its components are already operational. Runtime testing has been proposed for, e.g., component-based systems [Brenner et al., 2007b, González et al., 2008a, Suliman et al., 2006], web services [Brenner et al., 2007a], and for Java applications [Murpy et al., 2009].

Figure 1.1 depicts both an off-line and an on-line testing process. The system on the left-hand side is being tested off-line. The modification is performed first in a separate instance of the system, where it is tested. If the testing succeeds, the modification is applied to the production instance of the system. On the right-hand side, the system is being runtime tested. The modification (in this case, the addition of a new sub-system) is applied to the running system, and tested in a way that does not disrupt the normal system operation, and without using a separate instance for testing.

Runtime tests will cause an overhead since they will be executed on the running

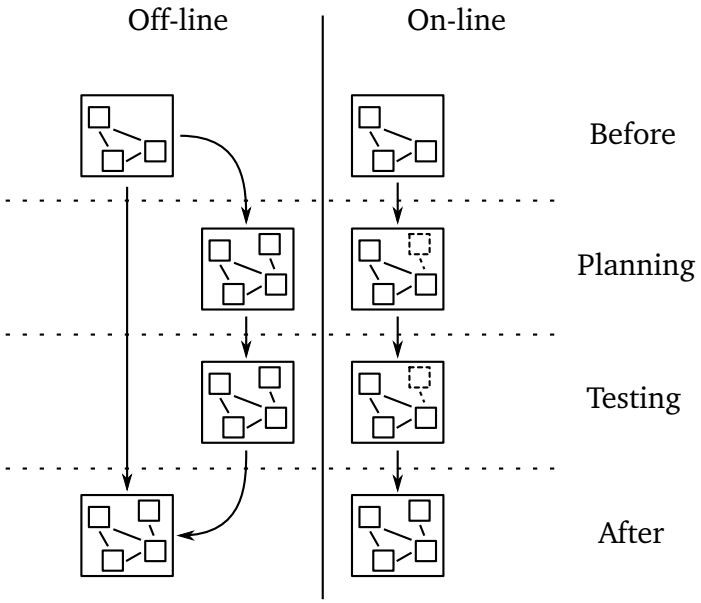


Figure 1.1: Non-runtime (off-line) vs. runtime testing (on-line)

system. Therefore, the number and duration of runtime tests have to be minimized while still ensuring the quality of the modified system. Consequently, minimizing testing cost is more critical than in an off-line testing scenario like traditional regression testing.

Runtime Fault Diagnosis

Testing will often uncover failures. This signifies the presence of a fault, that has to be localized in order to be isolated and repaired. *Runtime fault diagnosis* is a diagnostic process to identify the root cause of the failures that were observed during the runtime testing process. A diagnosis lists the system's components which, when faulty, explain the observations in terms of tests that pass or fail. Diagnostic accuracy is a critical success factor in the cycle of testing, diagnosing, acting/recovering, and repairing. A perfect diagnosis only contains the real faults. However, due to, e.g., the nature of tests, a small number of observations, and/or the lack of detailed specifications, a diagnosis usually contains multiple explanations. The diagnosis is typically returned to the developer as a list of suspect components, ranked by the probability of being faulty. Developers have to traverse that list, manually inspecting each of the suspects. Diagnostic cost comprises the cost of obtaining the list of explanations (*algorithmic cost*), and the effort spent while traversing the list (*inspection cost*). The diagnostic inspection cost depends on how many false positives are present in this list. In this thesis, algorithmic cost can be ignored compared to inspection cost, thus we will use the terms diagnostic inspection cost and diagnostic cost interchangeably.

To lower diagnostic cost, the number of suspects can be reduced by obtaining additional information to eliminate suspects. This is typically done by executing further tests, ideally until an unambiguous diagnosis is reached. Since these tests will again cause an overhead in the running system, their impact has to be traded off against diagnostic utility, such that the overall testing and diagnosis cost is minimized.

Runtime Test Preparation

When tests are performed on a running system, test operations can interfere with the state of the running system or its environment causing unacceptable consequences. In the worst case, runtime tests will affect the system's environment in critical ways that are difficult to control or impossible to recover from, e.g., firing a missile while testing part of a combat system.

Test isolation techniques are applied by test engineers in order to prevent test operations from interfering with the state or environment of the system, and to avoid that the production state and interactions of the system influence the test results. This property of the system is known as *runtime testability* [González et al., 2009a], i.e., the degree to which a system or a component facilitates runtime testing without being extensively affected.

The implementation of isolation measures can be represented as a cost. In order to test the system, an additional expense in the test *preparation* phase is required, in terms of the cost of isolation techniques. Without this investment in preparation costs, there will be many parts of the system that developers will not be able to test, severely affecting fault detection and fault localization.

1.2 Problem Statement

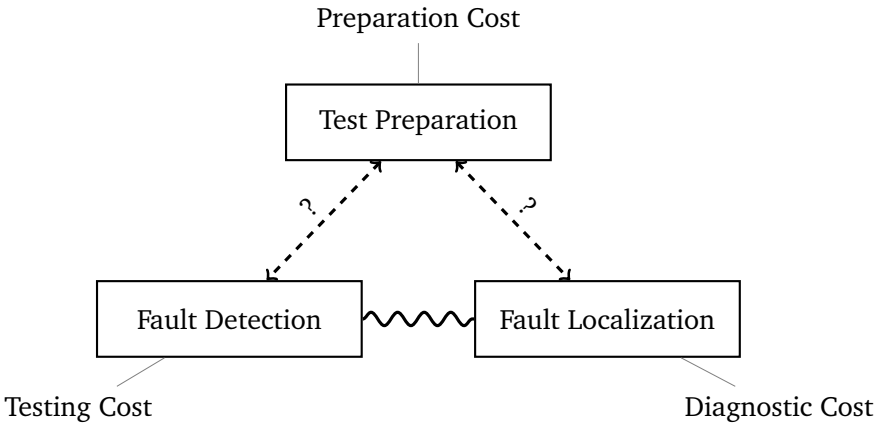


Figure 1.2: Goals and associated costs

In the previous section, three processes have been identified that are critical to the quality assurance of SoS: (1) test preparation, (2) fault detection, and (3) fault localization. These three processes are represented in Figure 1.2. Each of these processes has a cost attached to it:

1. **Preparation cost:** these costs are required to prepare the system for the execution of runtime tests, i.e., to achieve runtime testability. They are associated with the preparation actions required to implement test isolation measures in each of the system's components.

2. **Testing cost:** these are the costs related directly to the execution of the tests. These costs come from detecting the presence of faults, and/or providing more information to the diagnostic algorithm.

3. **Diagnostic cost:** these are the costs associated with the localization of faults in the system, once the diagnosis is not refined any further by additional tests. They are related to the manual work that developers have to perform to inspect each of the suspects until the faults are found.

Of these three costs, preparation costs are specific to runtime testing, whereas testing and diagnostic costs are also present in more conventional software development contexts, such as the more traditional regression testing approach.

The overall cost of the system's quality assurance by runtime testing and diagnosis, which includes the already mentioned costs, has to be minimized, while maximizing the confidence in the integrated system. However, these goals may be contradictory, which means that optimal trade-offs have to be identified. From previous literature, we already know that fault detection and fault localization are mutually antagonistic goals (i.e., a good fault detection capability will cause poor fault localization performance). This is represented in Figure 1.2 by a snake line between the two. Runtime testing is a novel area of research, and our main task will be to determine the relationship between the preparation costs of a system, and the fault detection and localization goals.

The main research problem, subject of this thesis, can be formulated as follows.

> *What are the trade-offs between (1) fault detection, (2) fault localization and (3) test preparation, in terms of their associated costs and benefits?*

These trade-offs will be elaborated in more detail in the following subsections.

Fault Detection vs. Fault Localization

Testing to detect faults, and specifically test prioritization techniques, are founded on the reasoning that the sooner failures are found, the sooner fault localization

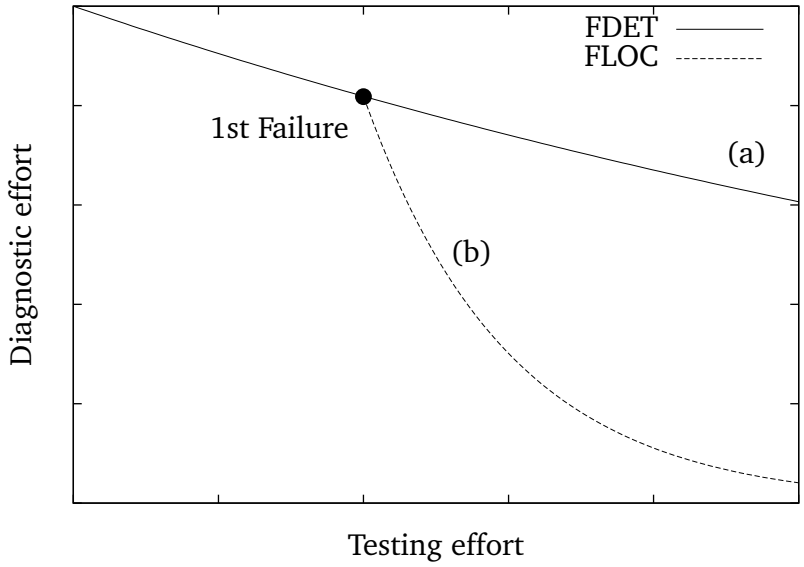


Figure 1.3: Test execution cost prioritization scenarios

(e.g., debugging) can commence. In this thesis, we will explore the trade-off between fault detection (i.e., testing cost) and fault localization (i.e., diagnostic cost). We will assume that the tests used to detect a fault, and to locate it, belong to the same test pool. Since existing literature suggests that the current prioritization approaches for fault detection have a very negative effect on fault localization [Yu et al., 2008, Jiang et al., 2009b], we will focus our efforts on devising new prioritization algorithms that maximize the reduction of diagnostic cost per unit of test cost.

Figure 1.3 illustrates the situation, showing the trade off between testing effort and diagnostic effort, in two possible scenarios. Scenario (a) represents a situation in which a prioritization heuristic for fault detection (FDET) is used during testing. The diagnostic information each test provides is very small and the final diagnostic effort when we run out of test time is very high (conversely, refining the diagnosis will require a large diagnostic test effort). Scenario (b), on the other hand, represents the improvement in the evolution of diagnostic effort if, after the first failure, we switch to a hypothetic diagnostic test prioritization strategy (FLOC), that selects test cases in a way such that the reduction of diagnostic cost is optimal.

Test Preparation vs. Fault Detection and Localization

A practical approach to minimizing preparation cost is to consider a prioritization of test preparation (isolation measures), in a similar way to test prioritization. However, for an optimal prioritization, we need to consider the trade-off between test preparation and fault detection. An insufficient investment in prepa-

ration cost will affect fault detection since some parts of the system will have to be left untested. Furthermore, since not all tests have similar fault detection capabilities, the utility of certain preparation costs will be linked to the tests they enable. Our research problem consists in finding ways to measure the utility of the different preparation costs, in terms of their fault detection return.

Given the antagonistic relationship between fault detection and localization, we need explore the diagnostic impact of preparation costs optimized for fault detection, and whether a better trade-off between test preparation and fault localization exists, by including diagnostic cost into the equation.

1.3 Contributions

In this section we will outline the seven main contributions of this thesis. Four contributions correspond to minimizing testing and diagnosis costs, and three correspond to minimizing preparation costs.

Testing and Diagnosis

Our contributions are motivated by the fact that most literature in the field is devoted to enhancing fault detection without any regard for fault localization. Our contributions focus on the following aspects:

1. Coincidental Correctness

An important performance factor for automatic fault localization algorithms is exploiting knowledge of the probability that test cases may not fail, even when involving a faulty component (false negatives or coincidental correctness). Many techniques use mutation testing to estimate the rate at which false negatives will be produced, which requires a relatively large testing cost. In this thesis, we evaluate to what extent prior knowledge about the false negative rates is useful.

2. Information Gain

The key for an effective diagnostic cost reduction during testing is diagnostic *information gain*, i.e., the reduction of the set of suspect components. Unfortunately, computing information gain is exponentially complex in the number of components of the system when multiple faults can be present. We propose a number of heuristic approximations that reduce the information gain estimation while maintaining good accuracy. We integrate these approximations in a prioritization algorithm called Sequoia (*SEQUencing fOr dIAgnosis*) that employs information gain as heuristic.

3. Ambiguity Reduction

The above information gain heuristic can be impaired by both the characteristics of the test suite, and by errors in the estimation of the heuristic's input parameters, especially the estimations of false negative rates of test cases. We study a low-complexity test prioritization heuristic which does not require any additional input parameters. We integrate this heuristic in a second test prioritization algorithm, dubbed RAPTOR (*gReedy diAgnostic Prioritization by ambiguiTy grOup Reduction*)

4. Test Suite Analysis

The results of the above prioritization algorithms do not always provide a large improvement over existing approaches. We present an analytical model for the prediction of the diagnostic effectiveness of a test suite, based on purely structural characteristics of the test coverage matrix. The model is not only useful for *predicting* the diagnostic effectiveness of test matrices, but also for selecting what prioritization algorithm to use, and to steer the *generation* of new test suites.

Test Preparation

Unlike test prioritization, with its extensive body of knowledge, the area of runtime testability and preparation prioritization is a novel area of research.

5. Runtime Testability

In order to optimize the preparation costs of runtime testing, we need an indicator of the return of investment that we obtain, which we call *runtime testability*. We relate runtime testability to the fault detection capability of a test suite in terms of the maximum coverage that can be reached by runtime tests. We perform an empirical study of the relation between testability and fault detection performance using two case studies, as well as a validation of the theoretical properties of runtime testability as a metric, called Runtime Testability Metric (RTM).

6. Runtime Testability Optimization

We present RiTMO (*RuntIme Testability Measurement and Optimization*), a method to calculate the RTM of a system, and to prioritize the available test isolation operations to be implemented. Since prioritization corresponds to a Binary Integer Programming problem, we develop two heuristic methods based on using greedy search.

7. Preparation Costs vs. Diagnosis Costs

In the prioritization of test preparations, we employ RTM, which expresses fault detection performance. A question left unanswered is whether good runtime testability also leads to good runtime fault localization quality. We investigate the relationship between testability and fault localization quality, and introduce a second measurement that captures fault localization better than RTM.

Our contributions show there is ample room for the improvement in the quality assurance of runtime-evolving systems, when considering the trade-offs between the costs of test preparation, test execution and diagnosis, and the return obtained in terms of fault detection and fault localization.

1.4 Thesis Outline

The seven contributions described in the previous section are presented in seven chapters. The relationship with each of the trade-offs investigated in this thesis is shown in Figure 1.4. In the first four chapters the relationship and trade-offs between testing cost and diagnostic cost are explored. The next two chapters of this thesis are devoted to the trade-off between the preparation costs of runtime testing, and the returns in fault detection. We dedicate the last chapter to the relationship between test preparation and the fault localization capabilities of runtime tests.

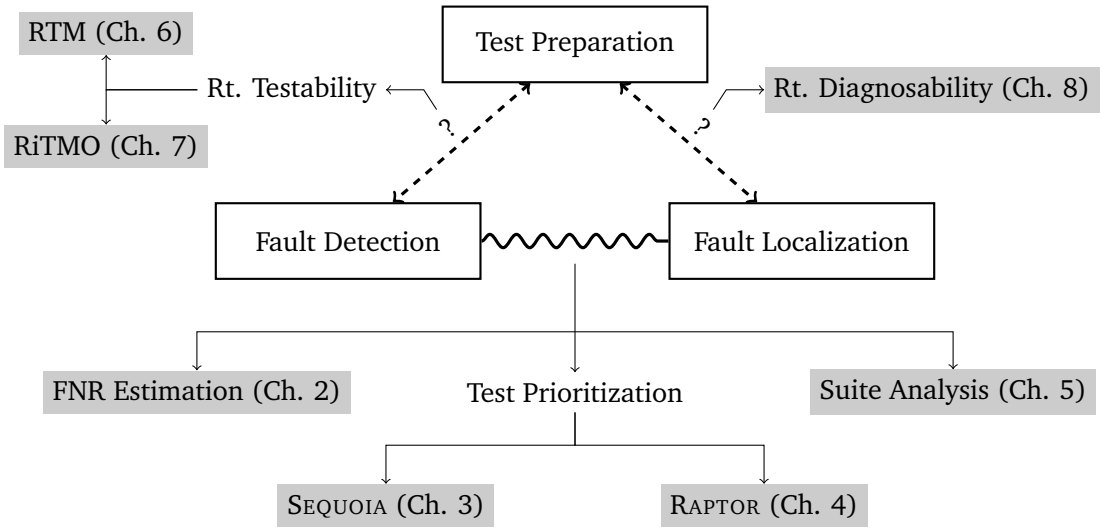


Figure 1.4: Goals and chapters

Chapter 2 covers the usage of prior false negative rate information during testing. Chapter 3 presents the concept of information gain, and our test prioritization algorithm SEQUOIA. Chapter 4 introduces the RAPTOR prioritization algorithm.

Chapter 5 contains a detailed analysis of the results observed for SEQUOIA and RAPTOR in the two previous chapters, culminating in a model that can be used to predict the diagnostic behavior of prioritization algorithms. Chapter 6 describes the concepts of runtime testing and testability, and introduces and validates RTM as a metric. Chapter 7 introduces the RiTMO method for runtime testability improvement and a sub-optimal preparation cost prioritization algorithm. Chapter 8 presents a different perspective on runtime test preparation, studying the trade-off with diagnosis cost. Chapter 9 presents our final conclusions and discusses directions for future work.

1.5 Origin of Chapters

Each of the chapters in this thesis has been published in at least one peer-reviewed publication. Most of the publications have been co-authored with Piel, Abreu, Gross, and Van Gemund. The following list gives an overview of these publications:

Chapter 2 has been published in the proceedings of the *26th Symposium on Applied Computing (SAC'11)* [Gonzalez-Sanchez et al., 2011a].

Chapter 3 is based on material published in the journal *Software: Practice and Experience* [Gonzalez-Sanchez et al., 2011e]. This article is an extension of earlier work appearing in the *10^{th} International Conference on Quality Software (QSIC'10)* [Gonzalez-Sanchez et al., 2010c]. The multiple-fault algorithm SEQUOIA has been published at the *21^{st} International Workshop on the Principles of Diagnosis (DX'10)* [Gonzalez-Sanchez et al., 2010a], and the *25^{th} Conference on Artificial Intelligence (AAAI'11)* [Gonzalez-Sanchez et al., 2011c].

Chapter 4 contains our work accepted for publication in the proceedings of the *26th International Conference on Automated Software Engineering (ASE'11)* [Gonzalez-Sanchez et al., 2011b].

Chapter 5 contains material published in the journal *Software: Practice and Experience* [Gonzalez-Sanchez et al., 2011e], the *21^{st} International Workshop on the Principles of Diagnosis (DX'10)* [Gonzalez-Sanchez et al., 2010a], and at the *Workshop on Testing and Debugging (TEBUG'11)* [Gonzalez-Sanchez et al., 2011d].

Chapter 6 comprises our findings published at the *4^{th} International Workshop on Advances on Model-based Testing (AMOST'09)*, the *34^{th} Annual IEEE Computer Software and Applications (COMPSAC'10)* [Gonzalez-Sanchez et al., 2010b], and the *2^{nd} International Conference on Advances in System Testing and Validation Lifecycle (VALID'10)* [Gonzalez-Sanchez et al., 2010d], where it obtained the best paper award.

Chapter 7 contains material published at the *9th International Conference on Quality Software (QSIC'09)* [Gonzalez-Sanchez et al., 2009], and the *34th Annual IEEE Computer Software and Applications (COMPSAC'10)* [Gonzalez-Sanchez et al., 2010b].

Chapter 8 is based on results published at the proceedings of the *Workshop on Testing and Debugging (TEBUG'11)* [Gonzalez-Sanchez et al., 2011f].

2

Using Testability Information in Software Fault Localization

An essential ingredient of effective and efficient fault localization is knowledge about the false negative rate of tests, which is related to the rate at which defective statements of a program will exhibit failures. In current fault localization processes, false negative rates are either ignored completely, or merely estimated a posteriori as part of the diagnosis.

In this chapter, we study the reduction in testing and diagnosis effort when false negative rates are known a priori. We deduce this information from testability, following the Propagation-Infection-Execution approach. Experiments with real programs suggest significant improvement in the diagnosis process, both in the single and the multiple-fault cases.

2.1 Fault Diagnosis **15**
2.2 Estimating False Negative Rates **18**
2.3 Experiment Setup **20**
2.4 Results **22**
2.5 Related Work **28**
2.6 Summary **28**

Testing is the most commonly used method for detecting the presence of faults in software. However, once the presence of a fault has been detected (by means of a failing test), its precise location has to be determined. Fault localization denotes the process of finding the root-cause of failures through diagnosis to support debugging. Diagnostic accuracy is a critical success factor in the cycle of testing, diagnosing, acting/recovering, and repairing. Typically multiple diagnoses are possible, and further tests are executed to narrow down the set of possibilities, ideally until a perfectly accurate diagnosis is reached.

In software, not all test inputs will produce a failure, because of the different paths taken in a test, introducing *false negatives* or *coincidental correctness*. Ignoring false negatives, as done in statistical diagnosis approaches [Abreu et al., 2007, Jones et al., 2002], results in degraded diagnostic performance [Abreu et al.,

2009c]. An alternative approach is to approximate the false negative rate (FNR) during a Bayesian diagnosis process [Abreu et al., 2009a, De Kleer, 2009]. However, the absence of prior knowledge about the false negative rates of faulty statements leads to a loss of diagnostic accuracy. The diagnosis algorithm must approximate the FNR while it computes the diagnosis, and that, again, depends on the FNR. The net result is that a large number of tests is required to obtain an accurate estimate of the FNR, thereby reducing the rate at which the diagnosis converges, and thus, increasing both testing effort and diagnostic effort.

In software fault localization, the FNR depends on the testability of the faulty statements, and can be determined *a priori*, using testability quantification techniques [Rothermel et al., 2001, Voas, 1992]. By providing testability-based information on the false negative rate, the estimation problem can be detached from the diagnosis, leading to significant performance gains in the fault localization process.

In this chapter we evaluate to what extent testability information can lead to performance gains in the fault localization process, both in single-fault and multiple-fault settings. We study and assess the testability of the statements in a system, and use that information as input to a Bayesian diagnosis algorithm. Existing techniques for testability quantification are performed in a single-fault setting, assuming that multiple-faults will cause failures as independent events, a condition that may not hold in software. The extent to which testability results can be used for diagnosing systems where multiple faults are present simultaneously remained unknown, and is also evaluated in this chapter.

In particular, the chapter makes the following contributions.

1. We compare the performance of Bayesian diagnosis with prior FNR information with (1) Bayesian diagnosis without prior knowledge of the FNR, and (2) with similarity coefficients. We perform our experiments in both the single and multiple-fault cases.

2. We evaluate to which extent testability estimations can be used to accurately predict the rate of false negatives when testing a system with multiple faults, i.e., whether failures of different faults are independent.

Our results show that the gains achieved by introducing prior knowledge about false negative rates are significant. For real programs containing real faults, we observed up to a 80% diagnostic effort reduction for the same testing effort in the single fault case, and a 62% for multiple faults.

The chapter is organized as follows. Section 2.1 introduces the main concepts of diagnosis and the main diagnostic techniques. Section 2.2 presents two different solutions for estimating the FNR of a system under test. The systems used in our experiments are presented in Section 2.3. An empirical validation is performed in Section 2.4 for single faults and for multiple faults. Section 2.5 describes related

	Program: Character Counter	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8
c_1	`main() {`	1	1	1	1	1	1	1	1
c_2	`int let, dig, other, c;`	1	1	1	1	1	1	1	1
c_3	`let = dig = other = 0;`	1	1	1	1	1	1	1	1
c_4	`while(c = getchar()) {`	1	1	1	1	1	1	1	1
c_5	`if ('A'<=c && 'Z'>=c)`	1	1	1	1	1	1	1	0
c_6	`let += 2; /* FAULT */`	1	0	1	1	0	0	1	0
c_7	`else if ('a'<=c && 'z'>=c)`	1	1	1	1	1	1	1	0
c_8	`let += 1;`	1	0	0	0	1	0	1	0
c_9	`else if ('0'<=c && '9'>=c)`	1	1	1	1	1	1	0	0
c_{10}	`dig += 1;`	1	1	1	1	0	0	0	0
c_{11}	`else if (isprint(c))`	0	0	0	0	1	1	0	0
c_{12}	`other += 1;}`	0	0	0	0	1	0	0	0
c_{13}	`printf("%d %d %d\n", let, dig, other);}`	1	1	1	1	1	1	1	1
	Test case outcomes	1	0	1	1	0	0	1	0

Inputs for test cases: t_1: `"A184"`, t_2: `"1337"`, t_3: `"T1"`, t_4: `"UB40"`, t_5: `"#lolcats"`, t_6: `"\0x10\0x05\0x00"`, t_7: `"Alberto"`, t_8: `""`

Table 2.1: Faulty program and fault diagnosis inputs.

work in the area of fault diagnosis. Section 2.6 concludes the chapter and presents our future research directions.

2.1 Fault Diagnosis

Consider the example system presented in Table 2.1, which shows an example faulty program [Jiang et al., 2009a]. The objective of fault diagnosis is to pinpoint the precise location of the fault (or faults) in the program by observing the program's behavior given a number of tests.

The following inputs are usually involved in automated diagnosis:

- A finite set $C = \{c_1, c_2, \ldots, c_j, \ldots, c_M\}$ of M components (e.g., source code statements, function points, classes, etc.) which are potentially faulty. We will denote the number of faults in the system as M_f.
- A finite set $T = \{t_1, t_2, \ldots, t_i, \ldots, t_N\}$ of N tests with binary outcomes $O = (o_1, o_2, \ldots, o_i, \ldots, o_N)$, where $o_i = 1$ if test t_i failed, and $o_i = 0$ otherwise.
- A $N \times M$ coverage matrix, $A = [a_{ij}]$, where $a_{ij} = 1$ if test t_i involves component c_j, and 0 otherwise.

For the remainder of this chapter, we will consider statement level as our level of granularity. Hence, the output of fault localization, is a *diagnosis*, i.e., a list or a

statement ranking of statement indices ordered by the likelihoods of each of them being faulty.

Diagnostic Effort

Diagnostic performance is accounted in terms of a metric C_d that measures the percentage of excess *effort* incurred in finding all statements at fault [Abreu et al., 2009a]. C_d measures *wasted effort*, independent of the number of faults M_f in the program, to enable an unbiased evaluation of the effect of M_f on C_d. Thus, regardless of M_f, $C_d = 0$ represents an ideal diagnosis technique (all M_f faulty statements on top of the ranking, no effort wasted on testing other statements to find they are not faulty), while $C_d = M - M_f$ represents the worst case (testing all $M - M_f$ healthy statements until arriving at the M_f faulty ones).

For example, consider the ranking $\langle c_{12}, c_5, c_6, \ldots \rangle$, while c_6 is the actual fault. In this case, the diagnosis leads the developer to inspect c_{12} and c_5 first. As both statements are healthy, C_d is increased with 2. The next statement to be inspected is c_6. As it is faulty, no more effort is wasted, and consequently, $C_d = 2$.

Statistical Fault Diagnosis

Statistical fault diagnosis is a well-known approach to fault diagnosis that originates from the Software Engineering domain. Its most known variant is Spectrum-based Fault Localization (SFL) [Abreu et al., 2007, Jones et al., 2002]. In SFL, the fault likelihood l_j is quantified in terms of *similarity coefficients*. A Similarity Coefficient (SC) measures the statistical similarity between statement c_j's test coverage $(a_{1j}, \ldots, a_{Nj})$ and the observed test outcomes, $(o_1, \ldots, o_N)$. Similarity is computed by means of four counters $n_{pq}(j)$ that count the number of times a_{ij} and o_i form the combinations $(0,0), \ldots, (1,1)$, respectively, i.e.,

$$n_{pq}(j) = |\{i \mid a_{ij} = p \wedge o_i = q\}| \;\; p, q \in \{0, 1\} \tag{2.1}$$

For instance, $n_{10}(j)$ and $n_{11}(j)$ are the number of tests in which c_j is executed, and which passed or failed, respectively. The four counters sum up to the number of tests N. Two of the most known SCs are the Tarantula [Jones et al., 2002], and Ochiai [Abreu et al., 2007] similarity coefficients, given by

$$\text{Tarantula:} \quad l_j = \frac{\dfrac{n_{11(j)}}{n_{11(j)} + n_{01(j)}}}{\dfrac{n_{11(j)}}{n_{11(j)} + n_{01(j)}} + \dfrac{n_{10(j)}}{n_{10(j)} + n_{00(j)}}} \tag{2.2}$$

$$\text{Ochiai:} \quad l_j = \frac{n_{11(j)}}{\sqrt{(n_{11(j)} + n_{01(j)}) \cdot (n_{11(j)} + n_{10(j)})}} \tag{2.3}$$

In our example system, after executing all the tests in the test suite, the counters for c_6 are $n_{11}(6) = 4$, $n_{10}(6) = 0$, $n_{01}(6) = 0$, $n_{00}(6) = 4$. Its likelihood being the faulty one according to Ochiai is $l_6 = 1.0$. The remaining statements have lower likelihoods, as they all have $n_{10}(j) > 0$ or $n_{01}(j) > 0$. For example, for c_5, $n_{11}(5) = 4$, $n_{10}(5) = 3$, $n_{01}(5) = 0$, $n_{00}(5) = 1$, and its Ochiai similarity is $l_5 = 0.71$.

Despite their lower diagnostic accuracy [Abreu et al., 2009a], similarity coefficients have gained much interest. A great advantage of similarity coefficients is their ultra-low computational complexity compared to probabilistic approaches and their independence from prior FNR knowledge. Therefore, we will include similarity coefficients in our evaluation.

Bayesian Fault Diagnosis

Bayesian fault diagnosis aims at obtaining a set of fault candidates $D = \langle d_1, \ldots, d_k \rangle$. Each candidate d_k is a subset of the statements which, at fault, explain the observed failures. For instance, $d = \{c_1, c_3, c_4\}$ indicates that c_1 **and** c_3 **and** c_4 are faulty, and no other statement. These diagnostic candidates are again sorted by the probability of being the correct diagnosis, $\Pr(d_k)$.

Initially, the probability of each candidate is

$$\Pr(d_k) = \prod_{c_j \in d_k} p_j \cdot \prod_{c_{j'} \notin d_k} (1 - p_{j'}) \tag{2.4}$$

After test case t_i is executed, the probability of each candidate $d_k \in D$ is updated depending on the outcome o_i of the test, following Bayes' rule:

$$\Pr(d_k|o_i) = \frac{\Pr(o_i|d_k)}{\Pr(o_i)} \cdot \Pr(d_k) \tag{2.5}$$

which is repeated recursively for each subsequent test.

In this equation, $\Pr(d_k)$ represents the prior probability of candidate d_k before the test is executed. $\Pr(o_i)$ is a normalization value that represents the residual probability of the observed outcome, given by

$$\Pr(o_i) = \sum_{d_k \in D} \Pr(o_i|d_k) \cdot \Pr(d_k) \tag{2.6}$$

$\Pr(o_i|d_k)$ represents the probability of the observed outcome o_i produced by a test t_i, if that candidate d_k is the actual diagnosis. It is defined as

$$\Pr(o_i|d_k) = \begin{cases} 0 & \text{if } o_i \text{ and } d_k \text{ are inconsistent} \\ 1 & \text{if } o_i \text{ is unique to } d_k \\ \varepsilon(d_k) & \text{otherwise} \end{cases} \tag{2.7}$$

The value of ε is related directly to the false negative rate of the tests that cover the statements in d_k. Multiple policies for ε exist. In the next section we will describe the most common one.

2.2 Estimating False Negative Rates

Correct information on the FNR of tests is crucial to a correct diagnosis, and its estimation has been a recurring research topic. The FNR of tests that cover the statements in a diagnosis candidate d_k depends on the FNR of every individual statement $c_j \in d_k$. The FNR of individual statements is modeled by a probability value $h_j \in [0, 1]$ (h for *health*). $h_j = 0$ means that when tested, a faulty statement will always produce a failure. $h_j = 1$ means that when tested, a faulty statement will never produce a failure.

The most common ε policy assumes that faulty statements will cause failures independently, e.g., a failure in a system with 2 faults will be produced when either of the faults causes a failure. This is known as the disjunctive failure model, i.e., "OR Model" [De Kleer, 2009].

The probability of a false negative occurring is, therefore, the probability of all faults not producing any failure. If the h_j values are known, they can be used to calculate this probability by

$$\varepsilon(d_k) = \begin{cases} \prod_{c_j \in d_k \wedge a_{ij}=1} h_j & \text{if } o_i = 0 \\ 1 - \prod_{c_j \in d_k \wedge a_{ij}=1} h_j & \text{if } o_i = 1 \end{cases} \tag{2.8}$$

For example, let's assume the h_j values for the system in Table 2.1 were known to be all 0.1, and we want to calculate the FNR of candidates $d_6 = \{c_6\}$ and $d_{8,10} = \{c_8, c_{10}\}$. In this case, for test t_1 ($o_1 = 1$), $\varepsilon(d_6) = 0.9$ and $\varepsilon(d_{8,10}) = 0.99$. For t_2 ($o_2 = 0$), $\varepsilon(d_6) = 1$ and $\varepsilon(d_{8,10}) = 0.1$. Assuming the initial priors are $\Pr(d_6) = \Pr(d_{8,10}) = 0.5$, the final updated value after executing all tests would be $\Pr(d_6) = 0.98$, $\Pr(d_{8,10}) = 0.02$.

Posterior False Negative Rate Estimation

Unfortunately, the h_j values are hardly ever known *a priori*. A number of approximation strategies have been proposed for the case when the individual h_j are unknown. In this chapter we will consider the $\varepsilon^{(2)}$ strategy [Abreu et al., 2008] given by

$$\varepsilon^{(2)}(d_k) = \begin{cases} g(d_k)^\eta & \text{if } o_i = 0 \\ 1 - g(d_k)^\eta & \text{if } o_i = 1 \end{cases} \tag{2.9}$$

where η is the the number of faulty statements in d_k that are also involved in the test t_i, and $g(d_k)$ is the "effective" FNR parameter for d_k, estimated by counting how many times statements in d_k are involved in failed and passed tests, according

to

$$g(d_k) = \frac{\sum_{i=1}^{N}[(\bigvee_{j \in d_k} a_{ij} = 1) \wedge o_i = 0]}{\sum_{i=1}^{N}[\bigvee_{j \in d_k} a_{ij} = 1)]} \tag{2.10}$$

where $[\ldots]$ casts truth values into $[true] = 1$, $[false] = 0$.

The failures in the example system of Table 2.1 can be explained by candidate $d_6 = \{c_6\}$ and also by $d_{8,10} = \{c_8, c_{10}\}$. The value for $g(d_6) = \frac{0}{4}$, whereas $g(d_{8,10}) = \frac{2}{6}$. For the failing tests, the $\varepsilon(d_6) = 1 - 0$. In the case of $d_{8,10}$ ε depends on the number of statements involved. For t_1 $\varepsilon(d_{8,10}) = 1 - 0.33^2 = 0.88$, and for t_3, t_4, and t_7, $\varepsilon(d_{8,10}) = 1 - 0.33^1 = 0.66$ because there is only one statement involved. Assuming that $\Pr(d_6) = \Pr(d_{8,10}) = 0.5$ initially, after the update with all tests $\Pr(d_6) = 0.97$ and $\Pr(d_{8,10}) = 0.03$.

A stable estimation requires a large number of tests, and still may be imprecise. This reduces the rate at which the diagnosis improves, requiring the execution of a large number of tests to obtain a high quality diagnosis.

Prior False Negative Rate Estimation: Testability

False negatives in the context of software fault diagnosis are intimately connected with the concept of *testability*. Of the multiple definitions of testability, the one proposed by Voas and Miller [Voas and Miller, 1995] is directly related to the concept of false negatives: *the degree to which software reveals faults during testing*. Testability relates the number of tests to be executed on a faulty statement before it exposes the fault as a failure, i.e., the complement of the FNR, $1 - h$.

Testability can be estimated by the so-called Propagation-Infection-Execution approach [Voas, 1992]. The Propagation-Infection-Execution (PIE) approach involves an expensive mutation analysis. However, its implementation is simple and can be automated. In this chapter, we use a variation of the PIE approach to obtain an estimation of the testability of a statement.

Algorithm 1 decribes the steps needed to calculate the FNR of each statement. First, for each statement c_j in the program P, we create a set of mutants, Mut. This was done by applying a small set of mutation operators [Offutt et al., 1996] to the arithmetic, logic and indexing operations contained in the statement or function. The mutations were done at the level of the bytecode representation used by the Zoltar [Janssen et al., 2009] fault diagnosis tool. Second, the program's full test suite T was run, recording test coverage A and failure information o, for each of the mutants. Finally, the FNR of each statement c_j was obtained by averaging the FNR of each mutation, calculated as the ratio of the number of tests which covered

Algorithm 1 Prior FNR Estimation

function PRIORFNR(P, C, T)
  for $j \in C$ **do**
    $Mut \leftarrow$ MUTATE(P, c_j)
    $H \leftarrow []$
    for $P' \in Mut$ **do**
      $A, o \leftarrow$ RUNTESTS(P', T)
      $h \leftarrow g(\{c_j\})$
      PUSH(H, h)
    $FNR[j] \leftarrow \sum_{h \in H} \frac{h}{|Mut|}$
  return FNR

the fault but did not produce a failure, to the number of tests which covered the fault, similar to the a posteriori formula in Equation 2.10.

It is important to note that program mutation can produce equivalent mutants, i.e., altered programs but tested as being correct. These should be removed from the average. In our study, though, we kept those mutants, because the fact that a mutant did not produce any failure in the tests could also mean that there is no test case to render the mutant faulty. Therefore, the values obtained by our FNR estimation should be seen as pessimistic. This approach was also used in [Rothermel et al., 2001]. The cost of a testability scales with $O(k \cdot M \cdot N)$, where k is the average number of mutants generated per component.

2.3 Experiment Setup

We will perform our experiments on the seven programs of the well-known Siemens benchmark set [Hutchins et al., 1994], and 5 larger programs obtained from the Software Infrastructure Repository (SIR) [Do et al., 2005]. Each program is provided with one correct version, and a set of faulty versions for each program. Every faulty version contains exactly one fault. As the few faults in Siemens would provide a very reduced set of fault combinations, in our experiments we use a larger set of faults obtained by random mutation, similar to the ones used in the PIE testability study. 100 faulty versions with 1 fault, and 100 faulty versions with 3 faults were created per program.

For each program, a test suite is provided to achieve full statement and branch coverage. As each program studied comes with a correct version, we use this as test oracle. Table 2.2 summarizes the programs used for empirical evaluation.

Following existing literature on fault diagnosis, we assume uniform *a priori* fault probability p_j [Abreu et al., 2009a]. The precise values of p_j have little influence on the performance of the diagnosis, as they are quickly corrected by the

Program	Mutants	LOC	Tests	Description
print_tokens	491	563	4,130	Lexical Analyzer
print_tokens2	294	509	4,115	Lexical Analyzer
replace	757	563	5,542	Pattern Matcher
schedule	281	412	2,650	Priority Scheduler
schedule2	212	307	2,710	Priority Scheduler
tcas	208	173	1,608	Aircraft Control
tot_info	396	406	1,052	Information Measure
space	6,282	9,126	150	ADL Compiler
gzip	20,811	7,933	210	Data Compression
sed	11,050	7,125	370	Stream Editor
grep	17,707	13,287	809	String Matching
flex	15,574	14,194	107	Lexer Generator

Table 2.2: Set of programs used in the experiments

subsequent Bayesian update process.

Table 2.3 provides an overview of the compared techniques. For completeness we have added a fifth technique: single-fault Bayesian diagnostic with PIE information (SFB-Bayes), i.e., where only d_k containing one statement are considered ($\varepsilon(d_k) = 0$ if $|d_k| > 1$).

SC		Tarantula	$l_j = \frac{\frac{n_{11(j)}}{n_{11(j)}+n_{01(j)}}}{\frac{n_{11(j)}}{n_{11(j)}+n_{01(j)}} + \frac{n_{10(j)}}{n_{10(j)}+n_{00(j)}}}$
		Ochiai	$l_j = \frac{n_{11(j)}}{\sqrt{(n_{11(j)}+n_{01(j)}) \cdot (n_{11(j)}+n_{10(j)})}}$
Bayes	Post	EPS2	$\varepsilon^{(2)}(d_k) = \begin{cases} g(d_k)^\eta & \text{if } o_i = 0 \\ 1 - g(d_k)^\eta & \text{if } o_i = 1 \end{cases}$
	Prior	PIE	$\varepsilon(d_k) = \begin{cases} \prod h_j & \text{if } o_i = 0 \\ 1 - \prod h_j & \text{if } o_i = 1 \end{cases}$
		SFB	$\varepsilon'(d_k) = \begin{cases} \varepsilon(d_k) & \text{if } \lvert d_k \rvert = 1 \\ 0 & \text{if } \lvert d_k \rvert > 1 \end{cases}$

Table 2.3: Overview of the compared techniques

2.4 Results

In this section we present and analyze the results of our experiments, and the threats to their validity. Additionally, we explore the validity of the typical assumption in diagnosis that failures are independent and obey a disjunctive (OR) model.

Single Fault Experiments

The box plot in Figure 2.1 shows the performance of each technique in terms of C_d for all programs. In addition, Table 2.4 shows, the average C_d per program. The values have been normalized by dividing by the number of non-faulty statements $M - M_f$. From our results it is apparent how the introduction of *a priori* FNR information greatly improves the performance in the diagnosis.

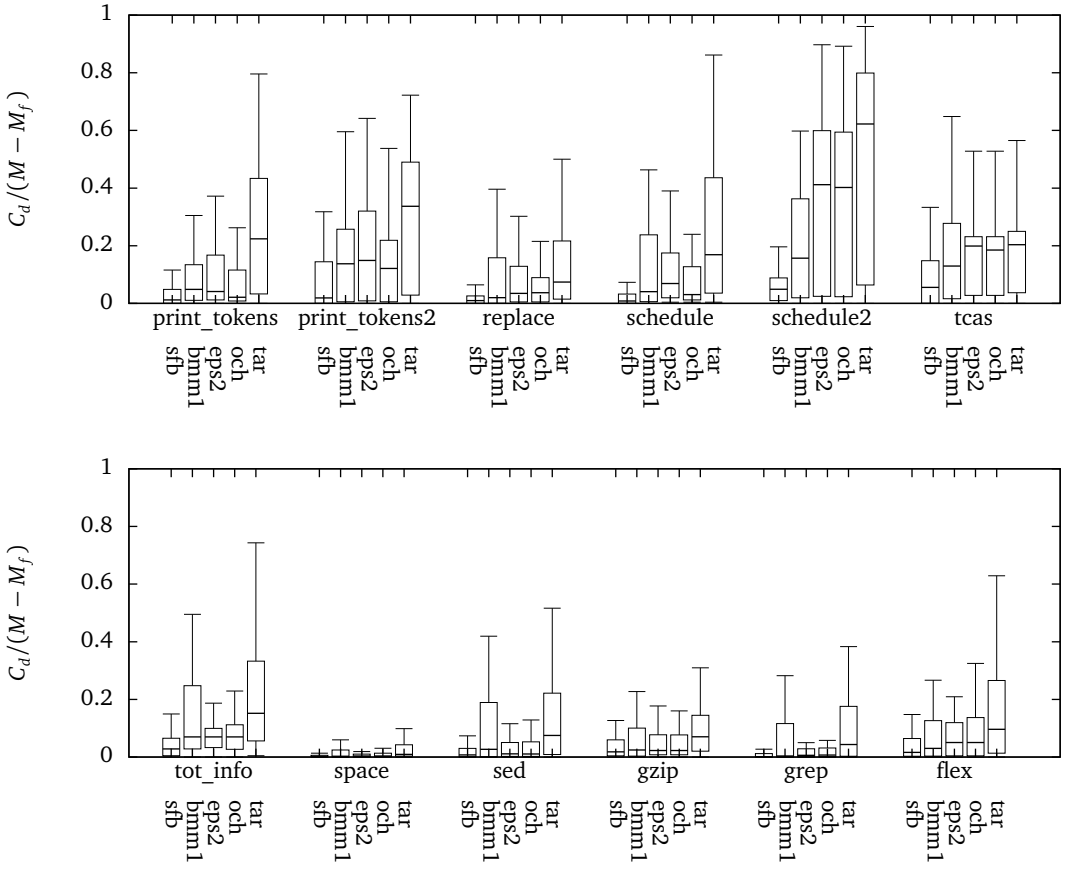


Figure 2.1: Performance for mutated programs and single faults

SFB-Bayes provides the largest improvement possible, with reductions of diagnostic effort that can reach 80% in the case of `schedule2` and are above 50% in many cases when compared with the Ochiai similarity coefficient. However, it

Program Name	SFB	PIE	EPS2	OCH	TAR
print_tokens	**0.039**	0.079	0.098	0.082	0.252
print_tokens2	**0.076**	0.159	0.189	0.143	0.317
replace	**0.019**	0.098	0.104	0.077	0.153
schedule	**0.024**	0.117	0.120	0.097	0.244
schedule2	**0.072**	0.199	0.362	0.343	0.490
tcas	**0.101**	0.178	0.207	0.194	0.217
tot_info	**0.053**	0.151	0.128	**0.094**	0.209
space	**0.006**	0.040	**0.018**	**0.022**	0.047
gzip	**0.035**	0.072	0.061	0.062	0.095
sed	**0.028**	0.098	0.067	0.065	0.144
grep	**0.010**	0.082	0.045	0.056	0.116
flex	**0.044**	0.093	0.120	0.089	0.171

Table 2.4: Residual diagnostic effort (single faults)

must be taken into account that SFB-Bayes will produce an inconsistent diagnosis if more than one statement is faulty. For this reason its applicability is limited.

It can be seen how removing the single fault assumption poses a slight penalty in the improvements that can be achieved, which is due to the diagnosis algorithm considering equally probable multiple fault candidates. Improvements on diagnosis quality with PIE-Bayes can reach 72% for `schedule2` and 60% for `schedule`. However, for some programs this is not the case. For example `tot_info` does not show any improvement, and `print_tokens2` shows a performance degradation with respect to Ochiai.

The reason for the dramatic improvement of the quality of the diagnosis, especially in the case of SFB-Bayes, is the fact that PIE information makes techniques more robust to ambiguities in the diagnosis, i.e., uncertainty if there are different statements with equal likelihoods (due to their columns in A being identical). Similarity coefficients and EPS2-Bayes estimate identical likelihood values for statements with identical columns in A, making the diagnosis more ambiguous, leading to an increase in diagnosis effort. However, the h_j values estimated for the statements inside the same ambiguity group are typically not identical. Even if their execution patterns are identical, SFB and PIE-Bayes will rank first the statements whose h_j is closer to the real FNR of tests, effectively removing ambiguity.

Multiple Fault Experiments

In this section we present the results of our experiments with programs seeded with multiple faults. Due to the small sizes of the programs in the Siemens set we seeded a maximum of $M_f = 3$ faults, in order to avoid unrealistic fault densities.

Diagnostic Performance

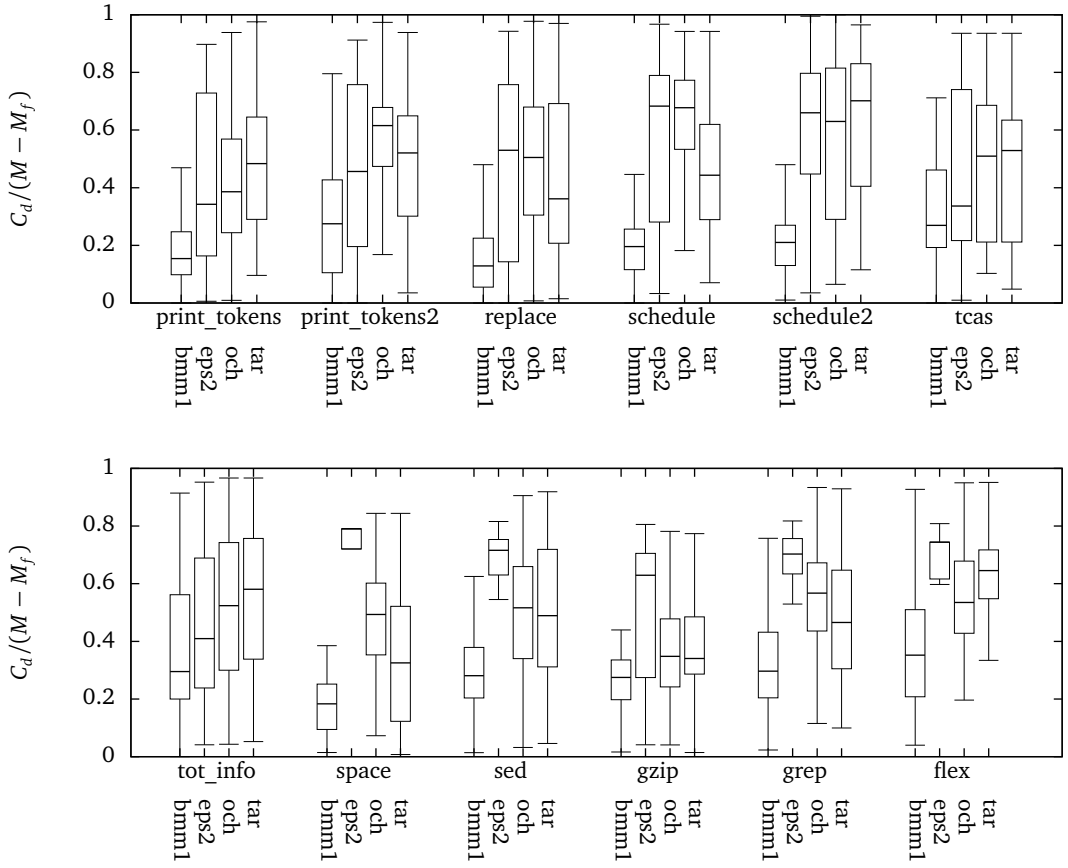


Figure 2.2: Performance for mutated programs and multiple faults

We will now comment on the performance of the multiple fault diagnosis techniques. It can be seen in Figure 2.2 and Table 2.5 how the introduction of PIE testability information greatly improves the quality of every diagnosis. PIE-Bayes provides improvements in diagnostic quality of an average of 37%, with `replace`, `schedule` and `schedule2` providing the largest improvements, of a 60%. On the bottom of the improvement scale is `tot_info` with just a 20%.

The reason for the improvements in the multiple fault case is similar to that for single faults. `tcas` and `tot_info` are the two programs with the highest approximation error on their FNR due to the failure independence assumption not holding, as we will explain in the following secion. This error is affecting the quality of the Bayesian update, reducing considerably the performance.

Program Name	PIE	EPS2	TAR	OCH
print_tokens	**0.210**	0.423	0.397	0.473
print_tokens2	**0.282**	0.456	0.550	0.482
replace	**0.160**	0.469	0.497	0.430
schedule	**0.189**	0.550	0.630	0.459
schedule2	**0.211**	0.605	0.566	0.630
tcas	**0.314**	0.481	0.478	0.452
tot_info	**0.361**	0.451	0.519	0.543
space	**0.182**	0.734	0.482	0.344
gzip	**0.271**	0.540	0.373	0.386
sed	**0.288**	0.658	0.496	0.502
grep	**0.333**	0.658	0.546	0.487
flex	**0.381**	0.703	0.549	0.617

Table 2.5: Residual diagnostic effort (multiple faults)

Testability and the OR Failure Model

Testability estimations obtained by the PIE method described in Section 2.2 are obtained in a single fault setting, i.e., the fault being studied is the only one existing in the system. However, when using PIE testability estimations in a *multiple-fault setting*, we must assess the error incurred when combining testability estimations by using the OR model assumption in Equation 2.8, and whether failure independence holds in a multiple fault environment. This is crucial since Bayesian diagnosis relies on the assumption that failures are independent events and failure rates can be combined disjunctively.

In order to answer these questions, we measured the real FNR of each possible executed fault combination (F_1, F_2, F_3, $F_1 \wedge F_2$, $F_2 \wedge F_3$, $F_1 \wedge F_3$, $F_1 \wedge F_2 \wedge F_3$) occurring in the test matrix of our mutated programs, and compared their actual values with the value estimated with Equation 2.8 and PIE.

Our experiment confirmed that, in general, the estimated FNR of a known combination of faults presents a moderate error due to the errors of the testability estimations used. However, in some cases, this error is extremely high. The scatter plots in Figure 2.3 show two examples of this situation. To explain these cases, one must remember that, in many cases, failures in software are heavily dependent on the distribution of failing inputs, which is abstracted away in fault diagnosis. The diagrams in Figure 2.4 present two situations in which the OR model and the failure independence assumption will hold, and two situations in which they will not. The circled area represents the inputs that cover a given fault, with passing inputs depicted as empty shapes, and failing inputs as filled shapes.

In case (a), the sets of inputs that cover each fault is disjoint. This case is

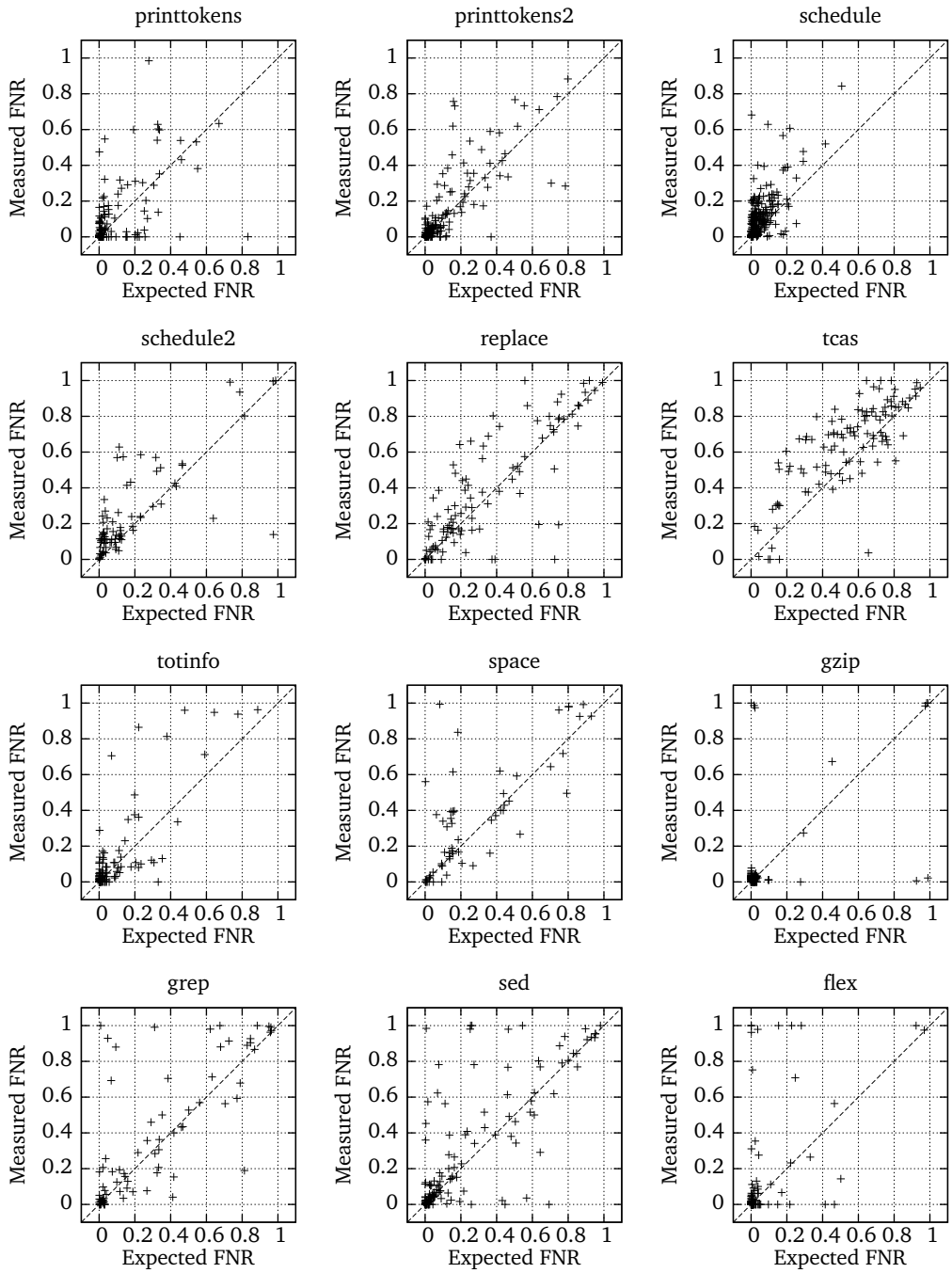


Figure 2.3: Estimation error of the FNR

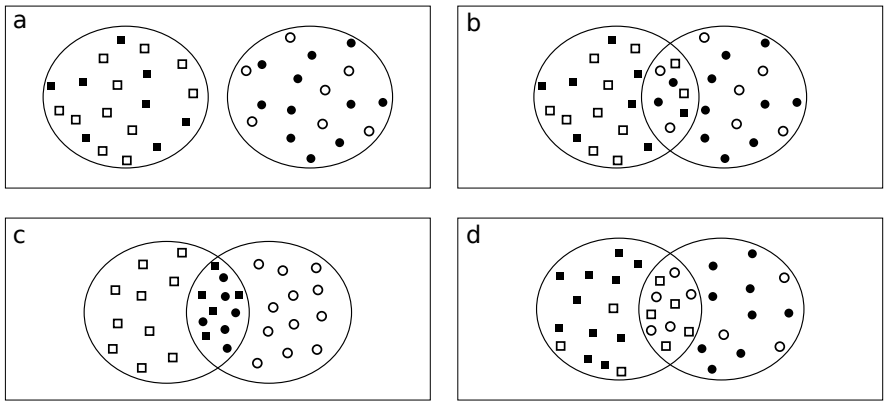


Figure 2.4: Weak OR-model at the input level

frequent in software if the faults occur in parts of the software dedicated to very different tasks. Case (b) shows a case where the OR model is applied for two faults whose failures are uniformly distributed across the input domain. When combined, this uniform distribution will still hold for the intersection area, and the OR model will present a low estimation error.

The faults in cases (c) and (d), on the other hand, do not produce failures that are uniformly distributed across their input domains. In case (c), the OR model will lead to a failure probability that is too low for the $F1 \wedge F2$ combination, and too high for the $F1$ and $F2$ combinations. Case (d) presents the reverse situation.

Threats to Validity

The Siemens programs used are small programs with very extensive and redundant test suites. The validity of is strengthened by the fact that we use the larger, more realistic programs provided by the SIR.

The validity of the results presented is further threatened by the fact that the same set of mutants were used to estimate h_j *and* to perform the diagnosis. Firstly, this could produce too optimistic results, as the estimated FNR corresponds exactly to the average of the actual FNR of the mutants being diagnosed. However, the seeded faults included in Siemens are, to a great extent, also generated by our mutation tool. Secondly, the density of the faults must be considered. As previously mentioned, we avoid performing our multiple fault experiments with more than 3 faults to keep the fault density at a realistic level. Finally, in our experiments we assumed that faults are distributed uniformly in the program code. The validity of our results could also be strengthened by considering the effect of fault distributions.

2.5 Related Work

Automated fault localization techniques minimize diagnostic cost and support debugging when failures occur during software testing. Statistical approaches include [Abreu et al., 2007, Jones et al., 2002, Liblit, 2008, Liu et al., 2005, Renieris and Reiss, 2003, Wong et al., 2008]. A recent, probabilistic approach of acceptable complexity is [Abreu et al., 2009a]. Although, different in the way they derive the ranking of the diagnosis, all techniques are based on measuring the coverage information and failure pattern of a program (also termed its spectrum), while ignoring or deriving the rate of false negatives a posteriori. The novelty of our approach consists in factoring the FNR estimation problem out of the diagnostic problem, so that FNR can be exploited from the beginning. False negatives in software are related to testability [Voas and Miller, 1995]. Testability quantification has been approached at the class level [Bruntink and van Deursen, 2006], function level [Freedman, 1991] and statement level [Offutt and Hayes, 1996, Voas, 1992, Voas and Miller, 1993, Woodward and Al-Khanjari, 2000, Zhao, 2006]. However, of the current state of the art, only [Voas, 1992] allows for a straightforward usage as FNR in our experiments. Our work is novel with respect to evaluating the degree to which software faults produce failures independently, a necessary assumption in fault diagnosis.

2.6 Summary

False negatives are an essential problem of fault localization. Previous work in diagnosis has considered FNR modeling a part of the diagnosis process. In this chapter we have studied the improvement in software fault localization when information on the false negative rate is available *a priori*, by means of a PIE testability study.

PIE-Bayes reduces the average effort for coming to the diagnosis by a 27% for real programs with single faults in average when using multiple-fault techniques. When using SFB-Bayes the performance increases dramatically, with a 60% reduction of diagnostic effort on average.

For multiple faults, prior FNR information achieves a 37% reduction of diagnostic effort on average. The performance for the multiple fault case does not only depend on the quality of the FNR estimations, but also on the degree to which they are independent and can be combined meaningfully in a disjunctive fashion.

The improvement in the fault localization process was traded against an expensive testability analysis, although, it should be noted that, in practice, the cost of such analysis is amortized over many debug-repair cycles.

3:

On-line Diagnostic Prioritization

Automatic fault localization techniques require the execution of test cases to obtain and refine their diagnosis. The order in which tests are executed influences the amount of diagnostic information each test provides. It has been shown that existing test prioritization techniques for fault detection can reduce the amount of useful information for fault localization, incurring in a sub-optimal expense of testing cost during diagnosis.

In this chapter we introduce SEQUOIA*, a novel, dynamic test case prioritization approach that maximizes the improvement of diagnostic information per executed test.* SEQUOIA *performs on-line prioritization, adapting its behavior depending on the outcome of the executed test cases.*

3.1 Test Case Prioritization **32**
3.2 Prioritization and Diagnosis **34**
3.3 Diagnostic Prioritization **36**
3.4 Evaluation **43**
3.5 Single Fault Results (IG) **45**
3.6 Multiple Faults (SEQUOIA) **49**
3.7 Related Work **50**
3.8 Summary **53**

Regression testing is a time-consuming but rather important task for improving software reliability after a modification. Two processes can be distinguished: (1) testing to find failures (e.g., regression tests), and (2) finding the fault or faults that are at the root of the failures (i.e., diagnosis).

Given the significant cost associated with regression tests, test *prioritization* has emerged as predominant technique to reduce testing effort. During regression testing, test prioritization is typically used to detect failures as soon as possible [Elbaum et al., 2004, Elbaum et al., 2002, Harrold et al., 1993b, Jiang et al., 2009a, Li et al., 2007, Rothermel et al., 2001, Smith and Kapfhammer, 2009, Wong et al., 1997] in order to reduce testing effort. The sooner failures are found, the sooner debugging can commence. During debugging, an automatic fault localization technique can be used to minimize the work the diagnostician (developer) has

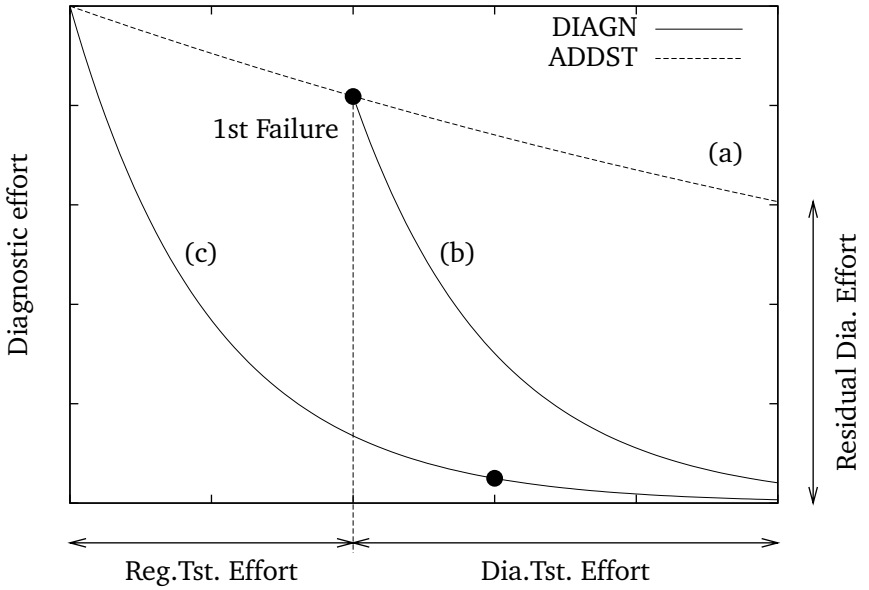


Figure 3.1: Usage of diagnostic test prioritization

to perform when going through the program to find the faults ("residual diagnostic effort") [Abreu et al., 2009b, Abreu et al., 2009a, Jones et al., 2002]. These techniques require additional tests, meaning that an additional "diagnostic testing effort" has to be invested to gather additional information to produce a refined diagnosis.

While existing prioritization techniques minimize the delay between testing and debugging, they do not maximize diagnostic information yield, and therefore will waste diagnostic testing effort by selecting useless test cases. The reason is that regression test prioritization aims at high code coverage, whereas obtaining a refined diagnosis requires partially *revisiting* already covered code to further exonerate or indict defective components. This drawback of regression test prioritization has indeed been identified [Jiang et al., 2009b, Yu et al., 2008]. To solve this problem, *diagnostic test prioritization* techniques, i.e., test prioritization techniques that maximize the yield of diagnostic information per test, have been proposed [Gonzalez-Sanchez et al., 2010a, Gonzalez-Sanchez et al., 2011e].

Figure 3.1 illustrates the situation, showing the evolution of residual diagnostic effort (residual uncertainty in the diagnosis after a certain test effort), in three possible scenarios. Scenario *a* represents the situation when using a traditional prioritization heuristic, e.g., additional statement coverage (ADDST), during diagnostic testing. The diagnostic information each test provides is very small, and if after the first failure we keep selecting test cases with ADDST, the final residual diagnostic effort when we run out of test time is very high (refining the diagnosis will require a large diagnostic test effort). Scenario *b*, on the other hand, represents

the improvement in the evolution of residual diagnostic effort if after the first failure we switch to a diagnostic test prioritization strategy (DIAGN). Finally, scenario *c* represents the potential gains if we opted for diagnostic test prioritization from the start, i.e., also during regression testing. While diagnosis cannot commence until a failure is observed, the tests executed until the first failure is observed do provide valuable diagnostic information as well if selected optimally. This scenario is suitable for software development processes with high code production rates, in which the probability of introducing at least one regression fault between subsequent commits is considerable. Consequently, the probability of having to apply diagnosis after testing is high.

This calls for a prioritization technique (i.e., *diagnostic prioritization*) that puts emphasis on *fault localization* performance from the first test, rather than failure detection performance, by performing a trade-off between testing and debugging effort.

In this chapter we present and evaluate a novel, *on-line* approach to diagnostic prioritization. Unlike traditional test prioritization, in our approach the tests are dynamically selected based on the actual pass/fail results of previously executed tests (which varies per regression cycle), leading to higher diagnostic performance per test.

When considering only single faults, the complexity of a greedy heuristic solution scales linearly with the number of possible faults (which depends on the size of the system, M). However, without any space-complexity reduction strategy (e.g., considering only minimal diagnoses), the multiple fault case scales exponentially since the heuristic must take into account all possible 2^M combinations of faults, which makes the approach unusable. In this chapter we introduce a diagnostic prioritization approach for multiple faults that delivers high diagnostic performance at sufficiently low algorithmic complexity to provide a cost-effective solution. In particular, we make the following contributions:

1. We present an analysis of *why* test prioritization for failure detection deteriorates the performance of fault localization algorithms, which motivates our alternative approach.

2. We introduce a test prioritization strategy with the goal of fault localization, contrasting with existing approaches whose goal is failure detection. Our approach performs *on-line* test prioritization depending on the outcome of the tests based on diagnostic *information gain*.

3. We present SEQUOIA (SEQUencing fOr dIAgnosis), a low-complexity sequencing algorithm based on information gain, for multiple faults.

4. We study the performance of SEQUOIA for the Siemens set [Hutchins et al., 1994] as well as larger programs `space`, `gzip`, `sed`, `grep`, and `flex`, avail-

able from the Software Infrastructure repository (SIR) [Do et al., 2005], which we have extended to accommodate multiple faults.

The chapter is organized as follows. Section 3.1 surveys the existing prioritization techniques with which we will compare our approach. In Section 3.2, we describe why current prioritization techniques fall short for fault localization. Section 3.3 introduces diagnostic prioritization and the information gain heuristic. The heuristic approximation for the multiple fault case is also introduced here. Our evaluation goals and experimental setup are described in Section 3.4. Sections 3.5 and 3.6 present the results for single and multiple faults, respectively. Related work is surveyed in Section 3.7. Section 3.8 presents our final conclusions and future work directions.

3.1 Test Case Prioritization

Test case prioritization techniques order test cases with respect to a given goal, so that those tests with the highest utility (which bring the test process closer to its goal), are given higher priorities and therefore are executed earlier in the testing process.

A *failure* is a deviation of the expected behavior of a program, caused by a fault. The most common prioritization goal is to increase the rate of failure detection. It means, tests are executed in an order such that failures occur as early as possible in the testing process, so that confidence in the presence or absence of faults is reached faster. The following prioritization techniques have been proposed in order to achieve this goal, and will be used in the evaluation to compare with our diagnostic prioritization technique.

Random

Random prioritization (RND) is the most straightforward prioritization criterion. It orders test cases according to random permutations of the original test suite. Random permutations are used as control in many prioritization experiments [Elbaum et al., 2002, Rothermel et al., 2001, Smith and Kapfhammer, 2009].

Statement coverage

Following statement coverage prioritization, the test cases that cover the highest total number of statements are executed first. This approach is based on the assumption that the more statements are covered by a test, the higher is the probability of triggering a failure. If a statement is covered without producing a failure, covering it again is less useful as it is less likely it will produce a failure [Elbaum et al., 2002, Rothermel et al., 2001]. This reasoning conduces to the definition of the *additional coverage* heuristic (ADDST), where test cases are selected iteratively

in terms of the additional coverage they yield, taking into account all the test cases that were already executed, i.e.,

$$\text{ADDST}(t_i) = \sum_{j=1}^{M} a_{ij} \cdot (1 - cov_j) \tag{3.1}$$

where a_{ij} indicates whether test i covers statement j, and $cov_j = 1$ if statement j has been covered so far. M indicates the total number of statements in the system.

Fault-exposing potential (FEP)

This criterion adapts ADDST's "binary" approach (i.e., covered, not covered) to a continuous confidence value, to account for the fact that a test may produce a false negative. If a statement has been covered a number of times, the confidence on its correctness increases, making it less likely to be chosen again. We will consider only the additional FEP prioritization heuristic, given by

$$\text{FEP}(t_i) = \sum_{j=0}^{M} FEP_{ij} \cdot (1 - conf_j) \tag{3.2}$$

where $conf_j$ is the confidence in statement$j$ being correct. FEP_{ij} is the probability that test i will fail for statement j. The value of $conf_j$ is updated after executing test t_i according to

$$conf'_j = conf_j + FEP_{ij} \cdot (1 - conf_j) \tag{3.3}$$

In our experiments, the FEP_{ij} value corresponds to the complementary value of the false negative rate (FNR) described in Chapter 2: $FEP_{ij} = 1 - h_j$.

Prioritization criteria based on the *additional* policy are NP-Hard search problems, and are usually approximated by greedy search, although other sub-optimal algorithms do exist [Li et al., 2007].

Adaptive Random Testing

ART is a hybrid random-coverage-based test ordering [Jiang et al., 2009a]. It selects test cases in two steps, first it samples a group of tests randomly, and from that group it selects the test that maximizes a distance function with the already executed test cases. This distance function can be either the minimum distance with all executed tests, the maximum distance, or the average distance. In this thesis we will compare with the minimum distance heuristic, as it was cited [Jiang et al., 2009a] as the most promising one. It is defined as

$$\text{ART}(t_i) = \min_{t_j \in C}(\delta(t_i, t_j)) \tag{3.4}$$

where C is the set of already applied tests and δ is the distance function used, in [Jiang et al., 2009a] the Jaccard distance, given by

$$\delta(t_i, t_j) = 1 - \frac{|k|a_{ik} \wedge a_{jk}|}{|k|a_{ik} \vee a_{jk}|} \tag{3.5}$$

3.2 Prioritization and Diagnosis

Previous empirical work has shown that early failure detection and fault localization seem to be rather incompatible goals [Jiang et al., 2009b, Yu et al., 2008]. The evolution of the diagnostic effort C_d, per unit of test effort, C_t, is negatively affected by criteria for early failure detection. Random ordering, which has been traditionally considered the baseline prioritization technique [Elbaum et al., 2002, Rothermel et al., 2001, Smith and Kapfhammer, 2009], was found to perform as good as or better than all other prioritization techniques, except ADDST. However, even for the latter case the random order was better for some subject programs [Jiang et al., 2009b].

The main reason for the poor diagnostic performance of existing prioritization techniques is that they perform prioritization in such a way that tests maximize the probability of failing. This approach may be appropriate for regression testing, but not for fault diagnosis. When performing fault diagnosis, if a test has failed, the components covered by the test become important suspects. However, many regression prioritization algorithms will choose a next test that covers *different* components, whereas from the diagnostic point of view, the next test case should help differentiate between the *current* suspects. Therefore a test order independent of the outcomes of the tests cannot be used. The order has to be adapted *on-line*, depending on the output of the previous tests.

Table 3.2 shows an example of how traditional test prioritization for failure detection fails for diagnosis, when using the ADDST heuristic for the example system shown in Table 3.1. We use the single-fault Bayesian diagnosis approach from Section 2.1, assuming a fault will cause a 10% of false negatives ($h_k = 0.1$). For clarity, the fourth column shows only the probabilities of diagnostic explanations that are non-zero. Initially, no statement has been covered, and D ranks every candidate with uniform probability.

ADDST selects test t_1 as first test, as it covers the most test cases, and, indeed, t_1 produces the first failure. As a result of the failure, all the c_k covered by t_1 move to the top of the ranking. Unfortunately, the test case covered many statements, so C_d does not decrease too much. In the second step, test t_5 is selected because it provides the highest additional coverage, and passes. Because it passed, the updated probability of those candidates in D that correspond to statements covered by t_5 drop below the ones that were not covered. The extent of this drop depends on h_j, the lower h_j the lower they will drop. The statements that where not covered remain at the top of the ranking. Full coverage has been reached, so

	Program: Character Counter	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	Prior
c_1	`main() {`	1	1	1	1	1	1	1	1	$^1/_{13}$
c_2	`int let, dig, other, c;`	1	1	1	1	1	1	1	1	$^1/_{13}$
c_3	`let = dig = other = 0;`	1	1	1	1	1	1	1	1	$^1/_{13}$
c_4	`while(c = getchar()) {`	1	1	1	1	1	1	1	1	$^1/_{13}$
c_5	`if ('A'<=c && 'Z'>=c)`	1	1	1	1	1	1	1	0	$^1/_{13}$
c_6	`let += 2; /* FAULT */`	1	0	1	1	0	0	1	0	$^1/_{13}$
c_7	`elif ('a'<=c && 'z'>=c)`	1	1	0	1	1	1	1	0	$^1/_{13}$
c_8	`let += 1;`	1	0	0	0	1	0	1	0	$^1/_{13}$
c_9	`elif ('0'<=c && '9'>=c)`	1	1	0	1	1	1	0	0	$^1/_{13}$
c_{10}	`dig += 1;`	1	1	0	1	0	0	0	0	$^1/_{13}$
c_{11}	`elif (isprint(c))`	0	0	0	0	1	1	0	0	$^1/_{13}$
c_{12}	`other += 1;}`	0	0	0	0	1	0	0	0	$^1/_{13}$
c_{13}	`printf("%d %d %d\n", let, dig, other);}`	1	1	1	1	1	1	1	1	$^1/_{13}$
	Test case outcomes	1	0	1	1	0	0	1	0	

Inputs for test cases: t_1: `"Al84"`, t_2: `"1337"`, t_3: `"test1"`, t_4: `"UB40"`, t_5: `"#icanhaz"`, t_6: `"\0x10\0x05\0x00"`, t_7: `"Alberto"`, t_8: `""`

Table 3.1: Faulty program and Fault Diagnosis inputs

in the third step, the coverage is reset as described in [Rothermel et al., 2001], instead of falling back to a random sequence. Test t_4 provides the highest coverage, and indeed fails. However, it covered both c_6 and c_{10}, so it provides no extra information and neither D nor C_d change. This happens also in the fourth step for t_6. Finally, t_7, a test case that covers c_{10} but not c_6 is chosen. As it fails, c_{10}, which was not covered, is assigned a zero probability (since we assumed there is only one fault). c_6 remains as the only most likely explanation. As we can see, ADDST selects 2 tests that provide no information to the diagnosis, independent of their outcome, i.e., resulting in wasted testing effort C_t from the point of view of diagnostic performance C_d.

As a comparison, Table 3.3 shows the optimal test order for unequivocally diagnosing a fault in c_6. With just one test case, the set of candidates is drastically reduced. The next test case finalizes the diagnosis by using a test case that bisects D. The order of the remaining tests is irrelevant for the diagnosis, as none will provide more information. The plot in Figure 3.2 depicts the evolution of both approaches.

Although simple, this example demonstrates that maximizing the probability of failure does not maximize the information that the diagnostic algorithm receives. In fact, as test cases that cover many statements are those with the highest failure probability, those tests will not provide much useful information because the number of remaining diagnostic candidates will not decrease substantially.

In contrast, for test prioritization for diagnosis, we need test cases that revisit partially the already tested system, to introduce diversity in the coverage and reduce the set of suspect components.

Test	o_i	Covered Statements	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9	c_{10}	c_{11}	c_{12}	c_{13}	C_d
		0 0 0 0 0 0 0 0 0 0 0 0 0	0.07	0.07	0.07	0.07	0.07	0.07	0.07	0.07	0.07	0.07	0.07	0.07	0.07	0.500
t_1	1	1 1 1 1 1 1 1 1 1 1 0 0 1	0.09	0.09	0.09	0.09	0.09	0.09	0.09	0.09	0.09	0.09			0.09	0.357
t_5	0	1 1 1 1 1 0 1 1 1 0 1 1 1	0.03	0.03	0.03	0.03	0.03	0.34	0.03	0.03	0.03	0.34			0.03	0.038
t_4	1	1 1 1 1 1 1 1 0 1 1 0 0 1	0.03	0.03	0.03	0.03	0.03	0.35	0.03		0.03	0.35			0.03	0.038
t_6	0	1 1 1 1 1 0 1 0 1 0 1 0 1	0.00	0.00	0.00	0.00	0.00	0.48	0.00		0.00	0.48			0.00	0.038
t_7	1	1 1 1 1 1 1 1 1 0 0 0 0 1	0.01	0.01	0.01	0.01	0.01	0.93	0.01						0.01	0.000
t_2	0	1 1 1 1 1 0 1 0 1 1 0 0 1	0.00	0.00	0.00	0.00	0.00	0.99	0.00						0.00	0.000
t_3	1	1 1 1 1 1 1 0 0 0 0 0 0 1	0.00	0.00	0.00	0.00	0.00	0.99	0.00						0.00	0.000
t_8	0	1 1 1 1 0 0 0 0 0 0 0 0 1	0.00	0.00	0.00	0.00	0.00	0.99	0.00						0.00	0.000

Table 3.2: Evolution of D for the ADDST heuristic for our example system.

Test	o_i	Covered Statements	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9	c_{10}	c_{11}	c_{12}	c_{13}	C_d
		0 0 0 0 0 0 0 0 0 0 0 0 0	0.07	0.07	0.07	0.07	0.07	0.07	0.07	0.07	0.07	0.07	0.07	0.07	0.07	0.500
t_5	0	1 1 1 1 1 0 1 1 1 0 1 1 1	0.03	0.03	0.03	0.03	0.03	0.32	0.03	0.03	0.03	0.32	0.03	0.03	0.03	0.038
t_7	1	1 1 1 1 1 1 1 1 0 0 0 0 1	0.05	0.05	0.05	0.05	0.05	0.55	0.05	0.05					0.05	0.000
t_6	0	1 1 1 1 1 0 1 0 1 0 1 0 1	0.01	0.01	0.01	0.01	0.01	0.85	0.01	0.08					0.01	0.000
t_1	1	1 1 1 1 1 1 1 1 1 1 0 0 1	0.01	0.01	0.01	0.01	0.01	0.85	0.01	0.08					0.01	0.000
t_4	1	1 1 1 1 1 1 1 0 1 1 0 0 1	0.01	0.01	0.01	0.01	0.01	0.93	0.01						0.01	0.000
t_2	0	1 1 1 1 1 0 1 0 1 1 0 0 1	0.00	0.00	0.00	0.00	0.00	0.99	0.00						0.00	0.000
t_3	1	1 1 1 1 1 1 0 0 0 0 0 0 1	0.00	0.00	0.00	0.00	0.00	0.99	0.00						0.00	0.000
t_8	0	1 1 1 1 0 0 0 0 0 0 0 0 1	0.00	0.00	0.00	0.00	0.00	0.99	0.00						0.00	0.000

Table 3.3: Optimal evolution of D for c_6 in our example system.

Test	o_i	Covered Statements	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9	c_{10}	c_{11}	c_{12}	c_{13}	C_d
		0 0 0 0 0 0 0 0 0 0 0 0 0	0.07	0.07	0.07	0.07	0.07	0.07	0.07	0.07	0.07	0.07	0.07	0.07	0.07	0.500
t_3	1	1 1 1 1 1 1 0 0 0 0 0 0 1	0.14	0.14	0.14	0.14	0.14	0.14							0.14	0.214
t_8	0	1 1 1 1 0 0 0 0 0 0 0 0 1	0.04	0.04	0.04	0.04	0.40	0.40							0.04	0.038
t_2	0	1 1 1 1 1 0 1 0 1 1 0 0 1	0.01	0.01	0.01	0.01	0.08	0.86							0.01	0.000
t_5	0	1 1 1 1 1 0 1 1 1 0 1 1 1	0.00	0.00	0.00	0.00	0.01	0.98							0.00	0.000
t_6	0	1 1 1 1 1 0 1 0 1 0 1 0 1	0.00	0.00	0.00	0.00	0.00	0.99							0.00	0.000
t_1	1	1 1 1 1 1 1 1 1 1 1 0 0 1	0.00	0.00	0.00	0.00	0.00	0.99							0.00	0.000
t_4	1	1 1 1 1 1 1 1 0 1 1 0 0 1	0.00	0.00	0.00	0.00	0.00	0.99							0.00	0.000
t_7	1	1 1 1 1 1 1 1 1 0 0 0 0 1	0.00	0.00	0.00	0.00	0.00	0.99							0.00	0.000

Table 3.4: Evolution of D for the IG heuristic for our example system.

3.3 Diagnostic Prioritization

In the following we will present *diagnostic prioritization*, an on-line greedy prioritization approach that takes into account the observed test outcomes to determine the next test case. Our work is inspired by research in *sequential diagnosis* of hardware systems, where algorithms exist to diagnose systems with permanent [Shakeri et al., 2000] and intermittent [Raghavan et al., 1999] failures.

Diagnostic prioritization uses the same inputs as traditional test prioritization and fault localization techniques in software engineering: component set C, prior fault probabilities p_j, tests T and coverage matrix A, as shown in the example

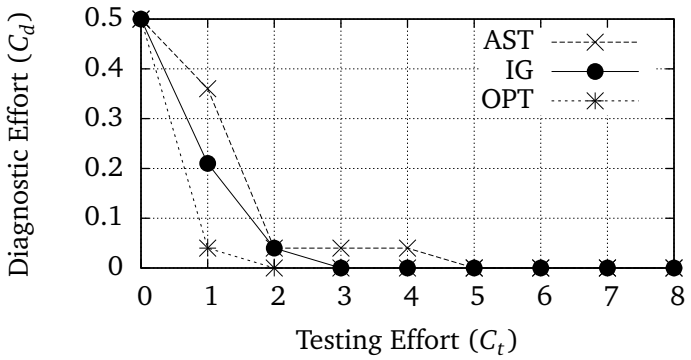


Figure 3.2: $C_d(C_t)$ for three prioritization approaches

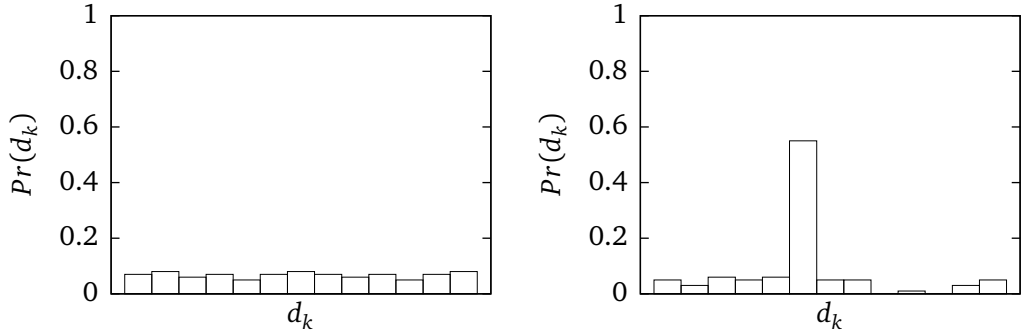


Figure 3.3: Very ambiguous diagnosis (left) and very precise diagnosis (right).

system in Table 3.1.

For diagnosis, the best tests are those that, at each step, maximize the reduction of diagnostic cost C_d. However, since we do not know the correct diagnosis (d_*) a different heuristic has to be used.

Information Gain

In general, good diagnoses are those in which one of the candidates has a much larger probability than the rest, i.e, it stands out over all other candidates. Figure 3.3 illustrates what we want to achieve. The plot on the left shows a poor diagnosis where no clear candidate exists. Our objective is to produce a diagnosis like the one on the right hand side plot.

A reduction in the ambiguity of the diagnosis can be seen as an increase in diagnostic information, i.e., a reduction of the information entropy of the candidate set D. Applying this reasoning, at each decision step in the test sequence, the test yielding the highest average information gain is chosen. The information gain

heuristic [Johnson, 1960], IG, is defined as

$$\mathrm{IG}(D, t_i) = \mathrm{H}(D) - \Pr(o_i = 0) \cdot \mathrm{H}(D|o_i = 0) - \Pr(o_i = 1) \cdot \mathrm{H}(D|o_i = 1) \quad (3.6)$$

where $\mathrm{H}(D)$ is the information entropy of the diagnostic candidate set D, defined as

$$\mathrm{H}(D) = -\sum_{d_k \in D} \Pr(d_k) \cdot \log_2(\Pr(d_k)) \quad (3.7)$$

$D|o_i = 0$ represents the updated diagnosis if test t_i passes, and $D|o_i = 1$ if it fails. In the case when any $\Pr(d_k|o_i) = 0$, H can still be calculated, as $\lim_{x \to 0} = x \cdot \log_2 x = 0$.

The rationale of the IG heuristic is that H is an estimation of both the remaining tests towards an unambiguous diagnostic, and the residual diagnostic cost if testing would stop at the given state. Under ideal conditions, diagnostic prioritization performs a binary search, bisecting the set of candidates after each test. Therefore, the number of tests (C_t) needed to reach a diagnosis is related to the number of binary tests needed to separate the candidates.

Furthermore, H and C_d are both monotonically decreasing after each test. Ideally, after each test, D contains half the number of candidates with non-null probabilities, reducing C_d in half and H by 1 bit. Therefore, a decrease in H also represents a reduction in residual diagnostic cost C_d, even when their correlation is not so strong.

Single-fault Diagnostic Prioritization

Assuming there can only be one fault in the system, the IG heuristic can be easily calculated in linear time, since only M diagnostic candidates are possible (i.e., the candidates containing only one component).

Algorithm 2 Greedy Information Gain Prioritization

$D \leftarrow (\{c_1\}, \ldots, \{c_M\})$
for all $d_k \in D$ **do**
  $\Pr[d_k] = p_j$
for $l \leftarrow 1, N$ **do**
  $i = \arg\max\left(A, IG(D, t_i)\right)$
  $o_i = \textsc{RunTest}(t_i)$
  for all $d_k \in D$ **do**
    $\Pr[d_k] = \frac{\Pr(o_i|d_k) \cdot \Pr[d_k]}{\Pr(o_i)}$
  $\textsc{RemoveRowIn}(A, i)$
return $\textsc{Sort}(D, \Pr)$

The pseudocode in Algorithm 2 describes all the steps in the information gain prioritization procedure. At every decision step, the test that maximizes IG is chosen, and then executed by the RunTest command. Once the result is available,

the probabilities of each of the candidates are updated. To avoid repeating a test, its row in A is removed by RemoveRowIn. Table 3.4 shows the evolution of D and $\Pr(d_k)$ in our example, for each test selected by the algorithm. The plot in Figure 3.2 depicts the evolution of C_d with respect to C_t compared to ADDST and the optimal solution.

Time/Space Complexity

Conceptually, when considering all the possible test outcomes, a test suite prioritized for IG diagnosis is a tree, in contrast to off-line prioritization techniques using a static list. Figure 3.4 shows an example of such a tree. Circular nodes contain the updated sets of diagnostic candidates D at each point in the decision process, and rectangular nodes represent which test is applied at each step.

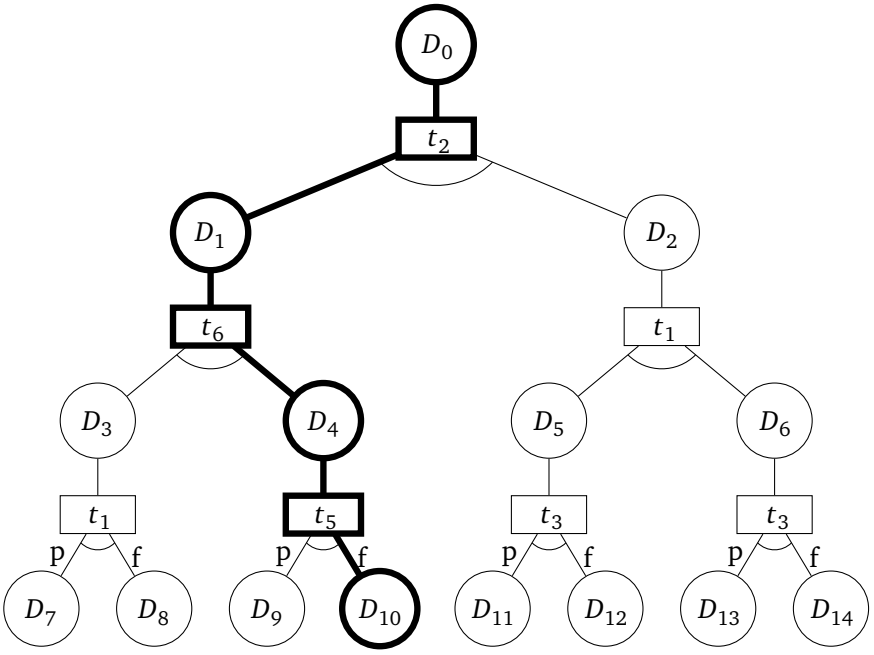


Figure 3.4: Decision tree of the IG-prioritized test suite

Although the complete tree has up to $O(2^N)$ nodes, and therefore the complexity of diagnostic prioritization is $O(M \cdot N \cdot 2^N)$, when calculated on-line, only the path corresponding to the observed test outcomes has to be calculated. This is marked with thicker lines in Figure 3.4. Consequently, the algorithmic complexity of the information-gain approach is $O(M \cdot N^2)$, similar to the ADDST heuristic. Comparison with the worst case $O(M^3 \cdot N)$ complexity of ART [Jiang et al., 2009b] depends on the relative size of M and N. In the benchmark suite used in our experiments N is much bigger than M, therefore ART has a somewhat lower cost.

Multiple-fault Diagnostic Prioritization

Due to the exponential complexity of computing a Bayesian diagnosis that considers all possible combinations, computing the IG heuristic would be impractical for realistic system sizes. In this section we present our sequencing method SEQUOIA, featuring a new, low-cost *approximate* Bayesian diagnosis kernel, that trades some diagnostic accuracy for polynomial cost. Our approximation approach is based on a number of innovations, described in the following sections.

Approximate Information Gain

One of the main problems of IG prioritization is that it requires maintaining $O(2^M)$ candidates, d_k, which is prohibitive for realistic problem sizes. To reduce the number of candidates we first tried the usual approach in diagnosis of keeping only *minimal* ones, instead of all possible candidates (e.g., only generate $\{c_6\}$ instead of $\{c_6\}$ **and** $\{c_6, c_8\}$, as the former candidate already fully explains all test outcomes and is much more probable). To this end we employ the STACCATO algorithm [Abreu and van Gemund, 2009] that generates minimal diagnoses in (approximate) order of posterior probability at polynomial cost. To further control complexity the set of candidates produced by STACCATO is typically truncated to the first λ elements as the probability mass is concentrated in the first candidates.

However, while diagnostic reports comprising *minimal* diagnoses have high practical utility in fault localization [de Kleer et al., 1992, Reiter, 1987, Wotawa, 2010], they have limited use for IG computation based on D since they distort the entropy value H 3.7. Consider the following matrix A, i.e.,

c_1	c_2	c_3	o
1	0	0	1
0	1	0	1
0	0	1	1

corresponding to a triple-fault $D = \langle\{c_1, c_2, c_3\}\rangle$. After the first test, a minimal diagnosis algorithm will return the singleton $D = \langle\{c_1\}\rangle$ and $H = 0$. Regardless whether we consider test 2 or 3 the information gain will be zero since D will return a double-fault with, again, $H = 0$.

Clearly, the quality of the H computation is compromised due to the fact that essential (non-minimal) candidates in D are missing. To remedy this, we present an approximation, based on *expanding* each minimal candidate returned by STACCATO with *one* additional component. We refer to this expansion as *first-order* approximation of the entire candidate set in D (HS1).

Figure 3.5 shows the evolution of H versus i for a randomly generated A with $N = 150$, $M = 10$, $M_f = 3$, FNR $h_j = U(0, 1)$. Three plots are shown, i.e., H for the perfect D with all non-minimal candidates (OPT), the first-order approximation (HS1), and the original MHS-based D (MHS), respectively. The results clearly

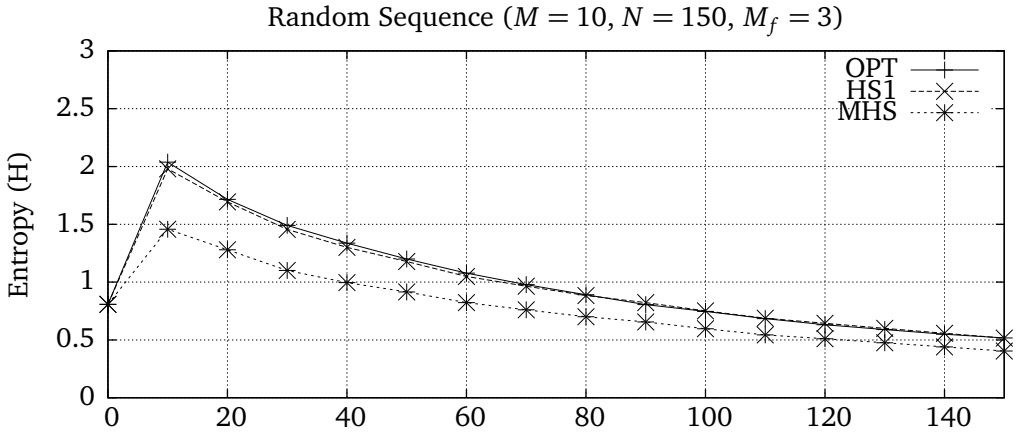


Figure 3.5: Different approximations of H

show that a first-order approximation already delivers good accuracy. This is significant since the cost of generating first-order subsumptions is non-negligible (in this example approximately 100 additional candidates are generated per MHS). Therefore HS1 is also limited by a parameter, γ.

Algorithm

Algorithm 3 describes the SEQUOIA algorithm. The algorithm takes as arguments the set of components $COMPS$ and the associated priors p_j and failure intermittencies h_j, the test matrix A, and test costs, C_i, and two entropy estimation parameters: the maximum number of minimal diagnosis candidates λ as generated by STACCATO and the maximum number of non-minimal candidates γ subsequently generated by the 1st-order hitting set extension.

T represents the index set of the available tests. A' (initially empty) represents the version of A whose rows are sorted by the sequencing algorithm, and o stores the pass/fail outcomes of the tests. The INITIALD function initializes D to the first γ candidates in descending order of posterior probability, each d_k probability is stored in the array PrD.

The main loop performs the test sequencing. First, a test is selected from a *subset* of σ tests (SAMPLE) that yields the highest information gain per unit of cost (C_i represents the cost of each test). Merely considering a subset (e.g., $\sigma = 100$) instead of all available tests ($O(N)$) yields a substantial gain, while the sacrifice in optimality is small[1]. Next, the test is executed based on the *current* D and Pr. The selected test and its outcome are appended to A' and o (the || operators denote appending a matrix row, and vector element, respectively). Second, D

[1]The information gain selected by the argmax is the highest order statistic from the information gain distribution in the test set. Its mean value grows less than logarithmic with the sample size [David, 1970].

Algorithm 3 SEQUOIA

procedure SEQUOIA(COMPS, A, C, λ, γ)
  $T \leftarrow \{1, \ldots, N\}$
  $A' \leftarrow \emptyset$
  $o \leftarrow \emptyset$
  $D, \text{PrD} \leftarrow$ INITIALD(COMPS, γ)
  while $T \neq \emptyset$ **do**
    $i \leftarrow \arg\max_{i \in \text{SAMPLE}(T)} \frac{\Delta\text{H}(D, i)}{C_i}$
    $o_i \leftarrow$ RUNTEST(i)
    $A' \leftarrow A' || A_{i*}$
    $o \leftarrow o || o_i$
    $D \leftarrow$ STACCATO(A', e, λ)
    $D \leftarrow$ EXPANDSUBSUMED(D, COMPS, γ)
    for all $d_k \in D$ **do**
      $\text{PrD}[d_k] \leftarrow$ BAYESUPDATE(d_k, A', o)
    $T \leftarrow T \setminus \{i\}$

is updated by the STACCATO algorithm, and subsequently extended to contain the first order subsumed (non-minimal) candidates by EXPANDSUBSUMED. In order to keep complexity under control, only the first λ minimal candidates are generated. Likewise, only the first γ first-order subsumed candidates are generated. Third, the probabilities of the newly obtained candidates are calculated by the BAYESUPDATE function. Finally, the selected test (index) i is removed from the set of available tests T.

Time Complexity

To minimize computational cost SEQUOIA only stores a limited number of candidates in D. As a general guideline from our experiments, $\lambda = M$ and $\gamma = 10 \cdot M$ provide good performance.

For every iteration of the main loop, ΔH has to be calculated for the σ test cases in the sample, with a cost of $O(\gamma M)$. The STACCATO algorithm has a complexity of $O(M_f \cdot M \cdot (N + \log M))$ [Abreu and van Gemund, 2009] although in practical situations, its complexity can be considered $O(M \cdot N)$ since M_f is small and $N \gg \log M$. The Bayesian update of a candidate accounts $O(\gamma \cdot M \cdot N)$.

The combined complexity of each of the steps in the inner loop is, in light of these results, $O(M \cdot N)$, albeit with a significant factor. Our timing experiments show that 1000×1000 matrices require 11s per test choice on average.

Fall-back Mechanism on FNR Estimation Errors

Both the single and multiple-fault experiments can suffer from imprecise FNR rates (h_j) can affect, which can affect the IG heuristic in SEQUOIA very negatively. Espe-

cially, if the IG provided by tests is very low (see Chapter 5), any small error in FNR will have a great effect on the value of IG.

A simple solution is to monitor the value of the IG heuristic, and switch to a random order if a decrease of IG below a threshold is observed. This *mode switch* greatly reduces the cost of SEQUOIA. Since the entropy of D decreases exponentially with the number of tests, SEQUOIA only needs to be invoked during a small number of tests providing significant gains over other approaches. The IG threshold for declaring a test matrix as exhausted depends on the characteristics of the system (especially the average FNR) and the test matrix A.

Algorithm 4 SEQUOIA with Fallback

```
procedure SEQUOIA(COMPS, A, C, λ, γ)
    T ← {1, ..., N}
    A' ← ∅
    o ← ∅
    D, PrD ← INITIALD(COMPS, γ)
    while T ≠ ∅ do
        if not FALLBACK( ) then
            i ← arg max_{i∈SAMPLE(T)} ΔH(D, i) / C_i
        else
            i ← RANDOM(T)
        o_i ← RUNTEST(i)
        A' ← A' || A_{i*}
        o ← o || o_i
        D ← STACCATO(A', e, λ)
        D ← EXPANDSUBSUMED(D, COMPS, γ)
        for all d_k ∈ D do
            PrD[d_k] ← BAYESUPDATE(d_k, A', o)
        T ← T \ {i}
```

Algorithm 4 shows the modification to SEQUOIA. The FALLBACK function returns `true` if the maximum IG has been below 0.2 for 3 iterations. More sophisticated fall-back conditions can be devised, by taking into account the parameters identified in Chapter 5, however, this is left for future work.

3.4 EVALUATION

In order to evaluate the applicability and performance of diagnostic prioritization, we address the following questions.

Question 1: What is the evolution of diagnostic effort (C_d) with respect to testing effort (C_t) for the information gain heuristic IG and SEQUOIA? How do IG and SEQUOIA (with and without fall-back) compare to RND, ADDST, FEP and ART?

Question 2: What is the fault detection performance of the sequences produced by IG? Since IG is not aimed at maximizing fault probability, we expect an impact on the failure detection performance. Our aim is to quantify this impact.

In our validation we opt for three different approaches:

1. **All-synthetic matrices:** We generate synthetic observations based on sample (A, o) generated for various values of N, M, and number of injected faults M_f. The motivation for using synthetic data next to data from real programs is the ability to study the effect of the various parameters in a controlled setting. whereas real programs only represent a few parameter settings in the multi-dimensional parameter space.

   Component activity a_{ij} is sampled from a Bernoulli distribution with parameter ρ, i.e., the probability a component is involved in a row of A equals ρ. All test costs are equal. For the M components we also set $h_j = U(0,1)$. Thus the probability of a component being involved *and* generating a failure equals $\rho \cdot (1 - h_j)$. A row n in A generates an error ($o_n = 1$) if at least 1 of the M_f components generates a failure (or-model), as done in Chapter 2.

2. **Real matrices with synthetic failures:** The coverage matrix A of each program is obtained by instrumenting the compiled program binary with the `Zoltar` tool [Janssen et al., 2009], to obtain the statements covered by each test case. Instead of using the failure information of the test cases, we simulate a bigger sample of faults and error vectors than the ones provided by the Siemens set. The test outcomes are obtained by randomly choosing M_f faulty statements with uniform probability. The simulated error vector is obtained by using the faulty statements' columns in A. Every time any of the faults is covered, an error is simulated following the same reasoning as in the synthetic case. To achieve more realistic results, the simulated FNR rate of each component corresponds to the value calculated by the PIE method in Chapter 2.

3. **Real matrices and real failures:** In this scenario, the programs were seeded a certain number of faults at random positions in the source code, by means of mutation operators. Using the correct version of each program as oracle, we recorded the program spectra and test outcomes, to obtain real coverage and inputs to the diagnostic and prioritization algorithms.

For our study with real programs, we use the Siemens set [Hutchins et al., 1994], and 5 programs from the Software Infrastructure Repository (SIR) [Do et al., 2005], already presented in Chapter 2.

To answer Question 1, we measure and plot the evolution of C_d with respect to C_t for the first 100 tests of each program's prioritized test suite, for 10 runs for 100 sample faults (1000 runs in total per technique). We compare the RND, ART,

ADDST, FEP and IG heuristics in terms of *diagnostic effectiveness*, the normalized area S under the C_d curve and above the diagnostic asymptote (the C_d value when the complete test suite has been executed), per unit of test cost C_i, according to

$$S = \sum_{n=1}^{N} \left(\frac{C_d(n-1) - C_d(N)}{M - M_f} \cdot C_i \right) \tag{3.8}$$

We prefer this metric to, e.g., the value of C_d after an arbitrary number of tests or the number of tests required to reach an arbitrary C_d. The reason for this is that we want to avoid the definition of arbitrary performance thresholds.

With respect to Question 2, the test case in which the first failure occurs is stored, for each of the prioritized test suites. We compare the occurrence of the first failure for RND, ART, ADDST, FEP and IG heuristics. Following [Rothermel et al., 2001] we calculate the APFD measure to evaluate the rate of fault detection for the prioritized test suites. For a test suite with N tests and a set of m faults, where each fault F_i is first revealed in test n_{ffi}, the APFD value of such test suite is given by

$$APFD = 1 - \frac{n_{ff1} + n_{ff2} + \ldots + n_{ffm}}{Nm} + \frac{1}{2N} \tag{3.9}$$

3.5 Single Fault Results (IG)

Question 1: Fault Localization Performance

We first compare the effectiveness of the IG heuristic in a single-fault setting, where the exact heuristic value can be calculated. Tables 3.5, 3.6, and 3.7 shows the score values for each program and heuristic with simulated failures. The value marked in **bold** indicates the best performing technique or techniques with a 95% confidence according to the Bonferroni mean separation test. To further illustrate our results, Figure 3.6 shows the evolution of C_d with respect to the number of executed tests C_t, per program, for the $\bar{h} = 0.5$ scenario.

In the synthetic and simulated cases, IG is consistently better than any other technique for every program. No other technique achieves the improvement rate of IG. Improvements range from 17% for `tcas` to 84% for `gzip`, with an average 57.3% improvement in effectiveness. In our experiments we observe how ADDST, and especially FEP, are the worst in terms of C_d evolution, being even worse than RND. However, this would seem to contradict previous literature [Jiang et al., 2009b], where random sequences were generally worse than ADDST except for one case. The reason is that the work in [Jiang et al., 2009b] was analyzing the performance exclusively for the small set of faults present in the Siemens set for each program. As pointed out in [Rothermel et al., 2001], the original faults in Siemens are located in hard to reach areas (i.e., covered only by a few tests), which favors ADDST and FEP. However, in our simulations faults can be *anywhere* in the program's code, eliminating the bias that favors ADDST and FEP.

	Single-fault Bayes, Synthetic								
$M \times N, \rho$	RND	ART		ADDST		FEP		IG	
$100 \times 250, \rho = 0.5$	1.15	1.21	+5%	1.21	+6%	1.21	+6%	**1.02**	**-11%**
$100 \times 250, \rho = 0.9$	2.76	2.52	-9%	3.59	+30%	4.16	+51%	**1.57**	**-43%**
$500 \times 250, \rho = 0.5$	1.15	1.16	+1%	1.16	+1%	1.19	+4%	**1.09**	**-6%**
$500 \times 250, \rho = 0.9$	2.77	2.56	-7%	3.08	+11%	3.13	+13%	**2.06**	**-26%**

Table 3.5: Single-fault S performance results, synthetic matrices

	Single-fault Bayes, Simulated, $M_f = 1$								
Program	RND	ART		ADDST		FEP		IG	
`print_tokens`	2.35	1.80	-23%	6.60	+181%	5.58	+138%	**1.01**	**-57%**
`print_tokens2`	2.25	1.85	-18%	3.90	+73%	5.25	+133%	**1.31**	**-42%**
`replace`	1.72	1.55	-9%	2.15	+25%	2.50	+45%	**0.94**	**-45%**
`schedule`	2.49	2.24	-10%	3.00	+21%	13.41	+439%	**0.85**	**-66%**
`schedule2`	3.73	3.35	-10%	6.35	+70%	13.15	+253%	**1.14**	**-69%**
`tcas`	3.79	3.88	+2%	5.81	+53%	5.07	+34%	**3.14**	**-17%**
`tot_info`	2.34	2.40	+3%	2.28	-2%	5.49	+135%	**1.23**	**-47%**
`space`	4.25	3.01	-29%	2.17	-49%	1.96	-54%	**1.53**	**-64%**
`gzip`	5.85	2.82	-52%	1.17	-80%	1.15	-80%	**0.92**	**-84%**
`sed`	3.67	2.61	-29%	1.30	-65%	1.35	-63%	**1.11**	**-70%**
`grep`	3.01	3.04	+1%	1.72	-43%	1.57	-48%	**1.49**	**-51%**
`flex`	3.43	1.57	-54%	1.21	-65%	1.20	-65%	**0.83**	**-76%**

Table 3.6: Single-fault S performance results, simulated faults

	Single-fault Bayes, Real, $M_f = 1$								
Program	RND	ART		ADDST		FEP		IG	
`print_tokens`	3.22	2.18	-32%	10.29	+220%	14.70	+356%	**1.03**	**-68%**
`print_tokens2`	2.53	**1.60**	**-37%**	7.88	+212%	15.71	+521%	1.75	-31%
`replace`	2.39	2.18	-9%	3.76	+57%	4.94	+107%	**2.13**	**-11%**
`schedule`	2.84	2.63	-7%	6.84	+141%	18.33	+545%	**2.37**	**-17%**
`schedule2`	2.81	2.42	-14%	9.56	+240%	16.18	+475%	**1.70**	**-40%**
`tcas`	4.55	**3.68**	**-19%**	9.22	+103%	10.27	+126%	11.18	146%
`tot_info`	2.75	**2.06**	**-25%**	4.16	+51%	9.42	+242%	2.87	+4%
`space`	5.98	4.52	-24%	**3.48**	**-42%**	3.92	-34%	3.90	-35%
`gzip`	7.17	3.61	-50%	1.51	-79%	1.50	-79%	**1.38**	**-81%**
`sed`	5.11	2.51	-51%	**1.30**	**-75%**	**1.32**	**-74%**	1.57	-69%
`grep`	4.22	3.44	-18%	**2.64**	**-38%**	2.94	-30%	2.97	-30%
`flex`	3.12	1.36	-56%	**0.84**	**-73%**	**0.86**	**-72%**	**0.88**	**-72%**

Table 3.7: Single-fault S performance results, real faults

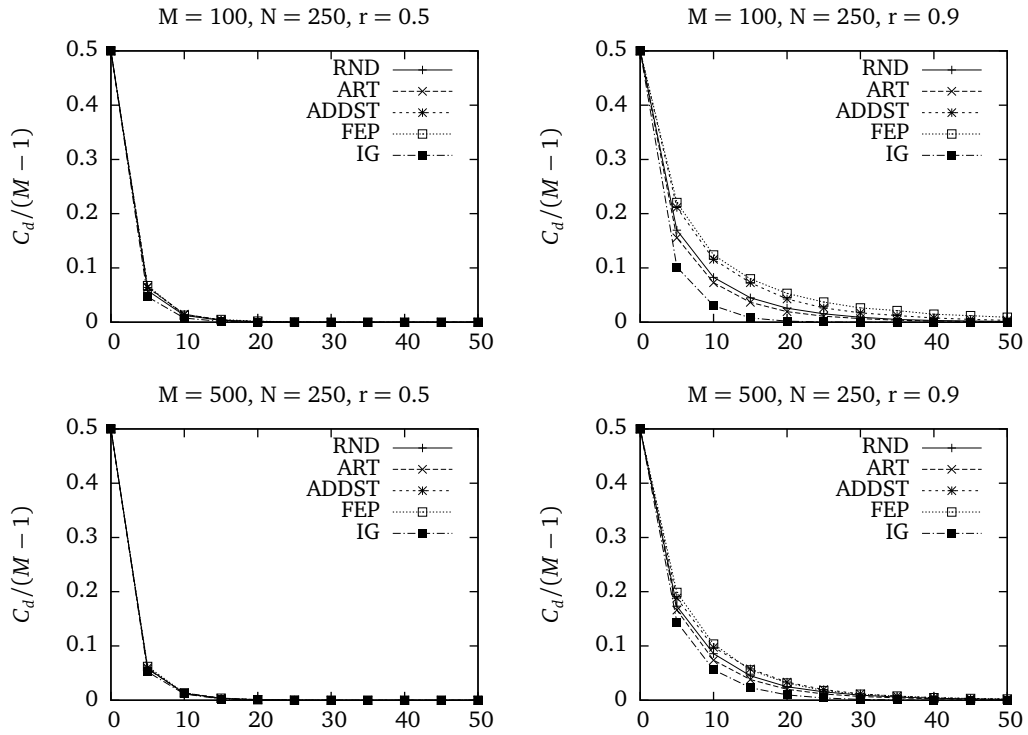


Figure 3.6: $C_d/(M-1)$ evolution for synthetic matrices

Our experimental results are consistent with our hypothesis in Section 3.2, that maximizing failure probability hinders the improvement of C_d because it causes the execution of tests that cover an excessive number of components.

In most cases, the order created by ART is better than RND because it chooses tests always at a certain distance to the already applied ones. By doing this, the chance of choosing a test that bisects the current set of diagnostic candidates increases, providing a slight advantage over RND.

Apparently contradicting our predictions, the performance of ADDST and FEP is actually better than RND for the larger SIR programs. The answer to this apparent contradiction resides in the shape and distribution of statement coverage in the test matrix A, and will be discussed in Chapter 5.

In summary, based on the plots, and the statistical analysis, we conclude that IG is most suitable for the purpose of fault localization, given its performance and robustness to test matrices of differing sizes and shapes and false negative rates. In the next section we will see how this implies a trade-off with failure detection.

	Single-fault Bayes, Synthetic							
$M \times N, \rho$	RND	ART		ADDST		FEP		IG	
$100 \times 250, \rho = 0.5$	0.98	0.98	+0%	**0.99**	**+1%**	**0.99**	**+1%**	0.97	-1%
$100 \times 250, \rho = 0.9$	0.99	0.99	+0%	**1.00**	**+0%**	**1.00**	**+0%**	0.99	-0%
$500 \times 250, \rho = 0.5$	0.98	0.98	+0%	**0.99**	**+0%**	**0.99**	**+0%**	0.98	-0%
$500 \times 250, \rho = 0.9$	**1.00**	**0.99**	**-0%**	**1.00**	**+0%**	**1.00**	**+0%**	0.99	-0%

Table 3.8: Single-fault APFD results, synthetic matrices

	Single-fault Bayes, Simulated								
Program	RND	ART		ADDST		FEP		IG	
`print_tokens`	0.95	0.97	+2%	**1.00**	**+5%**	**1.00**	**+5%**	0.99	+4%
`print_tokens2`	0.97	0.98	+1%	0.98	+2%	**0.99**	**+3%**	0.98	+2%
`replace`	0.93	0.94	+1%	**0.98**	**+5%**	**0.99**	**+6%**	0.97	+4%
`schedule`	0.97	0.98	+1%	**0.99**	**+2%**	**0.99**	**+2%**	0.98	+0%
`schedule2`	0.99	0.99	+0%	**1.00**	**+0%**	**1.00**	**+1%**	0.99	-0%
`tcas`	0.89	0.92	+3%	**0.93**	**+4%**	**0.94**	**+5%**	**0.93**	**+4%**
`tot_info`	0.98	0.97	-1%	**0.98**	**+1%**	**0.98**	**+1%**	0.97	-0%
`space`	0.74	0.79	+6%	0.88	+18%	**0.91**	**+22%**	**0.90**	**+21%**
`gzip`	0.70	0.82	+17%	**0.94**	**+34%**	**0.94**	**+34%**	**0.94**	**+33%**
`sed`	0.81	0.88	+8%	**0.93**	**+15%**	**0.94**	**+15%**	**0.93**	**+15%**
`grep`	0.84	0.86	+2%	**0.92**	**+9%**	**0.93**	**+11%**	**0.91**	**+9%**
`flex`	0.97	0.99	+1%	**0.99**	**+2%**	**0.99**	**+2%**	**0.99**	**+2%**

Table 3.9: Single-fault APFD results, simulated faults

	Single-fault Bayes, Real								
Program	RND	ART		ADDST		FEP		IG	
`print_tokens`	0.94	0.96	+3%	**0.98**	**+5%**	0.96	+2%	**0.97**	**+4%**
`print_tokens2`	0.94	0.96	+3%	**0.98**	**+5%**	**0.99**	**+6%**	0.95	+2%
`replace`	0.90	0.90	+0%	0.88	-2%	**0.92**	**+3%**	**0.92**	**+2%**
`schedule`	**0.95**	**0.95**	**-0%**	0.91	-4%	**0.94**	**-1%**	0.93	-3%
`schedule2`	**0.92**	**0.92**	**+0%**	**0.94**	**+2%**	0.91	-2%	0.89	-3%
`tcas`	**0.82**	**0.84**	**+2%**	0.80	-3%	0.78	-5%	0.79	-4%
`tot_info`	0.98	0.98	+0%	**0.99**	**+1%**	**0.98**	**+1%**	0.98	+1%
`space`	0.69	0.76	+10%	**0.79**	**+15%**	**0.78**	**+14%**	**0.77**	**+12%**
`gzip`	0.76	0.85	+13%	**0.96**	**+27%**	**0.95**	**+26%**	**0.95**	**+25%**
`sed`	0.76	0.91	+20%	**0.94**	**+24%**	**0.94**	**+25%**	**0.93**	**+23%**
`grep`	0.76	**0.82**	**+8%**	**0.84**	**+10%**	**0.81**	**+7%**	0.79	+4%
`flex`	0.94	**0.98**	**+4%**	**0.98**	**+4%**	**0.98**	**+4%**	**0.98**	**+4%**

Table 3.10: Single-fault APFD results, real faults

Question 2: Failure Detection Performance

Table 3.8, 3.9, and 3.10 shows the averaged APFD scores for each heuristic. The value marked in **bold** indicates the best performing technique or techniques with a 95% confidence according to the Bonferroni mean separation test.

Failure detection performance can be explained analytically by modeling the number of tests that need to be executed until the first failure occurs, n_{ff}, by a geometric distribution, $X \sim Geo(p)$, whose expected value is $E[X] = p^{-1}$. The objective of FEP and ADDST is to choose tests with maximum failure probability, ideally $p = 1.0$. Therefore approximately 1 test is needed on average ($n_{ff} \approx 1$). On the other hand, IG tends to select test cases that balance the probability of passing and failing, ideally $p = 0.5$, and therefore on average needs 2 tests ($n_{ff} = 2$).

It can be clearly seen how ADDST, and especially FEP, are the best performing techniques. This is expected, as the assumptions under which FEP and ADDST were devised, i.e., maximization of failure probability, are the most favorable. The failure detection performance of IG is lower than FEP and ADDST and slightly higher than ART and with a lower dispersion. ART has a better performance than random and a lower dispersion, consistent with [Jiang et al., 2009a]. Again, this is caused by the coverage distance kept between each test.

The APFD performance of IG for the SIR programs is similar to FEP. This is again caused by the shape and sparsity of the matrices, and will be discussed in Chapter 5.

In summary, when considering early failure detection as the main goal, FEP and ADDST are more suitable for this purpose than IG. This result is consistent with previous literature [Rothermel et al., 2001]. However, given the small difference with IG, and the fact that for the large programs and very high FNR scenarios IG behaves essentially like FEP, IG is a suitable alternative. Furthermore, in the cases where FEP and ADDST clearly outperform IG, the trade-off of APFD for C_d is compensated greatly, as we will see in the next section.

3.6 Multiple Faults (Sequoia)

Tables 3.11 to 3.16 show the results for $M_f = 1$, and $M_f = 3$, for (1) synthetic matrices, (2) simulated test outcomes, and (3) actual outcomes (according to the test suite's test oracle), respectively. In this section we will use the Sequoia diagnostic prioritization algorithm for multiple faults instead of the single-fault IG algorithm, and compare the results of Sequoia with and without fall-back.

For the synthetic and simulated failure cases, Sequoia clearly improves over all other prioritization techniques. However, this is not always true when the real test case outcomes are used. For $M_f = 3$ we see how the improvements achieved by Sequoia are consistently lower than in the simulations, and how in the real cases of `tcas` and `tot_info` Sequoia performs worse. In these cases the estimation

of the FNR is poor and causes SEQUOIA to make wrong decisions. Since RND is able to attain a good final C_d, these two cases highlight the fact that sequencing performance (the IG heuristic) is affected much more by FNR estimation errors than the Bayesian update itself. We can see how the simple fall-back mechanism does have the potential to solve FNR estimation errors, however, since our fall-back criterion is not very refined, in some cases the fall-back happens too early, reducing the effectiveness with respect to plain SEQUOIA.

The improvement achieved by SEQUOIA in our simulations in real matrices reach 81% for the $M_f = 1$ case and 79% for the $M_f = 3$ case, both in the `gzip` program.

In the case with real matrices, similar improvements are possible, for example, we can see that `schedule2` is a very favorable case, with a 58% reduction for $M_f = 1$ and 77% for $M_f = 3$. For `tcas`, on the other hand, things are much worse because D contains too large a quantity of equal minimal diagnoses (so-called ambiguity groups) that have relatively high posterior probability. Even the 1st-order expansion of D cannot avoid a substantial loss of information that causes frequent underestimation of the failure probabilities. As a result, SEQUOIA will erroneously compensate by selecting tests that include too many components, causing loss of diagnostic power. This situation is easily detected by fall-back-enabled SEQUOIA, and switching at this point from IG to RND mode completely solves this performance problem.

3.7 Related Work

The influence of test-suite extension, reduction, modification, and prioritization on fault detection and diagnosis has received considerable attention [Yu et al., 2008, Baudry et al., 2006, Hao et al., 2005]. In particular, Jiang *et al.* [Jiang et al., 2009b] show how some prioritization techniques are worse than random sequencing and that those which are better, do not provide a significant improvement. In contrast to the works cited above, which use coverage-based heuristics to reduce the size of the test suite, SEQUOIA employs an heuristic to select the best test case to optimize diagnostic accuracy.

Test case prioritization is a mature and active area of research whose most common goal is to increase failure detection rate, which dates back to [Harrold et al., 1993b, Wong et al., 1997]. The failure detection effectiveness of different coverage-based prioritization techniques was studied by Rothermel *et al.* [Elbaum et al., 2002], who also proposed the FEP heuristic [Rothermel et al., 2001] that has already been discussed elsewhere in the chapter. Other work on prioritization includes [Elbaum et al., 2004, Li et al., 2007, Rothermel et al., 2001]. Jiang *et al.* [Jiang et al., 2009a] propose a hybrid random and coverage-based prioritization technique (ART) which has already been discussed throughout this chapter. SEQUOIA fundamentally differs from traditional prioritization techniques in that its main goal is not to optimize the rate of failure detection, but to minimize debug-

	Bayes, Synthetic, $M_f = 1$									
$M \times N, \rho$	RND	ART		ADDST		FEP		SEQUOIA	
$100 \times 250, \rho = 0.5$	1.24	1.20	-3%	1.29	+4%	1.28	+4%	**1.09**	**-12%**
$100 \times 250, \rho = 0.9$	3.03	2.76	-9%	3.87	+28%	4.45	+47%	**1.71**	**-44%**
$500 \times 250, \rho = 0.5$	1.28	1.25	-3%	1.31	+2%	1.34	+4%	**1.18**	**-8%**
$500 \times 250, \rho = 0.9$	3.27	2.95	-10%	3.59	+10%	3.69	+13%	**2.42**	**-26%**

Table 3.11: Multiple-fault S performance results, synthetic matrices, 1 fault

	Bayes, Simulated, $M_f = 1$											
Prog.	RND	ART		ADDST		FEP		SEQUOIA		FALLBACK	
pt	2.58	1.88	-27%	6.68	+159%	6.23	+141%	**1.13**	**-56%**	1.33	-49%
p2	2.29	1.96	-15%	4.16	+81%	6.63	+189%	**1.53**	**-33%**	1.60	-30%
re	1.97	1.86	-6%	2.52	+28%	3.05	+55%	**1.21**	**-39%**	**1.26**	**-36%**
sc	2.73	2.52	-8%	3.46	+27%	15.14	454%	**1.29**	**-53%**	1.38	-50%
s2	3.97	3.66	-8%	6.58	+66%	13.83	249%	**1.50**	**-62%**	2.03	-49%
tc	4.07	3.91	-4%	5.45	+34%	5.32	+31%	**3.19**	**-22%**	3.67	-10%
ti	2.55	2.65	+4%	2.59	+2%	5.86	+130%	**1.40**	**-45%**	**1.44**	**-43%**
sp	4.89	3.52	-28%	2.66	-46%	2.51	-49%	**1.91**	**-61%**	3.35	-32%
gz	6.31	3.08	-51%	1.37	-78%	1.32	-79%	**1.18**	**-81%**	3.05	-52%
se	4.21	3.04	-28%	**1.66**	**-61%**	1.77	-58%	1.83	-56%	2.02	-52%
gr	3.91	3.82	-2%	2.21	-43%	**2.11**	**-46%**	**2.16**	**-45%**	3.12	-20%
fl	5.79	2.58	-56%	1.65	-71%	1.81	-69%	**1.46**	**-75%**	1.53	-74%

Table 3.12: Mutiple-fault S performance results, simulated faults, 1 fault

	Bayes, Real, $M_f = 1$											
Prog.	RND	ART		ADDST		FEP		SEQUOIA		FALLBACK	
pt	3.79	2.62	-31%	10.12	167%	13.00	243%	**1.60**	**-58%**	2.48	-35%
p2	3.23	1.83	-43%	6.46	+100%	14.23	340%	**1.51**	**-53%**	1.96	-39%
re	2.70	**1.88**	**-30%**	3.87	+43%	3.99	+48%	2.88	+7%	2.51	-7%
sc	1.77	2.20	+24%	7.59	+328%	11.99	576%	1.89	+6%	**1.37**	**-23%**
s2	3.16	2.77	-12%	8.66	+174%	11.70	271%	1.77	-44%	**1.37**	**-57%**
tc	3.34	2.73	-18%	9.16	+174%	10.83	224%	10.36	210%	**1.78**	**-47%**
ti	**1.28**	**1.32**	**+3%**	6.38	+401%	8.52	+568%	3.88	+205%	2.12	+66%
sp	7.13	5.77	-19%	**2.51**	**-65%**	3.25	-54%	3.06	-57%	6.12	-14%
gz	8.08	4.06	-50%	**1.54**	**-81%**	1.75	-78%	2.24	-72%	5.68	-30%
se	5.72	3.00	-48%	2.04	-64%	2.23	-61%	**1.93**	**-66%**	3.48	-39%
gr	6.21	3.86	-38%	**3.04**	**-51%**	3.57	-43%	3.91	-37%	5.49	-12%
fl	3.40	1.69	-50%	**1.09**	**-68%**	**1.12**	**-67%**	1.41	-59%	1.24	-63%

Table 3.13: Mutiple-fault S performance results, real faults, 1 fault

	Bayes, Synthetic, $M_f = 3$									
$M \times N, \rho$	RND	ART		ADDST		FEP		SEQUOIA		
$100 \times 250, \rho = 0.5$	12.97	11.08	-15%	15.68	+21%	15.15	+17%	**9.52**	**-27%**	
$100 \times 250, \rho = 0.9$	19.35	16.86	-13%	22.03	+14%	20.79	+7%	**13.11**	**-32%**	
$500 \times 250, \rho = 0.5$	16.32	14.96	-8%	17.20	+5%	17.04	+4%	**14.12**	**-14%**	
$500 \times 250, \rho = 0.9$	14.91	13.95	-6%	15.52	+4%	14.45	-3%	**12.03**	**-19%**	

Table 3.14: Multiple-fault S performance results, synthetic matrices, 3 faults

	Bayes, Simulated, $M_f = 3$											
Prog.	RND	ART		ADDST		FEP		SEQUOIA		FALLBACK		
pt	8.91	6.00	-33%	17.40	+95%	11.01	+24%	**2.38**	**-73%**	4.41	-51%	
p2	11.68	8.31	-29%	15.76	+35%	19.26	+65%	**7.48**	**-36%**	8.53	-27%	
re	15.50	10.65	-31%	12.21	-21%	22.56	+46%	**6.53**	**-58%**	8.65	-44%	
sc	22.40	20.23	-10%	22.53	+1%	41.20	+84%	**6.84**	**-69%**	13.02	-42%	
s2	24.56	23.21	-6%	32.00	+30%	39.64	+61%	**6.49**	**-74%**	14.81	-40%	
tc	6.63	4.86	-27%	10.87	+64%	12.11	+83%	4.37	-34%	**4.11**	**-38%**	
ti	11.81	9.09	-23%	9.99	-15%	27.88	136%	**4.67**	**-60%**	6.49	-45%	
sp	11.78	8.63	-27%	8.37	-29%	8.58	-27%	**7.58**	**-36%**	8.18	-31%	
gz	12.27	6.07	-51%	3.35	-73%	2.63	-79%	**2.52**	**-79%**	2.92	-76%	
se	9.05	5.52	-39%	6.10	-33%	5.57	-38%	**4.15**	**-54%**	4.70	-48%	
gr	13.96	8.76	-37%	8.22	-41%	**7.55**	**-46%**	7.72	-45%	9.56	-32%	
fl	7.07	3.22	-54%	2.24	-68%	2.93	-59%	**1.62**	**-77%**	2.42	-66%	

Table 3.15: Mutiple-fault S performance results, simulated faults, 3 faults

	Bayes, Real, $M_f = 3$											
Prog.	RND	ART		ADDST		FEP		SEQUOIA		FALLBACK		
pt	6.09	6.75	+11%	26.07	328%	27.76	356%	5.87	-4%	**3.16**	**-48%**	
p2	8.56	6.51	-24%	21.09	146%	27.77	224%	5.74	-33%	**5.58**	**-35%**	
re	10.95	8.84	-19%	10.36	-5%	11.79	+8%	**5.12**	**-53%**	6.50	-41%	
sc	13.07	9.25	-29%	9.72	-26%	21.83	+67%	**5.81**	**-56%**	7.27	-44%	
s2	11.78	9.07	-23%	9.79	-17%	19.81	+68%	**2.70**	**-77%**	4.56	-61%	
tc	4.44	3.94	-11%	13.16	196%	14.88	235%	12.53	182%	**3.70**	**-17%**	
ti	3.00	**2.20**	**-27%**	3.65	+22%	10.06	235%	4.43	+48%	2.60	-13%	
sp	12.48	10.04	-20%	7.48	-40%	**6.64**	**-47%**	8.33	-33%	9.78	-22%	
gz	14.07	7.14	-49%	2.03	-86%	**1.95**	**-86%**	3.69	-74%	7.48	-47%	
se	10.88	6.49	-40%	**3.40**	**-69%**	4.62	-58%	3.82	-65%	5.93	-46%	
gr	10.78	8.22	-24%	**3.68**	**-66%**	3.77	-65%	6.52	-40%	8.18	-24%	
fl	9.90	5.04	-49%	3.63	-63%	4.09	-59%	**3.26**	**-67%**	3.64	-63%	

Table 3.16: Mutiple-fault S performance results, real faults, 3 faults

ging cost. Cost-cognizant test prioritization techniques have been proposed [Smith and Kapfhammer, 2009, Elbaum et al., 2001, Zhang et al., 2009] to address the cost variability problem. Those techniques attempt to maximize failure probability (as FEP does), therefore their performance for Diagnostic Prioritization is expected to be lower than random prioritization. To the best of our knowledge, no cost-aware version of ART exists at the moment.

Automated fault-localization techniques also aim at minimizing diagnostic cost when failures occur during the testing phase. Statistical approaches include [Abreu et al., 2009b, Jones et al., 2002, Liblit, 2008, Liu et al., 2005, Renieris and Reiss, 2003, Wong et al., 2008]. A recent, probabilistic approach of acceptable complexity is [Abreu et al., 2009a]. Unlike our dynamic approach, all the above approaches do not address test sequencing to optimize diagnostic accuracy, and are essentially similar to RND. Other approaches to fault localization include [Baah et al., 2008, Santelices et al., 2009, Wang et al., 2009].

Sequential diagnosis aims at finding the test sequence that optimizes diagnostic performance based on the current test outcomes. Sequential diagnosis is often applied to hardware systems where tests can produce false negatives [Raghavan et al., 1999]. Typically, only one fault is assumed present in the system. If the system can have multiple faults, sequential diagnosis complexity becomes exponential. In [Shakeri et al., 2000] an approximation is described for systems with permanent faults ($h = 0$), an assumption which does not hold for most software systems. SEQUOIA differs in that it employs a sequential approach for multiple faults and tests with false negatives, at polynomial complexity due to our low-cost heuristics instead of the Bayesian approach.

To the best of our knowledge our diagnostic approach to test prioritization has not be described before. A preliminary version of our work is described in [Gonzalez-Sanchez et al., 2011e], that reveals the high potential of (IG-based) dynamic prioritization. However, to avoid the combinatorial explosion of Bayesian diagnosis, in that work we assumed single faults and unit costs, both of which are unrealistic for large systems.

3.8 SUMMARY

In this chapter, we have introduced a specific prioritization technique of test cases, dubbed diagnostic prioritization, that reduces the loss of diagnostic information to a minimum. Our experiments have shown that in terms of diagnostic information gain per test case, diagnostic prioritization is the best technique. This comes at the price of a slightly reduced APFD failure detection performance with respect to additional-coverage techniques, although not in all cases.

We have presented a spectrum-based sequential diagnosis approach for multiple faults coined SEQUOIA, that greedily selects tests out of a suite of tests to narrow down the set of diagnostic candidates with a minimum number of tests. SEQUOIA

performs on-line prioritization, adapting the next test case to be executed depending on the pass/fail information of the previously executed tests.

To achieve polynomial complexity SEQUOIA uses an approximate information gain computation approach. Synthetic data shows, that the dynamic selection of the next best test based on the observations made so far, allows SEQUOIA to achieve much better decay of diagnostic uncertainty compared to other test prioritization techniques. Real programs also show that SEQUOIA has better performance, except when the input intermittency parameters are erroneous or the diagnosis involves ambiguity sets that are too large for the entropy estimation to handle.

4

Off-line Diagnostic Prioritization

Algorithms that prioritize test cases to enhance the diagnostic information yield have been proposed, based on the information gain heuristic (IG). However they are not practical for the diagnosis of multiple faults, or require the precise estimation of the test false negative rate and are very sensitive to erroneous estimations.

In this chapter, we present RAPTOR*, a test prioritization algorithm for fault localization that reduces the similarity between component execution patterns as the testing progresses. Unlike previous diagnostic prioritization algorithms,* RAPTOR *does not require false negative information, is much less complex, and can work off-line.*

4.1 Raptor **56**
4.2 Empirical Evaluation **61**
4.3 Discussion **66**
4.4 Related Work **70**
4.5 Summary **71**

The approach to test prioritization presented in Chapter 3 is based on the information gain heuristic (IG), and, although in theory their diagnostic effectiveness is optimal [Gonzalez-Sanchez et al., 2010a], it has a number of drawbacks that restrict its applicability in practice. First, its performance depends heavily on the precise estimation of a number of parameters (fault density and false negative test rate) that are very expensive to obtain, as well as very error-prone. This has been shown to cause severe impact on the quality of IG-based techniques [Gonzalez-Sanchez et al., 2010a, Gonzalez-Sanchez et al., 2011e]. Second, the algorithmic complexity of IG is exponential, assuming that more than one fault may be present in the system. This is aggravated by the fact that, unlike traditional test prioritization techniques, in IG-based, diagnostic test prioritization the tests are selected on-line, based on the actual pass/fail results of previously executed tests (which varies per regression cycle). On-line calculations impose a large overhead in the test process if test cases are relatively short.

In this chapter we introduce a novel, off-line diagnostic test prioritization algorithm. In particular, our chapter makes the following contributions:

- We present RAPTOR (gReedy diAgnostic Prioritization by ambiguiTy grOup Reduction), a low-complexity diagnostic test prioritization algorithm that can be used off-line, which does not require any additional, input parameters.
- We evaluate the performance of RAPTOR for the Siemens set as well as for the larger programs `space`, `gzip`, `sed`, `grep`, and `flex`, available from the Software Infrastructure Repository (SIR) [Do et al., 2005]. All programs were extended to accommodate multiple faults.

In the above experiments we compare the performance of RAPTOR to the contemporary test prioritization algorithms ADDST [Rothermel et al., 2001], FEP [Rothermel et al., 2001], ART [Jiang et al., 2009a], the IG-based diagnostic test prioritization SEQUOIA [Gonzalez-Sanchez et al., 2010a], and to random prioritization (RND), our baseline.

To the best of our knowledge, a diagnostic approach to off-line test prioritization has not been described before. Our results show that RAPTOR can deliver a performance, in terms of residual diagnostic effort reduction per unit of test effort, superior to ADDST, FEP, ART, and RND, and even to the theoretical optimum, SEQUOIA. Improvements are, on average, 40% better with respect to the next best technique, and 55% better than the baseline (RND).

The chapter is organized as follows. In Section 4.1 our diagnosis-based approach to test case prioritization is presented. In Section 4.2 real programs are used to assess our technique. Section 4.3 discusses our results and the threats to their validity. We compare RAPTOR with related work in Section 4.4. In Section 4.5, we conclude and discuss future work.

4.1 RAPTOR

The IG heuristic requires very precise estimations of the FNR parameters, a costly and error-prone calculation. Experiments with real programs showed that errors in the FNR estimations can have disastrous results [Gonzalez-Sanchez et al., 2010a]. This makes the enormous investment in parameter estimation and the overhead incurred in the testing process totally useless.

For these reasons, we present an alternative diagnostic prioritization algorithm, RAPTOR (*gReedy diAgnostic Prioritization by ambiguiTy grOup Reduction*), that, while not being theoretically optimal, it is close enough to produce competitive results. Furthermore, since it is not affected by FNR estimation errors, its results are in many cases better than IG.

Ambiguity Groups

Not all the components in a system can be executed separately from each other. First, they may be too tightly coupled and are always executed together, e.g., all

	Program: Character Counter	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8
c_1	`main() {`	1	1	1	1	1	1	1	1
c_2	`  int let, dig, other, c;`	1	1	1	1	1	1	1	1
c_3	`  let = dig = other = 0;`	1	1	1	1	1	1	1	1
c_4	`  while(c = getchar()) {`	1	1	1	1	1	1	1	1
c_5	`   if ('A'<=c && 'Z'>=c)`	1	1	1	1	1	1	1	0
c_6	`    let += 2; // FAULT`	1	0	1	1	1	0	1	0
c_7	`   elsif ('a'<=c && 'z'>=c)`	1	1	0	1	1	1	1	0
c_8	`    let += 1;`	1	0	0	0	1	0	1	0
c_9	`   elsif ('0'<=c && '9'>=c)`	1	1	0	1	1	1	0	0
c_{10}	`    dig += 1;`	1	1	0	1	0	0	0	0
c_{11}	`   elsif (isprint(c))`	0	1	0	0	1	1	0	0
c_{12}	`    other += 1;`	0	0	0	0	1	1	0	0
c_{13}	`  printf("%d %d %d\n",` `  let, dig, others);}`	1	1	1	1	1	1	1	1
	Test case outcomes	1	0	1	1	1	0	1	0

Inputs for test cases: t_1: `"Al84"`, t_2: `"1337"`, t_3: `"ABC"`, t_4: `"UB40"`, t_5: `"#CheezBurger"`, t_6: `"\0x10\0x05\0x00"`, t_7: `"Alberto"`, t_8: `""`

Table 4.1: Example program and diagnosis inputs

statements inside a basic block of code. Second, there may simply be no test in the test suite that separates the execution traces of two components even when it is possible. Any individual component belonging to a group of components with identical columns (signatures) can never be uniquely identified as faulty. Such group is termed *ambiguity group* [Stenbakken et al., 1989] or *dynamic basic block* [Baudry et al., 2006]. The example system in Table 4.1 has a large ambiguity group formed by the first 4 statements (initialization and loop guard) and the last statement (output). Should a fault occur in any of those statements, it would be impossible to pinpoint that fault with total precision.

Since the basic working principle of any diagnostic algorithm is analyzing the differences and similarities between the signatures of each of the components of the system under test (each of the columns in the test coverage matrix A) and the failure pattern of the system (the test outcomes o_i), we can use the evolution of these differences to devise a new, simpler diagnostic test prioritization heuristic. A numerical measurement for the ambiguity of a test matrix can be defined based on the partition of ambiguity groups it generates, $AG = \{g_1, g_2, \ldots, g_L\}$. It is based on taking into account the probability of the system containing a fault belonging to each of the ambiguity groups, times the expected residual diagnostic effort of each group. Assuming faults are distributed uniformly through the program's code, the probability of group g_i occurring is $\Pr(g_i) = \frac{|g_i|}{M}$, and the residual diagnostic effort in such group corresponds to $C_d = \frac{|g_i|-1}{2}$. The value of the diagnostic ambiguity is

then

$$\mathrm{G}(AG) = \sum_{i=1}^{L} \frac{|g_i|}{M} \cdot \frac{|g_i| - 1}{2} \tag{4.1}$$

and can be seen as the expected (i.e., average) value of the residual diagnostic effort once the tests in A have been applied, assuming the correct group will always rank at top. The matrix in Table 4.1 produces the partition $AG = \{\{5\}, \{6\}, \{7\}, \{8\}, \{9\}, \{10\}, \{11\}, \{12\}, \{1, 2, 3, 4, 13\}\}$ whose ambiguity is $G(AG) = 8 \cdot (\frac{1}{13} \cdot \frac{0}{2}) + \frac{5}{13} \cdot \frac{4}{2} = 0.77$.

Since G(AG) is an estimation of the residual diagnostic effort C_d, the evolution, test by test, of the value of G(AG) can be used as alternative heuristic to IG. For our example system, $AG = \{\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13\}\}$, and G($AG$) = 6.00, when no test case has been executed. Every test that is executed breaks each ambiguity group into two new ambiguity groups, one corresponding to the components in the ambiguity group that are covered by the test, and one corresponding to the components that are not covered. If we execute test t_1, the ambiguity group will be broken to $AG = \{\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 13\}, \{11, 12\}\}$ (only 2 statements are left uncovered), with G(AD) = 4.31. On the other hand, if we execute t_3, $AG = \{\{1, 2, 3, 4, 5, 6, 13\}, \{7, 8, 9, 10, 11, 12\}\}$ and G(AG) = 2.77, a much larger reduction.

Algorithm

Since a large reduction in the ambiguity G(AG) leads to a proportional reduction of the residual diagnostic effort C_d, it can be used as heuristic to select test cases in a diagnostic test prioritization algorithm. Using G(AG) has the advantage that it does not require knowing the test outcomes, i.e., can be performed off-line, nor requires estimation of additional input parameters, e.g., FNR.

Algorithm 5 details the implementation of such algorithm, that computes a sequence that optimizes the evolution of G(AG) in a greedy, incremental way.

At every iteration, a test t_i is selected, such that it produces a new set NG that maximizes the reduction of ambiguity. The BreakUp function takes a set of ambiguity groups G, and applies a test t to produce a new ambiguity group partition, NG. The greedy search heuristic calculates the difference in ambiguity between G and NG, using the Amb function. To account for the fact that tests typically have different costs, the reduction of ambiguity is divided by the cost of such reduction, in a spirit similar to [Elbaum et al., 2001]. The test that produces the highest improvement in ambiguity is concatenated to the result (operator ||) and removed from the set of free tests. Since the algorithm may reach a point where no test can produce an improved NG set, the algorithm checks for this case and, if detected, resets the G set and starts over again with the remaining tests. This policy is also used in [Rothermel et al., 2001]. An alternative policy also proposed in [Rothermel et al., 2001] would be to simply randomly order the remaining tests.

Algorithm 5 RAPTOR test prioritization

```
function RAPTOR(A, M, N, C)
    free ← {1, 2, ..., N}
    res ← ∅
    AG ← {{1, 2, ..., M}}
    while free ≠ ∅ do
        next, AGmax, Hmax ← 0, ∅, 0
        for i ∈ free do
            NG ← BREAKUP(AG, A[i, ∗])
            H ← (AMB(AG, M) - AMB(NG, M)) / C[i]
            if H > Hmax then
                next, AGmax, Hmax ← i, NG, H
        if i ≠ 0 then                          ▷ Found a breaking test
            AG ← AGmax
            res ← res||next
            free ← free \ {next}
        else                                   ▷ Reset
            AG ← {{1, 2, ..., M}}
    return res

function BREAKUP(AG, t)
    NG ← ∅
    for g_i ∈ AG do
        C ← ∅                                  ▷ Split AG for the covered
        NC ← ∅                                 ▷ Split AG for the not covered
        for j ∈ g_i do
            if t[j] == 1 then                  ▷ Covered by the test?
                C ← C ∪ {j}
            else
                NC ← NC ∪ {j}
        NG ← NG ∪ {C, NC}
    return NG

function AMB(AG, M)
    sum ← 0.0
    for g_i ∈ AG do
        sum ← sum + (|g_i| / M) · ((|g_i| − 1) / 2)
    return sum
```

In RAPTOR, the BREAKUP function calculates the new partition of ambiguity groups incrementally. Each ambiguity group in the partition is split in two sets, one for the covered components, C, and one for the not covered, NC, which are added to the resulting partition NG.

Example

RAPTOR would prioritize the tests in our example test suite as follows. As shown already, the first test choice is t_3 producing

$$AG = \{\{1,2,3,4,5,6,13\},\{7,8,9,10,11,12\}\}$$

with ambiguity G(AG) = 2.77. This is the same test that IG chooses.

Table 4.2 shows the sequence of tests chosen by RAPTOR together with the evolution of the partition of ambiguity groups and the value for the diagnostic ambiguity. RAPTOR chooses t_2 as second test, unlike IG in Section 2.1 which would choose t_8.

i	Ambiguity Groups	G(AG)
	$\{\{1,2,3,4,5,6,7,8,9,10,11,12,13\}\}$	6.00
3	$\{\{1,2,3,4,5,6,13\},\{7,8,9,10,11,12\}\}$	2.77
2	$\{\{1,2,3,4,5,13\},\{6\},\{7,9,10,11\},\{8,12\}\}$	1.69
8	$\{\{1,2,3,4,13\},\{5\},\{6\},\{7,9,10,11\},\{8,12\}\}$	1.31
7	$\{\{1,2,3,4,13\},\{5\},\{6\},\{7\},\{8\},\{9,10,11\},\{12\}\}$	1.00
5	$\{\{1,2,3,4,13\},\{5\},\{6\},\{7\},\{8\},\{9,11\},\{10\},\{12\}\}$	0.84
4	$\{\{1,2,3,4,13\},\{5\},\{6\},\{7\},\{8\},\{9\},\{10\},\{11\},\{12\}\}$	0.77
	$\{\{1,2,3,4,5,6,7,8,9,10,11,12,13\}\}$ (reset)	6.00
6	$\{\{1,2,3,4,5,7,9,11,12,13\},\{6,8,10\}\}$	3.69
1	$\{\{1,2,3,4,5,7,9,13\},\{6,8,10\},\{11,12\}\}$	2.46

Table 4.2: Sequence generated by RAPTOR

Complexity and Timing Information

The function BREAKUP presents $O(M)$ time complexity since the innermost loop will be iterated M times in total. We assume that set union can be done in $O(1)$ time (the set can be implemented as a linked list since only the concatenation operation will be used) from the previous one. The AMB function can be calculated in $O(M)$ time since there will be at most M ambiguity groups. Since these two functions have to be repeatedly used for each of the remaining free tests, the complexity per text choice, i.e., for the innermost loop of RAPTOR equals $O(MN)$. A complexity similar to ADDST prioritization. Execution time measurements confirm our complexity analysis, as can be observed in Table 4.3. The timing experiment was performed using an 8-core 2.6 GHz. Xeon machine with 32Gb of memory, with matrices generated randomly with coverage density $\rho = 0.5$, i.e., coverage probability of 0.5.

	$M = 10$	100	1,000
$N = 10$	0.0002	0.0005	0.0014
100	0.0015	0.0075	0.1047
1,000	0.0132	0.0761	1.0849

Table 4.3: Time per test selection (seconds)

4.2 Empirical Evaluation

In this section we compare the performance of Raptor to RND, ART, ADDST, FEP, and Sequoia. The prioritization performance of each algorithm is assessed by measuring the evolution of the residual diagnostic effort in the diagnosis. To strengthen the validity of our results we will use two different diagnostic techniques: the well-known similarity coefficient Ochiai, and Bayesian diagnosis as presented in Section 2.1. We opt for externally-derived FNR information as done in [Gonzalez-Sanchez et al., 2011a] since it provides the best diagnostic quality. Finally, we also study the trade off between fault detection and fault localization made by Raptor.

Subject Programs

We perform our experiments on the well-known Siemens benchmark set [Hutchins et al., 1994], as well as the `space`, `gzip`, `sed`, `grep`, and `flex` programs (obtained from SIR [Do et al., 2005]). Every program has a correct version, and a set of test inputs is also provided, which were created with the intention of providing full test coverage. Table 4.4 provides more information about the programs used in the experiments, where M corresponds to the number of lines of code (components at statement granularity).

	Program Name	Mutants	M	N	Program Type
pt	print_tokens	491	539	4,130	Lexical Analyzer
p2	print_tokens2	294	489	4,115	Lexical Analyzer
re	replace	757	507	5,542	Pattern Recognition
sc	schedule	281	397	2,650	Priority Scheduler
s2	schedule2	212	299	2,710	Priority Scheduler
tc	tcas	208	174	1,608	Altitude Separation
ti	tot_info	396	398	1,052	Information Measure
sp	space	6,283	9,126	500	ADL Parser
gz	gzip	20,811	6708	211	Compressor
se	sed	11,050	9014	184	Stream Editor
gr	grep	17,707	13,287	809	String Matching
fl	flex	15,574	14,194	107	Lexer Generator

Table 4.4: Programs used for evaluation

To obtain a large fault sample size, we have extended the subject programs with

versions for which we seeded random combinations of *multiple* faults similar to the ones in Siemens. These additional faults are obtained by applying mutation techniques [Andrews et al., 2005, Offutt et al., 1996] to the reference implementation of each program. As each program includes a correct version, we use the output of the correct version as oracle.

For our empirical study, coverage matrices A are obtained by instrumenting each of the programs at statement level with `Zoltar`, a spectrum-based fault localization tool set [Janssen et al., 2009]. Although the coverage of a test input can vary between regression cycles it is outside the scope of our research to compensate its effect, and we will assume that up-to-date coverage information is available when prioritizing. This is a usual assumption in test prioritization literature [Elbaum et al., 2002, Jiang et al., 2009a, Rothermel et al., 2001]. Test cost was measured by the POSIX `time` command.

For Bayesian diagnosis, we assume each component's prior fault probability p_j to be fixed to $p_j = 0.01$ (cf. 10 bugs/KLOC) as the prior distribution has been shown not to be very critical to diagnostic performance [Abreu et al., 2009a].

The false negative rates for FEP and SEQUOIA were obtained through a variation of the PIE (propagation, infection, execution) approach [Voas, 1992] to testability modeling, as done in [Rothermel et al., 2001], as also done in Chapters 2 and 3. First, for each statement for which we wish to calculate the FNR, a set of mutants was created by applying a small set of mutation operators [Offutt et al., 1996] to the arithmetic, logic and indexing operations contained in the statement or function. Second, the program's full test suite was run, recording test coverage and failure information for each of the mutants. Finally, the FNR of each component was obtained by averaging the ratio of the number of tests which covered the fault but did not produce a failure, to the number of tests which covered the fault.

For each program, 100 faulty versions seeded with 1 and 3 random mutation faults were created. For each of the faulty versions, 5 repetitions were made, to a total of 500 prioritized test suites per program.

Fault Localization Effectiveness

We compare in terms of the area difference between C_d and the minimum achievable C_d (once all the tests have been applied) according to the effectiveness formula in Equation 3.8.

$$S = \sum_{n=1}^{N} \left(\frac{C_d(n-1) - C_d(N)}{M - M_f} \cdot C_i \right)$$

Although the name "effectiveness" is counter-intuitive (effectiveness suggests that the more the better, but for S lower is better) this score is inspired by similar metrics used in diagnostic test prioritization literature [Jiang et al., 2009b, Yu et al., 2008] so we decided to adopt the same name. We exclude from M white space and static code to remove bias from the comparisons.

Statistical Diagnosis (Ochiai SC)

Figure 4.1 contains the plots and numerical results in terms of S score and relative percentage with respect to the random order. The best techniques with a 95% statistical confidence in the Bonferroni mean separation test are highlighted in boldface.

The SEQUOIA column was obtained by using the test order produced by SEQUOIA, discarding the internal Bayesian diagnosis and performing a new diagnosis using the similarity coefficient. It can be seen how SEQUOIA's effectiveness appears to be extremely poor, in contrast with its performance when using Bayesian diagnosis as shown in the following section. This is caused by the fact that similarity coefficients do not use all the information provided by IG efficiently.

It can also be seen how RAPTOR consistently provides an improvement in the efficiency for all programs with only one fault, and is the best in all but one case (`tcas`). On average, RAPTOR provides a 54% reduction in terms of S with respect to RND's baseline performance in the single-fault case, and a 45% in the multiple-fault case. When compared to the next best technique, ART, RAPTOR provides a 38% reduction in the single fault case on average, and a 15% in the multiple fault case. For multiple faults, RAPTOR's superiority is still clear, although there are some cases where the performance of RAPTOR is severely affected because the distribution of the faults is very biased, as will be discussed in Section 4.3.

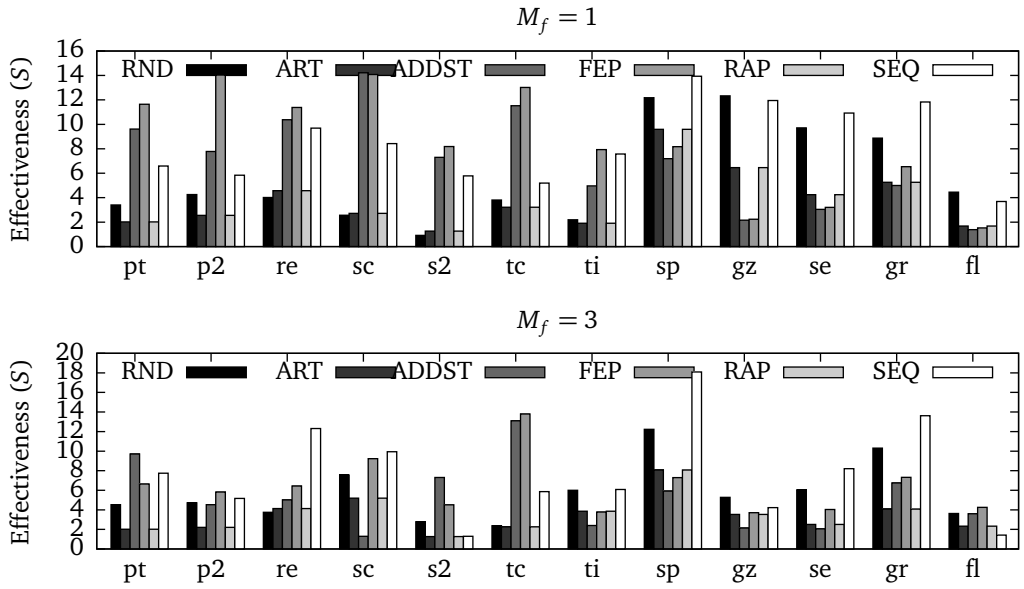
For the Siemens programs, ADDST and FEP are extremely poor even when compared to RND. However, for the SIR programs the trend reverses and produce much better results. The root causes for the low effectiveness of RAPTOR for `tcas` and the striking difference between the effectiveness of ADDST for Siemens and SIR programs will be analyzed in Section 4.3.

Bayesian Diagnosis

We now evaluate the effectiveness of RAPTOR and other techniques using Bayesian fault diagnosis (explained in Section 2.1). Figure 4.2 presents and plots the numerical results, in terms of the S score, and the reduction percentage with respect to RND. Again, the scores in boldface correspond to the top techniques with 95% confidence following the Bonferroni mean separation test. The results are identical to those in Chapter 3 with the only difference of the addition of the RAPTOR column and different statistical test results.

Combined with Bayesian diagnosis, SEQUOIA is no longer among the worst performing but among the best, showing the perfect match between the heuristic and the diagnosis algorithm. However, it can be seen how RAPTOR's effectiveness is on par and even surpasses SEQUOIA, the theoretical optimum, both in the single-fault and multiple-fault case. This is due to the fact that the IG heuristic used in SEQUOIA is largely affected by errors in the FNR estimations, whereas RAPTOR is not.

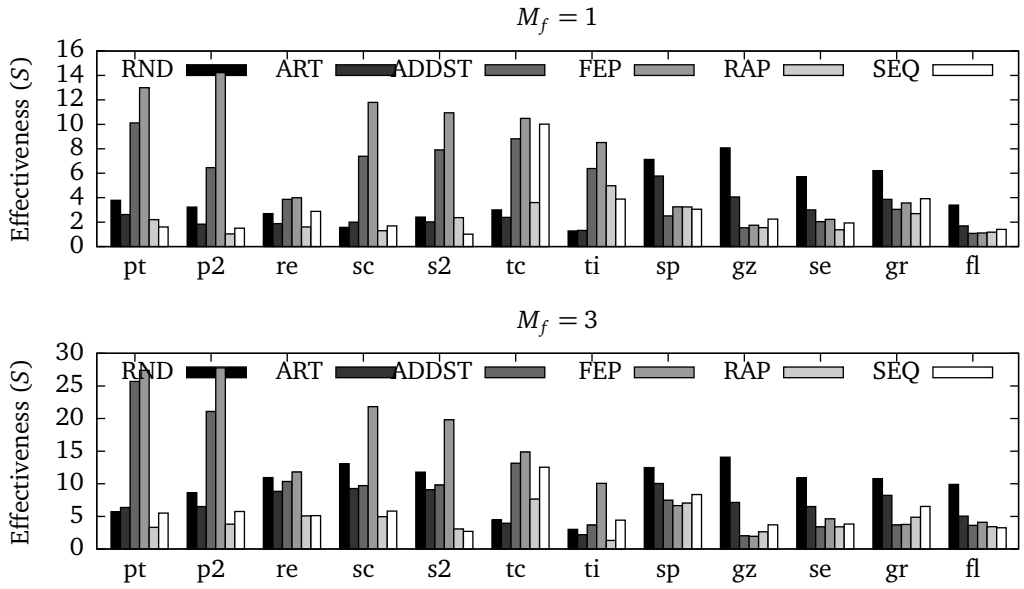
On average, RAPTOR provides a 40% reduction of S in the single fault case with



	$M_f = 1$											
	RND	ART		ADDST		FEP		RAPTOR		SEQUOIA		
pt	3.86	2.46	-36%	10.06	161%	12.09	214%	**1.56**	**-60%**	7.05	83%	
p2	4.90	3.21	-35%	8.43	72%	14.69	200%	**2.25**	**-54%**	6.48	32%	
re	4.64	5.19	12%	10.99	137%	12.01	159%	**3.44**	**-26%**	10.31	122%	
sc	6.19	6.34	2%	17.85	188%	17.71	186%	**3.77**	**-39%**	12.06	95%	
s2	1.77	2.11	19%	8.15	361%	9.03	411%	**1.30**	**-26%**	6.63	275%	
tc	3.81	**3.21**	**-16%**	11.53	203%	13.02	242%	3.47	-9%	5.20	37%	
ti	2.20	1.91	-13%	4.96	126%	7.93	261%	**1.68**	**-24%**	7.57	245%	
sp	13.20	10.61	-20%	8.21	-38%	9.20	-30%	**7.55**	**-43%**	14.95	13%	
gz	12.98	7.11	-45%	2.81	-78%	2.88	-78%	**2.43**	**-81%**	12.60	-3%	
se	10.83	5.35	-51%	4.15	-62%	4.32	-60%	**1.83**	**-83%**	12.03	11%	
gr	12.05	8.44	-30%	8.18	-32%	9.71	-19%	**6.43**	**-47%**	15.02	25%	
fl	4.47	1.71	-62%	1.40	-69%	1.55	-65%	**0.94**	**-79%**	3.71	-17%	

	$M_f = 3$											
	RND	ART		ADDST		FEP		RAPTOR		SEQUOIA		
pt	4.52	**2.01**	**-55%**	9.70	115%	6.64	47%	3.33	-26%	7.74	71%	
p2	4.96	2.45	-51%	4.77	-4%	6.07	22%	**1.42**	**-71%**	5.42	9%	
re	4.38	4.77	9%	5.65	29%	7.07	61%	**2.49**	**-43%**	12.94	195%	
sc	7.55	5.20	-31%	**1.29**	**-83%**	9.22	22%	7.37	-2%	9.94	32%	
s2	2.78	**1.27**	**-54%**	7.31	163%	4.51	62%	2.88	3%	**1.29**	**-54%**	
tc	2.37	**2.27**	**-4%**	13.10	453%	13.79	482%	3.68	55%	5.86	147%	
ti	6.73	4.59	-32%	3.13	-54%	4.50	-33%	**1.15**	**-83%**	6.81	1%	
sp	12.91	8.76	-32%	6.60	-49%	7.97	-38%	**5.76**	**-55%**	18.77	45%	
gz	5.28	3.52	-33%	**2.14**	**-59%**	3.71	-30%	**2.35**	**-56%**	4.22	-20%	
se	6.79	3.26	-52%	2.81	-59%	4.78	-30%	**1.77**	**-74%**	8.96	32%	
gr	10.95	4.72	-57%	7.39	-33%	7.96	-27%	**4.26**	**-61%**	14.26	30%	
fl	3.62	2.33	-36%	3.60	-1%	4.26	+18%	2.32	-36%	**1.42**	**-61%**	

Figure 4.1: Efficiency (S) in terms of statistical diagnosis (Ochiai)



	$M_f = 1$											
	RND	ART		ADDST		FEP		RAPTOR		SEQUOIA		
pt	3.79	2.62	-31%	10.12	167%	13.00	243%	2.20	-42%	**1.60**	**-58%**	
p2	3.23	1.83	-43%	6.46	100%	14.23	340%	**1.03**	**-68%**	1.51	-53%	
re	2.70	1.88	-30%	3.87	43%	3.99	48%	**1.61**	**-40%**	2.88	7%	
sc	1.57	2.00	27%	7.39	369%	11.79	650%	**1.30**	**-17%**	1.69	7%	
s2	2.41	2.02	-16%	7.91	229%	10.95	355%	2.37	-2%	**1.02**	**-58%**	
tc	3.00	**2.39**	**-20%**	8.82	194%	10.49	249%	3.60	20%	10.02	234%	
ti	**1.28**	**1.32**	**3%**	6.38	401%	8.52	568%	4.97	290%	3.88	205%	
sp	7.13	5.77	-19%	**2.51**	**-65%**	3.25	-54%	3.24	-55%	3.06	-57%	
gz	8.08	4.06	-50%	**1.54**	**-81%**	1.75	-78%	**1.55**	**-81%**	2.24	-72%	
se	5.72	3.00	-48%	2.04	-64%	2.23	-61%	**1.38**	**-76%**	1.93	-66%	
gr	6.21	3.86	-38%	3.04	-51%	3.57	-43%	**2.70**	**-57%**	3.91	-37%	
fl	3.40	1.69	-50%	**1.09**	**-68%**	**1.12**	**-67%**	1.18	-65%	1.41	-59%	

	$M_f = 3$											
	RND	ART		ADDST		FEP		RAPTOR		SEQUOIA		
pt	5.71	6.37	11%	25.69	350%	27.38	379%	**3.31**	**-42%**	5.49	-4%	
p2	8.56	6.51	-24%	21.09	146%	27.77	224%	**3.81**	**-55%**	5.74	-33%	
re	10.95	8.84	-19%	10.36	-5%	11.79	8%	**5.08**	**-54%**	**5.12**	**-53%**	
sc	13.07	9.25	-29%	9.72	-26%	21.83	67%	**4.95**	**-62%**	5.81	-56%	
s2	11.78	9.07	-23%	9.79	-17%	19.81	68%	3.06	-74%	**2.70**	**-77%**	
tc	4.44	**3.94**	**-11%**	13.16	196%	14.88	235%	7.66	72%	12.53	182%	
ti	3.00	2.20	-27%	3.65	22%	10.06	235%	**1.31**	**-56%**	4.43	48%	
sp	12.48	10.04	-20%	7.48	-40%	**6.64**	**-47%**	7.03	-44%	8.33	-33%	
gz	14.07	7.14	-49%	**2.03**	**-86%**	**1.95**	**-86%**	**2.64**	**-81%**	3.69	-74%	
se	10.88	6.49	-40%	**3.40**	**-69%**	4.62	-58%	**3.39**	**-69%**	3.82	-65%	
gr	10.78	8.22	-24%	**3.68**	**-66%**	3.77	-65%	4.87	-55%	6.52	-40%	
fl	9.90	5.04	-49%	3.63	-63%	4.09	-59%	3.41	-66%	**3.26**	**-67%**	

Figure 4.2: Effectiveness (S) in terms of Bayesian diagnosis

respect to the baseline set by RND. In the multiple fault case, the reduction with respect to RND is 55%. When compared with the next best technique, ART, RAPTOR achieves a reduction of 13% for single faults, and 39% for multiple faults.

Identically to what happened in the similarity coefficient case, there is a noticeable difference in the effectiveness of FEP and ADDST, between the Siemens and SIR programs, which will be analyzed in Section 4.3.

Fault Detection Performance

We evaluate the techniques in terms of their fault detection performance, by using the $APFD_c$ metric [Elbaum et al., 2001]. This information is useful to understand the modest trade-off between the test prioritization goals of fault detection and fault localization, should one decide to use diagnostic test prioritization even in the test phase (before the first failure). Consistently with previous literature [Jiang et al., 2009a, Rothermel et al., 2001], we will use only single-fault data.

Table 4.5 shows the results in terms of the improvement percentage of the $APFD_c$ metric with respect to RND. The values in boldface represent the best technique, with 95% confidence using the Bonferroni mean separation test.

Consistently with previous literature [Jiang et al., 2009a, Rothermel et al., 2001], it can be seen how ADDST and FEP are the best performing techniques, especially in the Siemens programs. For the larger SIR programs, however, RAPTOR has the best performance, tied with ADDST.

Both observations are consistent with our previous findings [Gonzalez-Sanchez et al., 2011e] indicating that prioritizing for fault localization (independently of the specific heuristic) tends to lower failure rates in some test suites. This reduced performance is not a very serious problem in practice, given the 2% difference between ADDST and RAPTOR. Only `schedule2` presents a significant reduction of 9% for RAPTOR with respect to ADDST. However, the increased test effort is completely outweighted by the large savings in residual diagnostic effort.

4.3 Discussion

In this section we analyze the results, discuss the threats to their validity, and introduce some criteria aimed at choosing the right technique under varying circumstances.

Technique Analysis

It can be seen in our results that there are significant differences in the effectiveness of each technique depending on the program. There is also a noticeable difference between the Siemens programs and the SIR programs that influences the behavior of ADDST, transforming it from the worst of all techniques, to a very suitable one.

	$M_f = 1$									
	RND	ART		ADDST		FEP		RAPTOR		SEQUOIA	
pt	0.93	0.96	+3%	**0.98**	**+5%**	0.96	+2%	**0.97**	**+4%**	**0.97**	**+4%**
p2	0.94	0.96	+2%	**0.98**	**+5%**	**0.99**	**+6%**	0.96	+3%	**0.98**	**+5%**
re	0.89	0.89	+0%	0.88	-1%	**0.92**	**+3%**	**0.92**	**+2%**	**0.93**	**+4%**
sc	**0.95**	**0.95**	**-0%**	0.91	-4%	**0.95**	**-0%**	0.93	-2%	**0.94**	**-1%**
s2	**0.92**	**0.93**	**+1%**	**0.93**	**+1%**	0.90	-2%	0.87	-5%	**0.92**	**+0%**
tc	0.81	**0.84**	**+3%**	0.80	-2%	0.79	-3%	**0.85**	**+5%**	0.80	-1%
ti	0.97	0.97	-0%	**0.99**	**+2%**	0.98	+0%	0.96	-2%	**0.99**	**+1%**
sp	0.70	0.76	+9%	0.79	+13%	0.77	+11%	**0.82**	**+17%**	**0.79**	**+13%**
gz	0.74	0.86	+16%	**0.95**	**+29%**	**0.95**	**+29%**	**0.96**	**+29%**	**0.95**	**+29%**
se	0.75	0.90	+19%	0.94	+24%	0.94	+25%	**0.97**	**+30%**	0.94	+24%
gr	0.75	0.81	+8%	**0.83**	**+11%**	0.81	+8%	**0.85**	**+14%**	0.82	+9%
fl	0.94	0.98	+4%	0.98	+4%	0.98	+4%	**0.99**	**+5%**	**0.99**	**+5%**

Table 4.5: Fault detection performance ($APFD_c$)

To explain the difference in the effectiveness of ADDST and FEP between the Siemens and SIR programs, it is necessary to explain the relationship between coverage, and information gain. For simplicity of exposition, let us assume that there is only one fault in the system, and that the fault will produce a failure whenever covered (FNR=0%). Let us also assume that the coverage of a test case is uniformly distributed throughout the program. In this situation, the fault will always be at the top of the suspect ranking, possibly tied to other candidates. In this simplified case, the information gain that a test provides is determined by the reduction of the size of the top-ranked suspect set. Assuming there are $|D|$ top-ranked suspects, a test of coverage density ρ will reduce the top-ranked set to $|D| \cdot \rho$ components if it fails, and $|D| \cdot (1-\rho)$ if it passes.

Under these conditions, we show in Chapter 5 that the information gain heuristic (Equation 3.6) can be modeled by

$$IG(D,\rho) = -\rho \log_2 \rho - (1-\rho)\log_2(1-\rho) \tag{4.2}$$

which is optimal for $\rho = 0.5$ (50% coverage probability, which also corresponds to 50% failure probability in this case).

In real test matrices this density, ρ, is not uniform, but varies for each test. The plot in Figure 4.3 depicts the coverage density histogram of a program from the Siemens set, and a program from the SIR tools. The IG curve is added as reference. Techniques like ADDST and FEP, designed to *maximize* coverage, will tend to select test cases with a very high ρ (so high failure probability) if they are available, which will fall *past* the IG maximum. This is the case for the Siemens programs. This affects their effectiveness greatly, as can be seen in our results. However, when there are no tests past the IG maximum (very sparse matrices), as is the case for

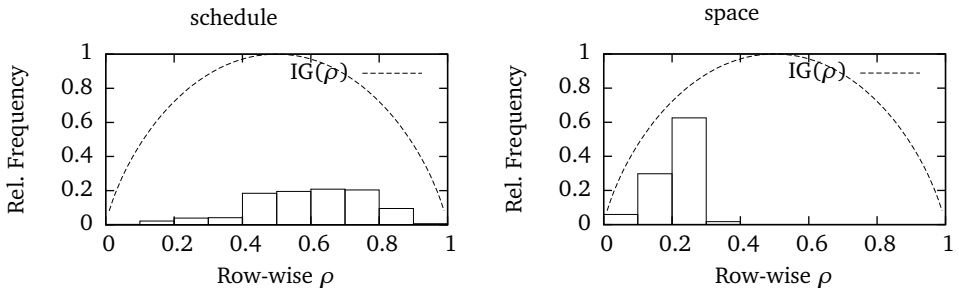


Figure 4.3: Coverage density distribution

`space` and the other SIR programs, ADDST and FEP are indirectly choosing tests that are close to the optimum by maximizing coverage.

The performance of ART and especially RND depends on the random distribution of the IG values of tests in the test suite across the curve in Figure 4.3. This coverage distribution also explains the difference in the $APFD_c$ score between Siemens and the SIR programs. For Siemens, RAPTOR and SEQUOIA *lower* the coverage, whereas for the SIR programs, they both maximize it, as ADDST does, hence the similar scores.

The reason for the bad results of RAPTOR for the `tcas` program is due to the program's nature and some subtle assumptions underlying RAPTOR. First, `tcas` has 174 lines of code but only 9 ambiguity groups, which impacts severely the performance of any similarity coefficient and makes any improvement in effectiveness very small. Second, our sample of seeded faults in `tcas` and `tot_info` are not uniformly distributed in the code, making RAPTOR less optimal. SEQUOIA, being a dynamic technique, can adapt to this non-uniformity, but the errors in the FNR estimation (which affect the IG heuristic) neutralize the gain.

Choice of Technique

We now analyze the factors that influence the decision to test for failures or faults from the first test, and to use either RAPTOR, SEQUOIA, or an alternative.

Wait for the first failure? We have seen in our evaluation that the fault detection capability, i.e., the test effort it takes to detect the first failure, can be slightly increased when using diagnostic test prioritization. The first question is whether to perform diagnostic test prioritization from test 0, or only after the first failure, since RAPTOR and SEQUOIA will return a moderately decreased $APFD_c$. If the introduction of a fault is unlikely, the best solution is to initially use ADDST or FEP, and switch to diagnostic test prioritization if a failure occurs. However, if the probability of introducing a fault during development is very high, or if tests have a very low FNR, it is better to start directly with diagnostic test prioritization since the tests executed until the first failure is found provide the most valuable diagnostic information (if chosen properly). The reduction of $APFD_c$ (increased test effort)

is greatly outweighted by the much more substantial gains in terms of diagnostic performance per unit of test effort. Therefore, even if it takes slightly longer to *detect* a fault, the reduction in the testing effort required to precisely *locate* the fault greatly compensates for this. Furthermore if coverage density ρ is very low, diagnostic prioritization will provide a similar $APFD_c$ as FEP or ADDST.

RAPTOR or SEQUOIA? If using low-cost similarity coefficients, RAPTOR is the clear choice. However, since SEQUOIA can potentially provide better results than RAPTOR combined with Bayesian diagnosis, it is necessary to consider the advantages and disadvantages of each technique.

1. **Computational cost:** Even though RAPTOR and SEQUOIA have a similar order of complexity, RAPTOR is based on fast bit-wise operators, whereas SEQUOIA is based on expensive floating point operations, and requires expensive prior parameter determination. Furthermore, it must be taken into account that SEQUOIA must perform on-line prioritization, as the tests are executed, potentially introducing a significant overhead.

2. **FNR estimation quality:** the IG heuristic used in SEQUOIA can be severely affected by poor FNR estimations. If the variance of the samples used to estimate each h_j is high (as is the case for `tcas`), one should not consider SEQUOIA.

3. **Ambiguity and non-uniformity:** a test matrix with very few and very large ambiguity groups, or a system with extremely biased fault distribution can cause problems to a static algorithm like RAPTOR. If the FNR estimation quality is good enough, one could opt for SEQUOIA. Otherwise, the best would be to opt for a random approach such as ART or RND.

Threats to Validity

The external validity of the results obtained with the Siemens programs can be questioned given their small sizes. In order to strengthen the validity of our results we provided validation experiments with larger programs such as the SIR tools, showing that indeed, for larger matrices where such as the ones for the SIR tools, the behavior of techniques is different.

The fact that we seeded artificial faults into our programs is a second threat to the external validity of our results. From the point of view of the representativeness of mutation faults, they have become usual in test prioritization and fault diagnosis literature [Rothermel et al., 2001, Wang et al., 2009]. Furthermore, previous research [Andrews et al., 2005] claims that mutation faults have similar characteristics to real faults. From the fault density point of view, we restricted the experiment for the Siemens programs to 3 simultaneous faults to keep fault density to a realistic level.

The construct validity of the S metric to measure the gains in effectiveness is another potential source of concern. We argue that this metric better models the reduction in test and residual diagnostic effort than the effectiveness metric used in [Jiang et al., 2009b] since we do not sample an arbitrary point during testing but aggregate the evolution for all tests. Furthermore, since the residual diagnostic effort depends on the diagnostic technique used, the fact that we used two different techniques strengthens the construct validity of our results.

4.4 Related Work

The influence of test-suite extension, reduction, modification, and prioritization on fault detection and diagnosis has received considerable attention. Hao *et al.* [Hao et al., 2005] study the effect of reduction of redundant tests on the Tarantula SC [Jones et al., 2002], concluding that reduction may actually have a beneficial effect on diagnosis. However, their conclusions partially contradict with those of Yu *et al.* [Yu et al., 2008] for a larger sample of programs, who conclude that test-suite reduction, especially coverage-based reduction, has a negative influence on the quality of the diagnosis. In particular, Jiang *et al.* [Jiang et al., 2009b] show how some prioritization techniques are worse than random sequencing and that those which are better, do not provide a significant improvement. All the works cited above study the effect of prioritization and reduction techniques that were designed for fault detection, on fault localization. RAPTOR is, on the other hand, designed with fault localization effectiveness in mind, to optimize the improvement of diagnostic accuracy per unit of test effort.

Baudry *et al.* [Baudry et al., 2006] propose enhancing test suites by adding new tests that increase the number of *dynamic basic blocks* (DBB) (i.e., sets of components which are always covered together), thus reducing diagnostic uncertainty. Their concept of DBB is equivalent to an ambiguity group, however, in their fitness function they consider only the number of groups, and not their size. Furthermore, the additional tests created by this technique are simply added to the test suite, and thus there's no guaranty they will be ordered appropriately, unless diagnostic test prioritization is used.

Test case prioritization is a mature and active area of research whose most common goal is to increase failure detection rate, which dates back to [Harrold et al., 1993b, Wong et al., 1997]. The failure detection effectiveness of different coverage-based prioritization techniques was studied by Rothermel *et al.* [Rothermel et al., 2001] that has already been discussed elsewhere in the chapter. In the same way we propose some guidelines to choose the appropriate diagnostic prioritization technique, Elbaum *et al.* propose a set of guidelines [Elbaum et al., 2004] for regression testing. Li *et al.* [Li et al., 2007] study different search algorithms for coverage-based prioritization. Jiang *et al.* [Jiang et al., 2009a] propose a hybrid random and coverage-based prioritization technique (ART) which has already

been discussed throughout this chapter. RAPTOR fundamentally differs from the above prioritization techniques in that its main goal is not to optimize the rate of failure detection, but to also minimize debugging effort. Cost-aware test prioritization techniques have been proposed [Elbaum et al., 2001, Smith and Kapfhammer, 2009, Zhang et al., 2009] to address the cost variability problem.

Automated fault-localization techniques aim at minimizing residual diagnostic effort when failures occur during the testing phase. Statistical approaches include [Abreu et al., 2009b, Jones et al., 2002, Liblit, 2008, Liu et al., 2005, Renieris and Reiss, 2003, Wong et al., 2008]. A recent, probabilistic approach of acceptable complexity is [Abreu et al., 2009a]. All the above approaches do not address test sequencing to optimize diagnostic accuracy, and are essentially similar to RND. Other approaches to fault localization include [Baah et al., 2008, Santelices et al., 2009, Wang et al., 2009].

Sequential diagnosis aims at finding the test sequence that optimizes diagnostic performance based on the current test outcomes, applied to hardware systems [Raghavan et al., 1999]. Typically, only one fault is assumed present in the system. The intelligent probing mechanism by Brodie *et al.* [Brodie et al., 2002] uses a similar measurement of ambiguity for test matrix reduction (minimal set of probes), applied to computer networks. If the system can have multiple faults, sequential diagnosis complexity becomes exponential. In [Shakeri et al., 2000] an approximation is described for systems with 0% FNR, an assumption which does not hold for most software systems and test suites. In [Gonzalez-Sanchez et al., 2010a], SEQUOIA, a near-optimal sequencing approach for multiple faults and tests with false negatives, based on information gain, was introduced. However, SEQUOIA requires the precise determination of a number of input parameters that may be very costly, unlike the robust alternative offered by RAPTOR.

4.5 SUMMARY

In this chapter we presented a diagnostic test prioritization algorithm, RAPTOR, that selects test cases by their ability to produce a refined diagnosis (fault *localization*) rather than to produce a failure as early as possible (fault *detection*). Our results show that RAPTOR's test prioritization can significantly improve the reduction of residual diagnostic effort per executed test case, when compared to random (RND), adaptive-random (ART), additional statement coverage (ADDST), and IG-based (SEQUOIA) prioritization. When using similarity coefficients, RAPTOR provides a 38% and 15% average reduction in S with respect to the next-best (ART) for single and multiple faults, respectively. In the case of Bayesian diagnosis, RAPTOR achieves a 13% and 39% average reduction of S with respect to the next-best (ART) for single and multiple faults, respectively. Given its negligible impact on fault detection capability (as measured by $APFD_c$), RAPTOR can be used as the sole test prioritization strategy during regression testing in many cases, especially when

the probability of introducing defects is high.

We have observed that the characteristics of the program and test suite (coverage density, ambiguity, FNR) can have a great impact in the effectiveness of all prioritization algorithms. The following chapter will be devoted to modeling the influence of these characteristics. An understanding of these factors will allow us to define new criteria to be used to create new diagnostic test suites or improve existing ones.

5: Performance Modeling of Prioritization Algorithms

Motivated by the fact that the diagnostic effectiveness achieved by the diagnostic test prioritization algorithms SEQUOIA *and* RAPTOR *is not always much better than a randomly ordered test suite, we investigate what structural characteristics of the coverage matrix induced by a test suite are desirable to obtain a good effectiveness.*

In this chapter we present an analytical model and a simulation model to predict the diagnostic effectiveness of test suites when prioritized with the information gain algorithm. We show that, besides the size of the system itself, an optimal coverage density and uniform coverage distribution are the most important test suite parameters. Our models allow us to decide whether using IG with our current test suite will provide a good diagnostic effectiveness, and enable us to define criteria for the generation or improvement of test suites.

5.1 Preliminaries **75**
5.2 Performance Prediction **75**
5.3 Effectiveness Prediction Using Ambiguity **82**
5.4 Experiments **84**
5.5 Related Work **87**
5.6 Summary **90**

As seen throughout the previous chapters, not every test suite has the same diagnostic power. This motivates the study of which characteristics of test suites are beneficial for a good diagnosis.

The Test-for-Diagnosis (TfD) test criterion was defined as a way to express the quality of a test suite for diagnosis [Baudry et al., 2006]. When a test suite cannot produce sufficiently accurate diagnoses its quality can be enhanced by generating new test inputs [Baudry et al., 2006]. However, is accuracy the only relevant dimension of a good diagnostic test suite? Is a perfect diagnosis acceptable if it required executing an disproportionately large number of test cases?

Diagnostic effectiveness, i.e., the rate at which the diagnosis improves, must also be taken into account. Recent work has partially addressed diagnostic effectiveness

by means of *diagnostic (test case) prioritization* [Gonzalez-Sanchez et al., 2011e]. By selecting test cases from the test suite by their fault diagnostic utility, diagnostic effectiveness can be greatly enhanced. Although the results of diagnostic prioritization are very promising, some cases do not provide a large improvement over random orderings (the default) or any other existing prioritization technique. This is a clear sign that there must be inherent characteristics of a test suite that affect the rate at which the diagnostic quality improves. For example, a regression test suite that tests program features individually will eventually produce a very precise diagnosis (since only the feature being tested can be blamed for it) but does not leave large room for improvement for any diagnostic prioritization algorithm precisely because of the lack of coverage overlap between tests.

An understanding of characteristics such as coverage density and overlapping will allow the definition of measurements. These measurements are not only useful for *predicting* the quality of test matrices, but also for deciding what prioritization algorithm to use, and as target criterion for the *generation* of new test suites.

In this chapter we present the following contributions:

1. An analytical model for the prediction of the diagnostic effectiveness of a test suite, based on purely structural characteristics of the test coverage matrix. Our model covers both the single and multiple-fault cases, and also covers the case when tests can produce false negatives.

2. A simulation model for the prediction of diagnostic effectiveness for the cases when the conditions of the validity of the analytical model are not met.

3. An evaluation of our models for synthetic matrices with uniformly distributed coverage, and for coverage matrices corresponding to the test suites of 12 real programs.

Our results show that both models capture the main characteristics that influence the effectiveness of diagnostic prioritization. Our analytic model proves that *coverage density* and *coverage distribution* are the two most important structural parameters to take into account, together with the false negative rate (FNR) of the test cases. Our simulation model produces the best predictions. However, it is not directly based on structural test suite characteristics and relies on expensive simulations, which is not suitable for usage in test generation algorithms that are based on meta-heuristics such as genetic algorithms.

The chapter is organized as follows. Sections 5.2 and 5.3 discuss the characteristics that have an influence on diagnosis effectiveness. Section 5.4 details the experimental setup and discusses the results. Section 5.5 discusses the most relevant related work. Section 5.6 wraps up the chapter and presents our future work directions.

5.1 Preliminaries

Diagnostic performance is accounted in terms of a metric C_d that measures the percentage of excess *effort* incurred in finding all statements at fault [Abreu et al., 2009a]. C_d measures *wasted effort*, independent of the number of faults M_f in the program, to enable an unbiased evaluation of the effect of M_f on C_d. Thus, regardless of M_f, $C_d = 0$ represents an ideal diagnosis technique (all M_f faulty statements on top of the ranking, no effort wasted on testing other statements to find they are not faulty), while $C_d = M - M_f$ represents the worst case (testing all $M - M_f$ healthy statements until arriving at the M_f faulty ones).

For example, consider the ranking $\langle c_{12}, c_5, c_6, \ldots \rangle$, while c_6 is the actual fault. In this case, the diagnosis leads the developer to inspect c_{12} and c_5 first. As both statements are healthy, C_d is increased with 2. The next statement to be inspected is c_6. As it is faulty, no more effort is wasted, and consequently, $C_d = 2$.

Diagnostic effectiveness S, measures the evolution of diagnostic effort C_d with respect to the amount of tests that have been used so far.

S is expressed in terms of the area under the C_d curve, and above the final C_d (i.e., C_d once the N tests in the test suite have been executed), according to

$$S = \sum_{n=1}^{N} \left(\frac{C_d(n-1) - C_d(N)}{M - M_f} \cdot C_i \right) \tag{5.1}$$

where C_i represents test cost. This metric is more desirable than, e.g., the value of C_d after an arbitrary number of tests or the number of tests required to reach an arbitrary C_d. The reason for this is that it avoids the definition of arbitrary performance thresholds. Since we assume all faults are perfectly diagnosable, $C_d(N) = 0$, and that tests will have unit cost, $C_i = 1$, Equation 5.1 can be simplified to

$$S = \frac{1}{M - M_f} \sum_{n=1}^{N} C_d(n-1) \tag{5.2}$$

5.2 Performance Prediction

In this section, we study the characteristics of the coverage matrices of test suites, A, that determine how fast the diagnosis will converge to its ultimate value, i.e., their diagnostic effectiveness, S. To this end, we will model the diagnostic effectiveness of information gain algorithms. Our objective is to obtain a model that depends exclusively on structural parameters of the test suite, A. We first assume that only one fault is present in the system. In the following section we will discuss what are the implications when multiple faults can be simultaneously present in the system.

Best-case values

The first step to predict the behavior of both the diagnostic performance, C_d, and the diagnostic effectiveness, S, is to model the most optimal behavior for a diagnosis.

Under optimal circumstances and following our assumptions of no false negatives, single fault, and perfect final diagnosis, each test case will provide $IG = 1$, i.e., the size of D will be halved. This is the optimal IG given that every test produces a binary outcome. Thus, it will take $\log_2(M)$ tests to find the diagnosis.

Since at $n = 0$ (without any information) one has to traverse on average $\frac{M-1}{2}$ candidates, the ideal shape of the evolution of diagnosis effort will be a negative-exponential decay

$$C_d(n) = \frac{M-1}{2} \cdot 2^{-n} \tag{5.3}$$

Therefore, the ideal diagnostic effectiveness for a system of M components (for large M) corresponds approximately to

$$S \approx \frac{1}{M-1} \int_0^N C_d(n)\, dn \approx \frac{1}{2 \cdot \ln(2)}$$

Worst-case values

If the coverage matrix is so sparse or so dense that it contains either a diagonal of 1's or a diagonal of 0's, there is no possibility for a negative exponential decay of C_d since the only possibility is to discard the components one by one. This causes a linear decay of C_d by

$$C_d(n) = \frac{M-1}{2} - \frac{1}{2} \cdot n$$

(each test eliminating only 1 component) and leads to an effectiveness value of

$$S = \frac{1}{M-1} \int_0^{M-1} C_d(n)\, dn = \frac{(M-1)}{4}$$

Uniform Coverage Matrices

Matrices of uniformly distributed coverage are used very often as synthetic benchmark for diagnosis algorithms to demonstrate near-best-case performance.

These matrices are obtained by uniformly distributing coverage across the matrix, with a density of ρ. In other words, each test covers a random sample of ρM components. Conversely, each component is covered randomly ρN times.

To understand the diagnostic effectiveness of uniform matrices, we must investigate the relationship between coverage density, ρ, and information gain. The value of IG depends on how the D set is split by every test. The set D is split into D_f of size $\rho \cdot |D|$ if the test fails, and into D_p of size $(1-\rho) \cdot |D|$ if it passes.

As explained in Section 2.1, the probability of a test failing, corresponds to

$$\Pr(o_i = 1) = \sum \Pr(d_k) \cdot \Pr(o_i | d_k)$$

There are a number of circumstances we can use to simplify this expression. First, given the single fault assumption, $\Pr(d_k)$ will be equal to $\frac{1}{|D|}$ for every candidate. Second, given the no false negative assumption, $\Pr(o_i|d_k) = a_{ik}$. And third, given the coverage uniformity assumption, there will be $\rho \cdot |D|$ covered. Following these assumptions, we can rewrite failure probability as

$$\Pr(o_i = 1) = \frac{\rho \cdot |D|}{|D|} = \rho$$

The information gain heuristic, therefore, can be rewritten as

$$IG(\rho, D) = \log_2(|D|) - \rho \cdot \log_2(\rho \cdot |D|) - (1-\rho) \cdot \log_2((1-\rho) \cdot |D|)$$

All $|D|$ terms can be removed from the formula, simplifying the equation to

$$IG(\rho) = -\rho \cdot \log_2(\rho) - (1-\rho) \cdot \log_2((1-\rho)) \tag{5.4}$$

The optimal $IG = 1$ value corresponds to $\rho = 0.5$. In the limit,

$$\lim_{\rho \to 0^+} IG(\rho) = \lim_{\rho \to 1^-} IG(\rho) = 0$$

which is also reasonable, since both a matrix filled with 1's and a matrix filled with 0's are not able to diagnose anything.

The evolution of C_d for uniform matrices can be modeled then by adjusting the exponential decay in Equation 5.3 by the $IG(\rho)$ term:

$$C_d(n, \rho) = \frac{M-1}{2} \cdot 2^{-n \cdot IG(\rho)} \tag{5.5}$$

From this formula it can be seen how matrices with $\rho = 0.5$ provide ideal effectiveness. Increasing or decreasing ρ will cause a reduced effectiveness. The density of a matrix cannot be arbitrarily low or high without compromising its diagnosability (the ability to uniquely distinguish all faults). There is a practical limit of $\rho = \frac{1}{M}$ that corresponds to the density of an identity matrix of size $M \times M$.

This can be seen graphically in Figure 5.1 for $M = 500, N = 1000$ uniform matrices. The points plot the real evolution of C_d, based on Monte Carlo simulations of IG-based diagnoses, whereas the lines correspond to the model's prediction.

The diagnostic effectiveness for a system of M components with a uniform test matrix (for large N) corresponds approximately to

$$S = \frac{1}{M-1} \int_0^N C_d(n)\, dn \approx \frac{1}{2 \cdot \ln(2) \cdot IG(\rho)} \tag{5.6}$$

Only if M is low this model will not hold since there is a practical minimum of $\rho = \frac{1}{M}$ which is ignored in the integration approximation of the discrete summation in Equation 5.6.

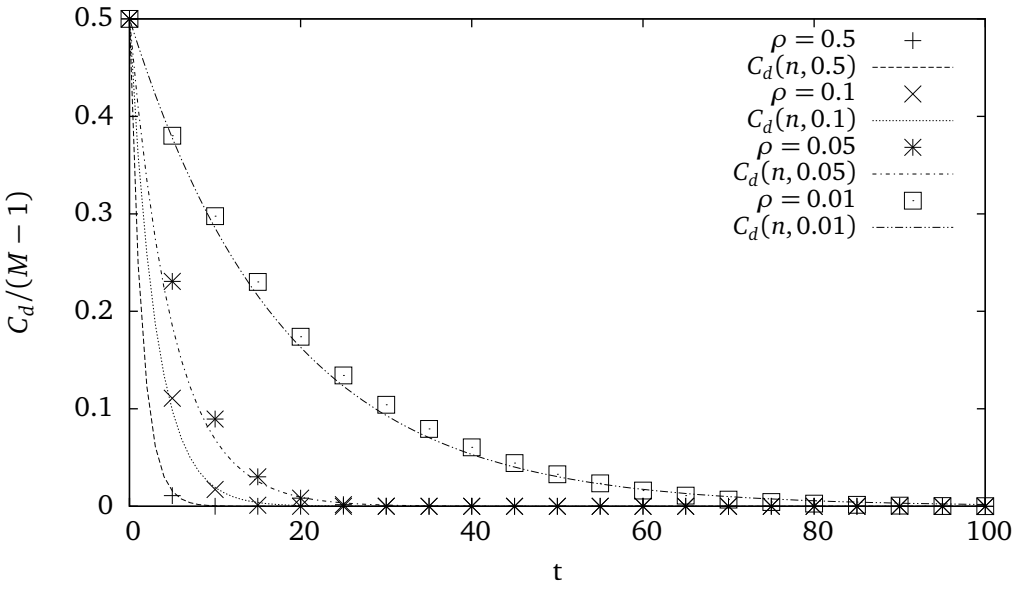


Figure 5.1: Real C_d and prediction for $M = 500 \times N = 1000$ uniform matrices

Influence of False Negatives and Uniform Matrices

In this section we derive the analytical model for uniform test matrices where tests can produce false negatives. We will assume that all components have the same false negative rate, h.

A test will cover $\rho|D|$ components, therefore the probability of having a failure is

$$\Pr(o_i = 1) = \frac{\rho|D|}{|D|} \cdot (1-h) = \rho \cdot (1-h)$$

which leads us to think that the IG formula for false negatives contains

$$IG \approx -\rho\,(1-h) \cdot \log_2(\rho) - \big(1-\rho\,(1-h)\big) \cdot \log_2\big(1-\rho\,(1-h)\big)$$

These terms cannot be the only ones since this formula does not produce $IG = 0$ for $\rho = 1$. Let us take a more detailed look at what happens to entropy when executing a test in which false negatives can occur.

As in the previous case, initially the diagnosis has an entropy equal to

$$H(D) = \log_2(|D|)$$

In case of failure, the tested candidates are the only ones that are kept in D after the test (h does not matter) leading to

$$H(D_f) = \log_2(|D| \cdot \rho)$$

In case of pass, the derivation is more complex. The probability of passing corresponds to $\Pr(o_i = 0) = 1 - \rho(1-h)$. Two separate updates take place in a passed test. For the non-covered components, their initial probability ($\frac{1}{|D|}$) is simply divided by $\Pr(o_i = 0)$. Since there are $(1-\rho)\cdot|D|$ of these candidates, their part in the H formula sum equals

$$H_A = -\frac{1-\rho}{1-\rho(1-h)} \cdot \log_2\left(\frac{1}{|D|(1-\rho(1-h))}\right)$$

For the covered components, their initial probability $\frac{1}{|D|}$ is multiplied by h (following Bayes rule) and then normalized by $\Pr(o_i = 0)$. Since there are $\rho \cdot |D|$ of them, their part in the H summatory is given by:

$$H_B = -\rho \cdot h \cdot \frac{1}{(1-\rho(1-h))} \cdot \log_2\left(\frac{h}{|D|(1-\rho(1-h))}\right)$$

By combining $H_A + H_B$, the entropy of the diagnosis after a pass, $H(D_p)$, results in

$$\begin{aligned} H(D_p) = & \frac{1-\rho}{1-\rho(1-h)} \cdot \left(\log_2|D| + \log_2\left(1-\rho(1-h)\right)\right) \\ & + \frac{\rho \cdot h}{1-\rho(1-h)} \cdot \left(\log_2(|D|) + \log_2\left(1-\rho(1-h)\right) - \log_2(h)\right) \end{aligned}$$

All the pieces can be combined into the analytical model of IG resulting in

$$\begin{aligned} IG(\rho,h) = & -\rho\,(1-h)\cdot\log_2(\rho) \\ & -\left(1-\rho\,(1-h)\right)\cdot\log_2\left(1-\rho\,(1-h)\right) \\ & +\rho\cdot h\cdot\log_2(h) \end{aligned} \tag{5.7}$$

which, for $h = 0$, reduces to Equation 5.4.

Figure 5.2 presents a 3D plot with ρ and h in the horizontal axes and $IG(\rho,h)$ in the vertical axis. It can be seen how the maximum possible IG value decreases as h increases. Furthermore, as h increases, the optimum coverage density shifts away from $\rho = 0.5$ towards lower densities.

Figure 5.3 shows how our model accurately predicts the effectiveness of uniform matrices with false negatives, for $M = 500, N = 1000$ uniform matrices. The points plot the real evolution of $C_d(t,\rho,h)$, based on Monte Carlo simulations of IG-based diagnoses, whereas the lines correspond to the model's prediction.

Prediction for Multiple Faults

To address the possibility of a system containing $M_f > 1$ faults, one can construct a new matrix A' with $\binom{M}{M_f}$ columns for all the combinations of M_f faults, such that columns correspond to diagnostic candidates of cardinality M_f.

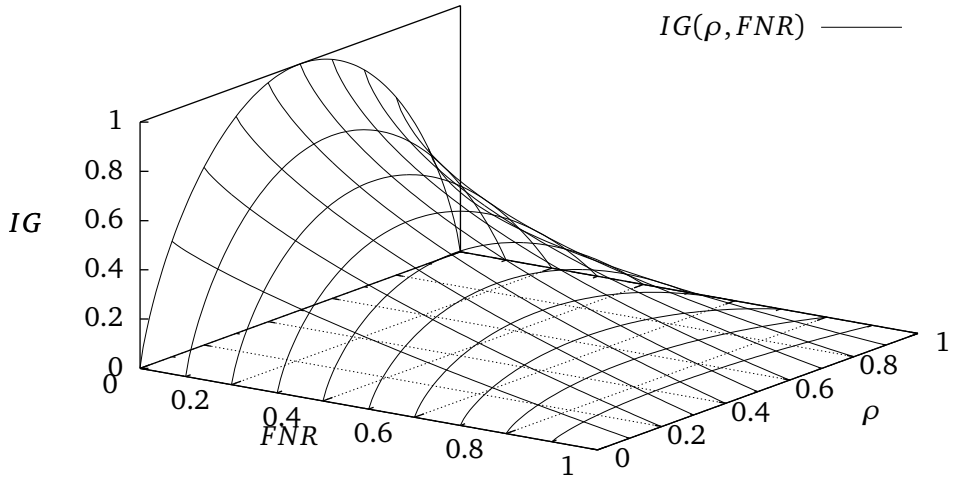


Figure 5.2: Information gain as a function of coverage and FNR

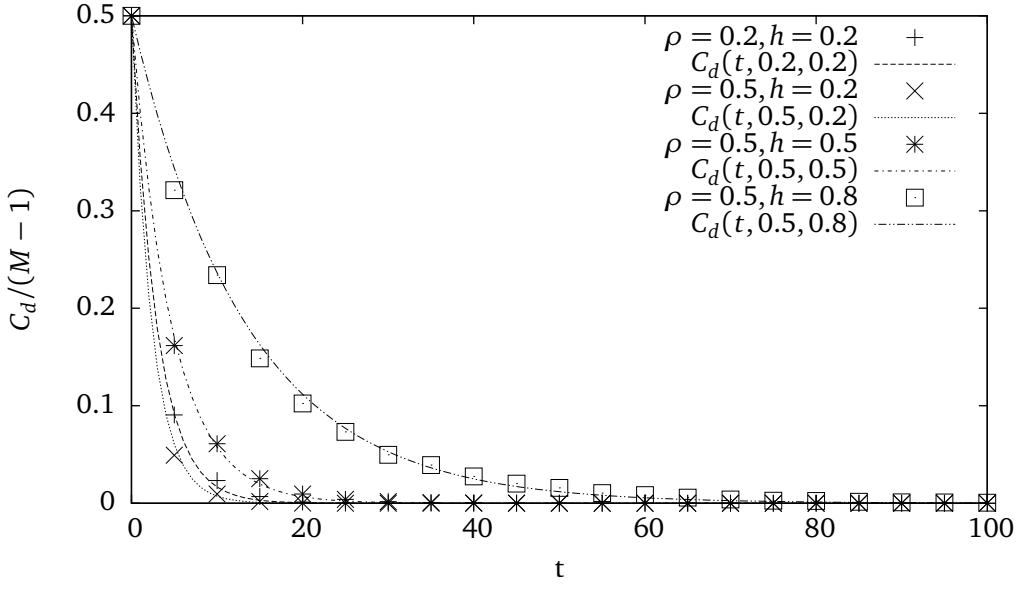


Figure 5.3: Real C_d and prediction for $M = 500 \times N = 1000$ uniform matrices, $h > 0$

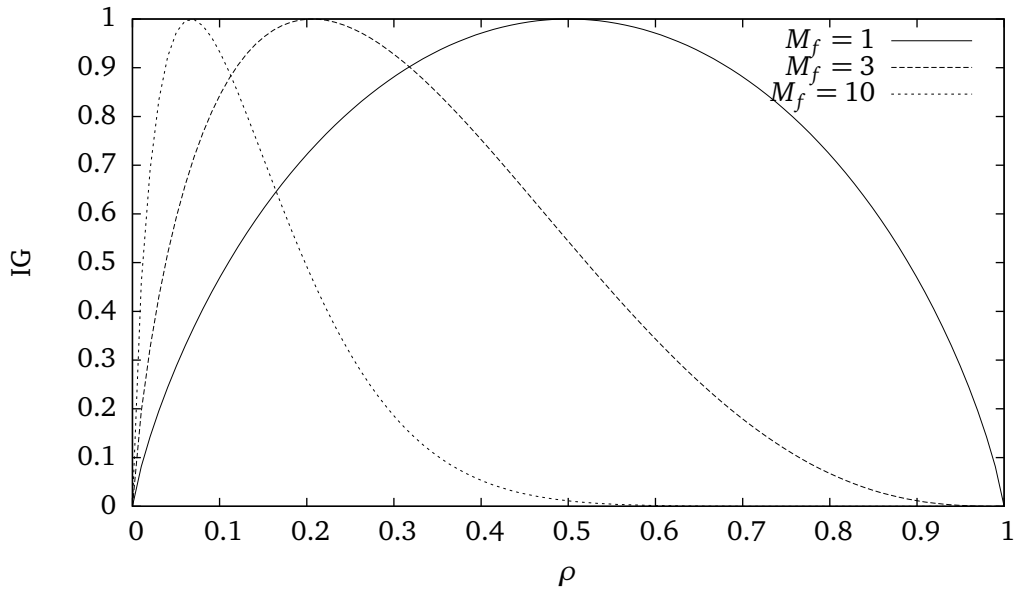


Figure 5.4: Information gain and coverage density

The coverage of A' will be denser than the original A, since, for example, the column for the combination $\{1,2,3\}$ will have 1's whenever either $a_{i1} = 1$, $a_{i2} = 1$ or $a_{i3} = 1$. Hence, the density in A' is given by

$$\rho' = 1 - (1-\rho)^{M_f}$$

which can be substituted in Equation 5.4 to calculate the performance for any given number of faults. Figure 5.4 shows the IG curve for different number of faults.

As can be seen, the optimal ρ is not the same, which has the very important consequence that the coverage density of a test suite must be adapted to the estimated fault density and system size to achieve an efficient diagnosis. The higher the fault density, the lower the coverage density must be.

In the single-fault case, IG(ρ) can be used to calculate C_d, since there is a direct relationship between candidates d_k and components c_j. However, in the multiple fault this relationship is more complicated since the model predicts the evolution of the number of candidates, which overlap many components.

Figure 5.5 shows some empirical results on multiple-fault sequential diagnosis using SEQUOIA for synthetic matrices of $M = 50$, $N = 250$. It can be seen how increasing M_f or ρ has a detrimental effect on effectiveness. The figure includes fitted curves to

$$C_d = \frac{M_f}{M_f + 1} \cdot 2^{\alpha \cdot n}$$

where the $\frac{M_f}{M_f+1}$ term adjusts the effort of finding M_f faults by random inspection, and α is the fitted parameter, which is the real decay parameter of C_d.

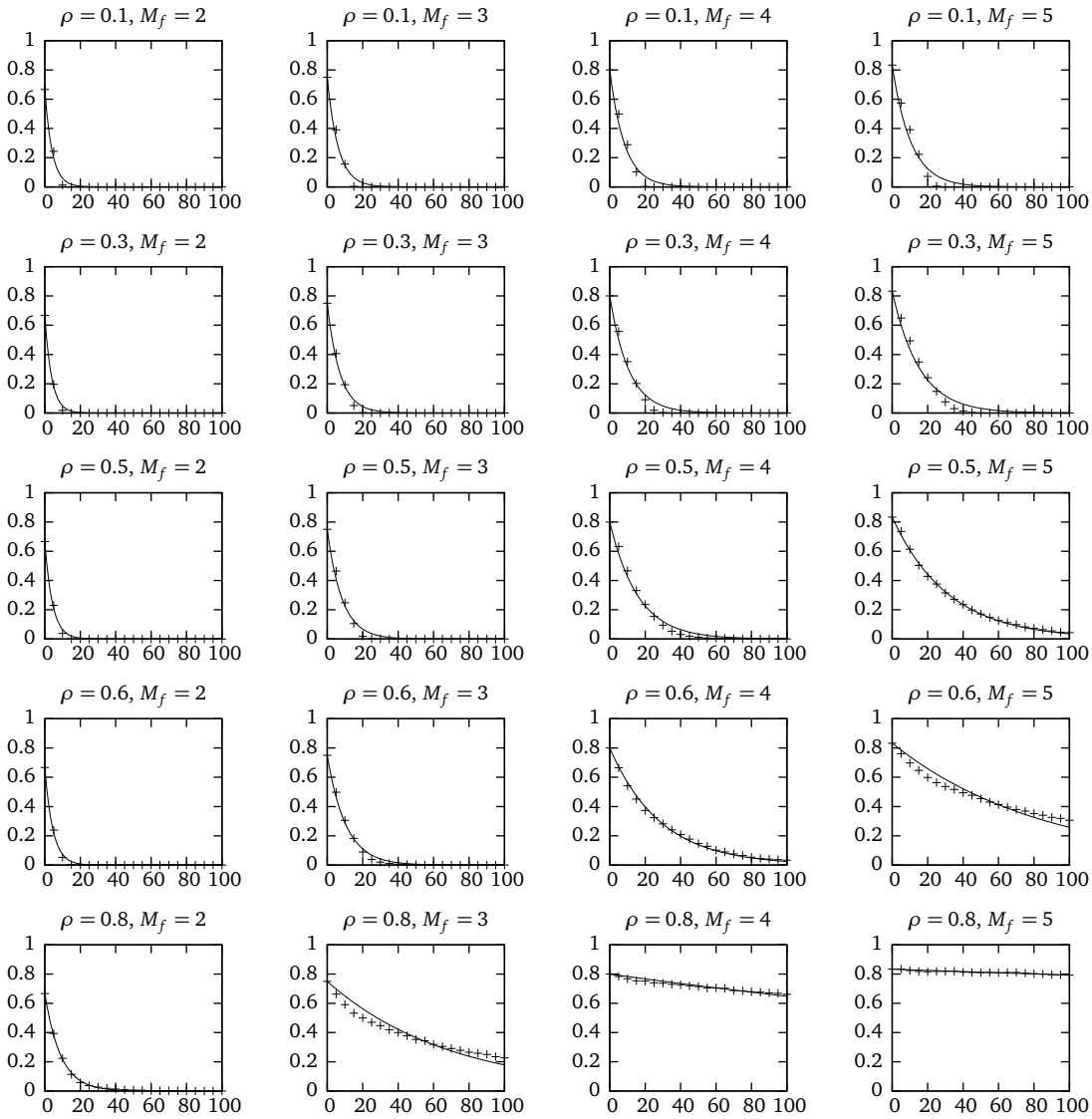


Figure 5.5: C_d evolution trends for multiple faults

5.3 Effectiveness Prediction Using Ambiguity

The model for uniform matrices in Section 5.2 will cause imprecisions when applied to real program coverage matrices, because their coverage distribution is not uniform. In fact, information gain per test choice will be far from constant during the diagnosis. To address non-uniform coverage, we will model the rate of ambiguity reduction of the test matrix, a measurement that is related to the entropy of the test cases. If the ambiguity reduction rate is good, IG will be good. If the ambiguity reduction rate is poor, IG will also have a poor performance.

As already explained in Chapter 4, as tests are applied, a test matrix A decom-

poses the system into a partition $G = g_1, g_2, \ldots, g_L$ of subsets of all components with identical columns in A. As the columns in such group are identical, any individual component belonging to such set can never be uniquely identified as faulty, hence the name *ambiguity group*. For example the matrix

$$A = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

has a higher ambiguity than

$$A' = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix}$$

since the latter can be used to uniquely identify all the faulty components, while the first one cannot distinguish between c_1 and c_2.

A numerical measurement for the ambiguity of a test matrix can be defined based on the partition of ambiguity groups it generates, G, by taking into account the probability of the system containing a fault belonging to each of the ambiguity groups, times the residual diagnostic effort of each group. The probability of group g_i occurring is $\Pr(g_i) = \frac{|g_i|}{M}$, and the residual diagnostic effort in such group corresponds to $C_d = \frac{|g_i|-1}{2}$. The formula for the average diagnostic ambiguity is then

$$\mathrm{G}(A) = \sum_{i=1}^{L} \frac{|g_i|}{M} \cdot \frac{|g_i| - 1}{2} \tag{5.8}$$

and represents the expected (i.e., average) value of the diagnostic effort once the tests in A have been applied.

Algorithm 6 Ambiguity Reduction Estimation

```
A ← ∅
S ← (M−1)/2
for l ← 1, N do
    i = arg min_{t_i} (G(A||t_i))
    A ← A||t_i
    S ← S + G(A)
return S
```

Algorithm 6 computes the evolution of $G(A)$ in a greedy way. At each step, it selects the test that minimizes the ambiguity of the matrix and appends the test to the matrix (operator $||$) . Initially the diagnosis is totally ambiguous ($C_d = \frac{M-1}{2}$). At every iteration, a test t_i is selected, such that, when concatenated to the matrix, it minimizes its ambiguity.

Obtaining the partition G has $O(M)$ time complexity when done incrementally. This has to be repeated N times per choice, for N choices. The complexity of

Algorithm 6 is $O(MN^2)$, similar to IG but with a much lower constant, and it must be done once instead of being repeated a large number of times.

5.4 Experiments

We have already shown the accuracy of our model with uniform matrices. With our experiments we want to evaluate the accuracy of our model when applied to real program matrices.

Experimental Setup

For our study, we use a set of seven test programs known in fault localization literature as the Siemens set [Hutchins et al., 1994], and five programs from the Software Infrastructure Repository (SIR) [Do et al., 2005]. Each program has a set of test inputs that ensures full code coverage. Table 5.1 provides more information about the programs (for more detailed information refer to [Hutchins et al., 1994] and [Do et al., 2005]). Although the Siemens set and SIR programs were not assembled with the purpose of testing fault diagnosis techniques, it is typically used by the research community as the standard set of programs to test their techniques.

Program	LOC	Tests	Description
`print_tokens`	563	4130	Lexical Analyzer
`print_tokens2`	509	4115	Lexical Analyzer
`replace`	563	5542	Pattern Matcher
`schedule`	412	2650	Priority Scheduler
`schedule2`	307	2710	Priority Scheduler
`tcas`	173	1608	Aircraft Control
`tot_info`	406	1052	Information Measure
`sed`	7125	370	Stream Editor
`space`	9126	150	ADL Compiler
`grep`	13287	809	String Matching
`gzip`	7933	210	Data Compression
`flex`	14194	107	Lexer Generator

Table 5.1: Set of programs and versions used in the experiments

The coverage matrix A of each program is obtained by instrumenting each of the programs with `Zoltar` [Janssen et al., 2009] to obtain the statements covered by each test case. Type and variable declarations and other static code, which are not instrumented by `Zoltar`, are excluded from diagnostic rankings and effort calculations. Since in this chapter we assume every component is uniquely diagnosable, we combine the columns of ambiguity groups into a single one. Since we assumed no false negatives can occur, we can also combine identical rows. This

caused some programs to dramatically drop the number of effective number of rows and columns. The final sizes of the test matrices are shown in Table 5.2.

To obtain the fault diagnosis effectiveness S of each test suite we simulated the occurrence of 100 individual faults, and applied Bayesian diagnosis for 10 IG-ordered test sequences, recording the evolution of $C_d(n)$ for each applied test case. These S values are compared and fitted to each of the different predictions.

The results of our empirical validation are shown in Table 5.2. Column "ES" represents the prediction according to the negative exponential model of Equation 5.6. The "AS" column contains the value of S predicted by the ambiguity reduction approach of Algorithm 6. Figure 5.6 presents the scatter plots of each model's prediction. Each of the plots contains the correlation coefficient and relative error with respect to S. A trend line is also added to the plot. Since the model has a large dependence on M, we divide $C_d(n)$ by $M-1$ in order to eliminate program size from the measurement and make the error caused by non-uniformity more visible.

Measurement of effectiveness

The first conclusion to be drawn from our results is that, despite being approximate models, both of them present a good correlation and moderate relative error. Second, it can be seen that our effort to obtain better predictions, taking non-uniformity into account by using the ambiguity reduction method does provide better results. AS presents a higher correlation, although no great improvement is achieved. In fact, the relative error of AS (17%) is larger than ES (12%). It must be noted that AS requires costly computations and that it is not based on structural properties of the test suite but on the simulation of a simplified IG process. Therefore, the negative-exponential model is more interesting since it relates directly the structural properties of the coverage of the test suite to effectiveness. Moreover, since coverage density is extremely cheap to calculate, it can be used on test generation algorithms (e.g., genetic algorithms) as fitness function for the addition of new test cases to the test matrix, which is one of the main motivations for this chapter.

Evolution of Performance

To further understand the sources of imprecision for each of the measurements, the plots for the actual and predicted C_d evolution are shown in Figure 5.8, together with the probability distribution function of the component-wise $IG(\rho)$ for each matrix.

The results show how, in some cases, the analytic model for uniform matrices produces very optimistic results. Especially in the cases of `grep` and `space`. The source for this imprecision is a relatively low coverage density paired to a very non-uniform distribution of coverage per test. Because the coverage distribution is not

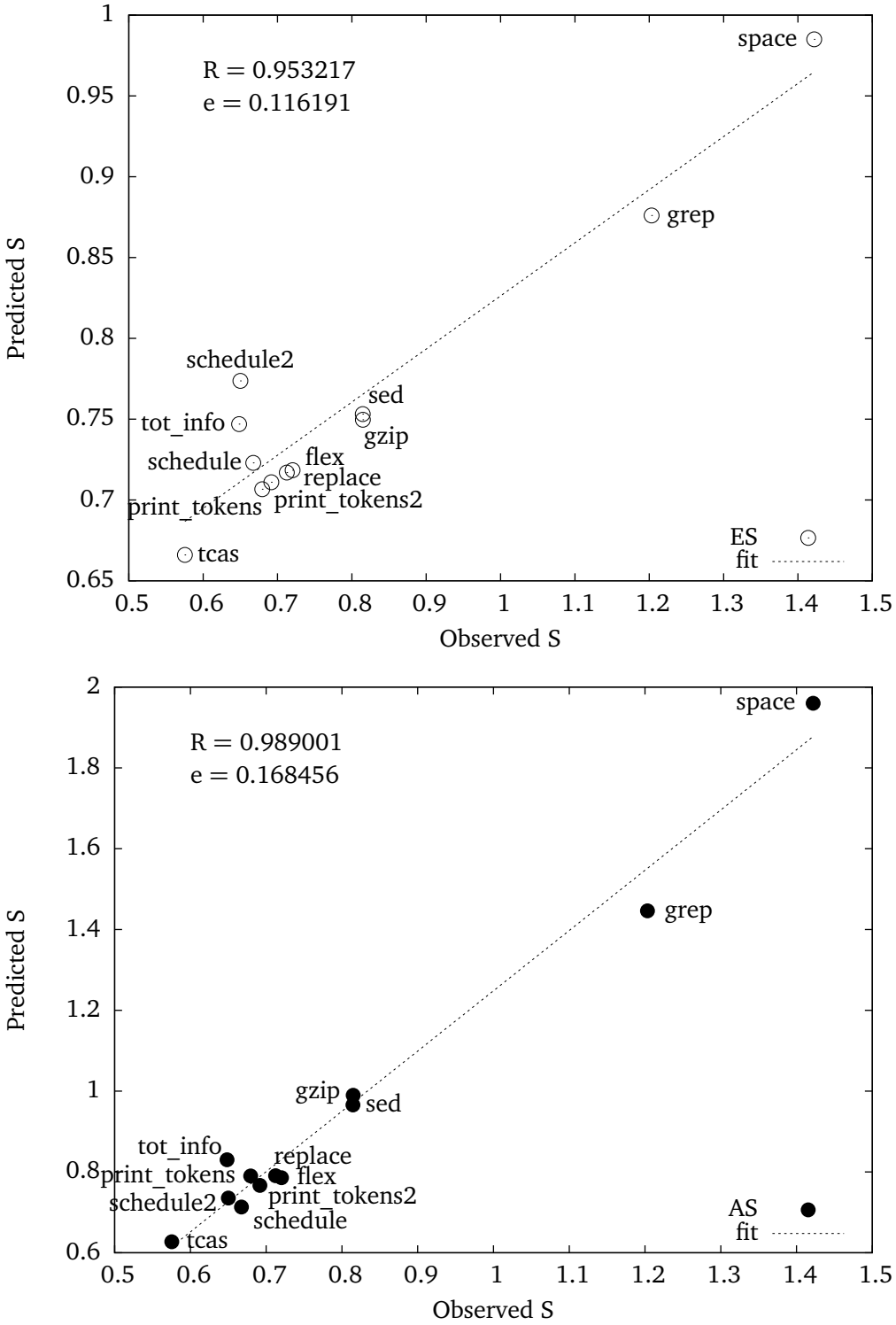


Figure 5.6: Prediction and real value of S for Siemens and SIR programs

Program	M	N	ρ	S	ES	AS
print_tokens	48	1840	0.4869	32.6180	33.9202	37.9167
print_tokens2	63	1721	0.5228	43.5736	44.7908	48.2937
replace	90	2023	0.4583	64.1245	64.5239	71.1500
schedule	39	460	0.5978	26.0262	28.1948	27.8077
schedule2	32	663	0.6811	20.8010	24.7586	23.5313
tcas	13	8	0.4904	7.4808	8.6585	8.1538
tot_info	32	170	0.6483	20.7483	23.9026	26.5625
sed	291	172	0.3750	237.0460	219.1694	281.0636
space	479	457	0.2045	681.1022	471.8260	938.8111
grep	404	694	0.2565	486.0986	353.9141	584.2129
gzip	95	161	0.3720	77.4090	71.2136	94.0158
flex	86	104	0.4488	61.9395	61.7828	67.5640

Table 5.2: Experimental results on real programs

uniform, the information gain per test will not be a constant during the diagnosis, altering the shape of C_d. It can be seen how initially both model and real behavior agree and, at some point where the information gain decreases, the descent of IG slows down. These can be explained by looking at the histograms, that represent the probability density function of the coverage density, per test, and the shape of the $IG(\rho)$ curve. Test suites with a good effectiveness have a majority of tests with large IG values, whereas space or grep have a small number of tests that provide a fast C_d decay, and quickly transition to the majority of tests with low IG.

The AS simulations are affected by non-uniformity, and it can be seen how the AS descent slows down. However, because AS is not IG-based, but a simplified version of it, AS evolves more slowly causing an overestimation of the diagnostic effort and effectiveness.

5.5 Related Work

To the best of our knowledge, there is no previous work on metrics to predict the diagnostic effectiveness of a test suite. There are, however, different works on test suite analysis that are aimed at measuring either the diagnostic or failure detection capabilities of test suites.

Test suite reduction and selection techniques include either explicitly or implicitly a measurement of the fault detection capability of a test suite. Regression test suite reduction techniques have the aim of eliminating obsolete and redundant tests [Harrold et al., 1993a, Rothermel et al., 1998, Heimdahl and George, 2004]. Test case selection [Rothermel and Harrold, 1993, Rothermel and Harrold, 1996, Rothermel and Harrold, 1997] for regression testing is based on the same principle, but applies the reduction on every execution to remove tests that have

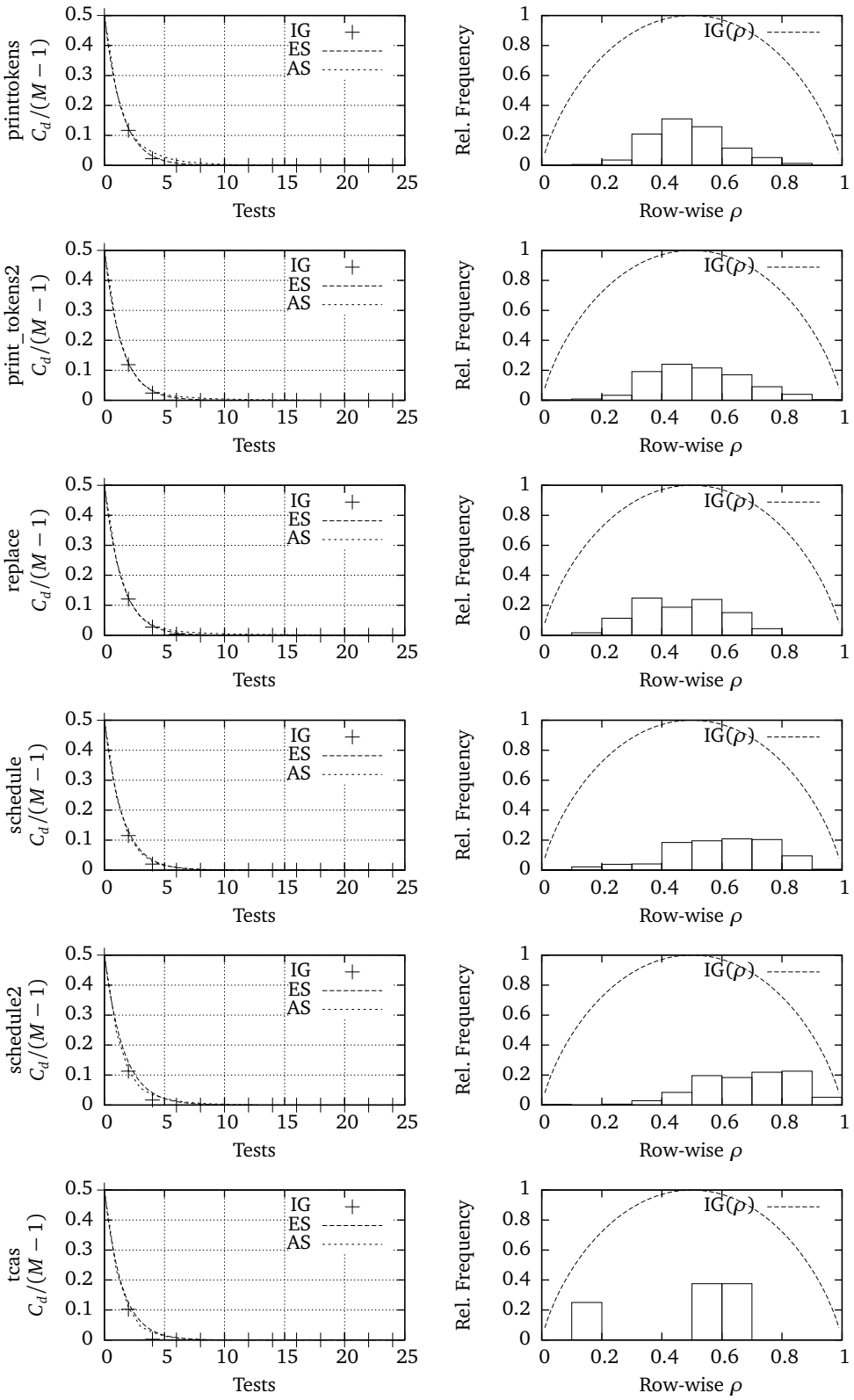


Figure 5.7: C_d prediction models and probability distribution of ρ

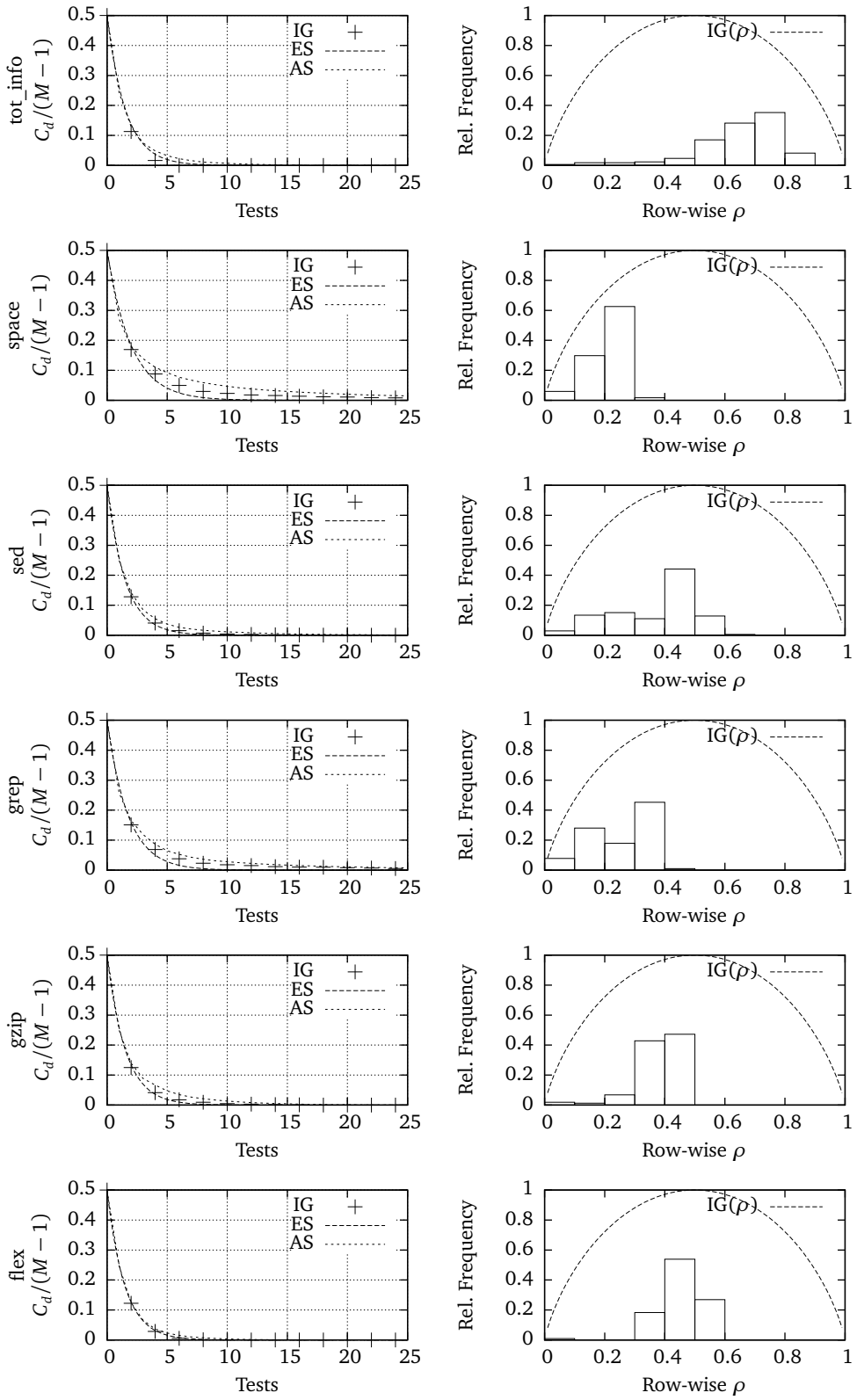


Figure 5.8: C_d prediction models and probability distribution of ρ (cont.)

not changed their outcome. Our measurement of diagnostic effectiveness can be used as base for defining criteria for test suite reduction and selection.

Baudry *et al.* [Baudry et al., 2006] propose enhancing test suites by adding new test cases that increase the number of *dynamic basic blocks* (i.e., reduce the size of ambiguity groups), thus reducing diagnostic uncertainty, hence, C_d. Their approach does not guarantee improving diagnostic effectiveness. Brodie *et al.* perform test suite reduction with the criterion of diagnostic ambiguity in the computer networks domain [Brodie et al., 2002].

Prior research work has tried to establish the connection between fault detection and fault localization capabilities. Hao *et al.* [Hao et al., 2005] study the effect of reduction of redundant tests on the Tarantula SC [Jones et al., 2002], concluding that reduction may actually have a beneficial effect on diagnostic performance. However, their conclusions partially contradict with those of Yu *et al.* [Yu et al., 2008] for a larger sample of programs, who conclude that test-suite reduction, especially coverage-based reduction, has a negative influence on the quality of the diagnosis. Unlike these works, our research focuses on the influence of the test suite in diagnostic effectiveness, rather than on diagnostic or fault detection performance.

5.6 Summary

In this chapter we have presented an analytical model and a simulation algorithm to predict the diagnostic effectiveness of test suites prioritized with the information gain algorithm, which is known to deliver the best diagnostic effectiveness.

Our research is motivated by our observation that the improvements on diagnostic effectiveness achieved by the information gain algorithm is not always much better than a randomly ordered test suite. We have investigated what structural characteristics of the coverage matrix induced by a test suite are desirable to obtain a good effectiveness.

Thanks to our models, we are now able to decide whether the computationally intensive IG will provide a good diagnostic effectiveness, or whether we should opt for a different prioritization technique, or even no prioritization. Second, given its low complexity, we can use our analytical model to define a test criterion for the generation or improvement of test suites. It can, for example, be integrated as fitness function in a genetic algorithm as it is done in [Baudry et al., 2006] with the TfD criterion.

6

Runtime Testability Measurement

Runtime testability is the degree to which a system or a component facilitates runtime testing without being extensively affected. Runtime testability is based on two main pillars: test sensitivity, which characterizes which operations interfere with the state of the running system during testing, and test isolation, which are the means test engineers have of preventing test operations from interfering with the system.

Ultimately, the implementation of isolation measures can be represented as a cost. In order to test the system, an additional expense in the test preparation phase is required, in terms of the cost of isolation techniques. Therefore, the preparation costs need to be minimized, by means of a preparation plan an optimal combination of modules of the system that have to be made runtime testable.

6.1 Runtime Testability **93**
6.2 Runtime Testability Measurement **96**
6.3 Theoretical Validation **99**
6.4 Empirical Validation **103**
6.5 Related Work **107**
6.6 Summary **108**

Critical and high-availability systems, such as air traffic control systems, systems of the emergency units, and banking applications, are becoming more and more complex and dynamic. The number and complexity of the components that form the systems is growing. Moreover, in the case of Systems of Systems, or Service Oriented Architectures components may not be available until deployment time, e.g., third party external services. Components can be even unknown at deployment time.

The testing process of these kind of systems was traditionally performed either in a separate, identical copy of the system, or by taking the system off line. This cannot be done anymore for modern systems such as the ones already mentioned [Brenner et al., 2007a, Dumitras et al., 2009, González et al., 2008b]. In a service-oriented system where many services are external to the organization, duplicating the system's components and their context is too complex, if possible

at all. Taking the system off line is also not possible given the high availability requirements of such systems.

Runtime testing is emerging as the solution for the validation and acceptance testing of the above systems. Runtime testing is a testing method that has to be carried out in the final execution environment of a system. It has been proposed for component-based systems [Brenner et al., 2007b, González et al., 2008a, Suliman et al., 2006], web services [Brenner et al., 2007a], and for Java applications [Murpy et al., 2009].

Testing cost needs to be minimized, while maximizing the confidence in the integrated system. Many approaches have been aimed at optimizing the trade-off between coverage and execution cost [Elbaum et al., 2002, Li et al., 2007, Smith and Kapfhammer, 2009, Yu et al., 2008], or between reliability and execution cost [Huang and Lyu, 2005, Okumoto and Goel, 1980]. However, in practice many parts of the system cannot be tested because they would affect the system in critical ways that are difficult to control or impossible to recover from. For instance, firing a missile while testing part of a combat system. In order to test those parts, an additional expense in the test *preparation* phase is required, both in software development, (e.g., test wrappers, testable components, simulators) and in hardware (e.g., an extra processor, more memory, higher capacity data links). Therefore, not only the execution costs of testing, but also the preparation costs need to be minimized, by means of a *preparation plan*: an optimal combination of modules of the system that have to be made runtime testable.

Runtime testability, therefore, constitutes an upper bound on what can be subsequently achieved in the test *execution* phase. The optimization problem of preparation cost with respect to runtime testability has only been addressed very recently. Previous work [González et al., 2009a, Gonzalez-Sanchez et al., 2009] defined runtime testability in terms of coverage, and introduced RTM (Runtime Testability Measurement) to predict the runtime testability of a system based on its structure and the location of untestable components.

In particular, this chapter makes the following contributions:

1. We present a new, improved model and definition of RTM that drops the most unrealistic assumptions.

2. We provide a measurement-theoretical characterization of RTM. We prove that it fulfills the requirements of a ratio scale metric, allowing us to perform arithmetic and statistical operations meaningfully.

3. We perform an empirical evaluation of RTM based on experiments performed on two real systems, demonstrating the improvement with respect to the old definition of RTM.

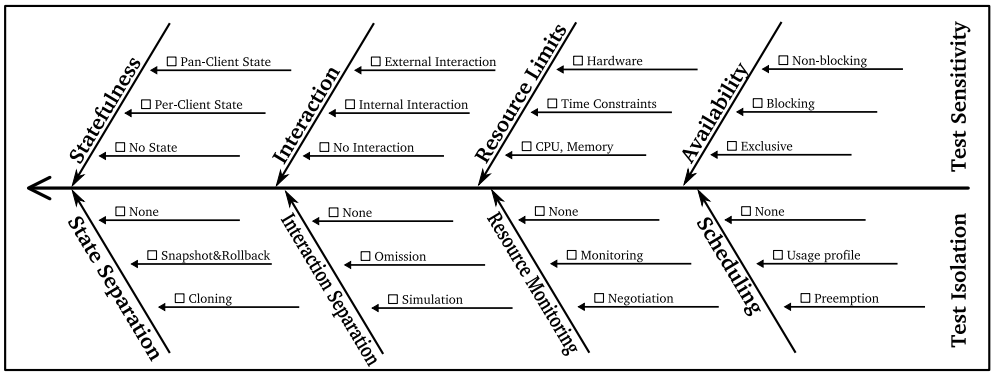


Figure 6.1: Qualitative factors that affect runtime testability

The chapter is organized as follows. Section 6.2 introduces RTM. In Section 6.3 its theoretical characterization is performed. The empirical study on the accuracy of RTM is presented in Section 6.4. Related work is presented on Section 6.5. Section 6.6 wraps up the chapter and addresses future work.

6.1 Runtime Testability

Runtime testing will interfere with the system state or resource availability in unexpected ways, as the production state and data of the system will mix with the testing. Even worse, test operations may trigger events outside the system's boundaries, possibly affecting the system's environment in critical ways that are difficult to control or impossible to recover, e.g. firing a missile while testing part of a combat system.

The fact that there is interference through runtime testing requires an indicator of how resilient the system is with respect to runtime testing, or, in other words, what adverse effects can be caused by tests on the running system. The standard definition of testability by the IEEE can be rephrased to reflect these requirements, as follows:

> *Runtime Testability is (1) the degree to which a system or a component facilitates runtime testing without being extensively affected; (2) the specification of which tests are allowed to be performed during runtime without extensively affecting the running system.*

This definition considers both (1) the characteristics of the system and the extra infrastructure needed for runtime testing, and (2) the identification of which test cases are admissible out of all the possible ones.

Runtime testability is based on two main pillars: *test sensitivity*, and *test isolation*. We will introduce the main factors that have an impact on both of them. Figure 6.1 depicts a fish bones diagram of them.

Test Sensitivity

It characterizes which operations, performed as part of a test, interfere with the state of the running system or its environment in an unacceptable way. In this section we will describe four of the main factors that have an influence on the test sensitivity of a component: a component having internal state, a component's internal/external interactions, resource limitations, and system availability.

Component State

Knowing if the component exhibits some kind of external state (i.e. the result of an input does not depend only on the value of the input itself, but also on the value of past inputs) is an important factor of test sensitivity. In traditional "off-line" testing, this is important because the invocation order will have an effect on the expected result of the test. In the case of runtime testing, knowing if a component has state is important for two additional reasons. Firstly because the results of runtime tests will be influenced by the state of the system if not handled correctly, and secondly, because the state of of the system could be altered as a result of a test invocation.

An interesting distinction, made in [Brenner et al., 2007a], is whether the state of the component under test can influence in the states of other components other than the tester component (i.e. each user sees the same common state) or not (i.e. each user sees a different state).

Component Interactions

On many occasions, components will use other components from the system, or interact with external actors outside the boundaries of the system. These interactions have the potential of initiating other interactions, and so forth. This means that the runtime testability of a component depends on the runtime testability of the components it interacts with during a test.

All these interactions will likely cause interferences with the state of the running system by changing the state of any of the components in the collaboration. In some cases, some of these interactions will cross the boundaries of the system and affect the states of other systems, and this may be difficult to prevent and fix. In the worst case, the interaction will reach "the outside world" by sending some output that will enable a physical output that may be impossible to undo, for example, firing a missile.

The main implication of interactions, besides altering the real world, is that the runtime testability of a system cannot be completely calculated if the some of the components are not known (something very common in a dynamic system like a Service Oriented System).

Resource Limitations

The two previous sensitivity factors are mainly affecting functional requirements of the system. However, runtime testing may also affect non-functional requirements. Because runtime tests will be executed on the running system, the load of these tests will be added to the load caused by the normal operation of the system. In some cases it will exceed the available resources of the system such as processor or memory usage, timing constraints, or even power consumption restrictions.

Appropriate measures such as the ones presented later on this chapter, must be implemented to ensure that runtime tests do not affect the availability of resources for the components that need them, or that, if the availability is impaired, the affected components can recover.

Availability

The availability requirements of the system in which the testing is going to be performed is also a factor. There exist two possibilities: if the component is going to be active only for testing purposes (exclusive usage), or for both the testing and normal service (shared usage). In a shared configuration two distinctions can be made: blocking and non-blocking. The first means that production operations will be blocked or rejected while the test is being performed, impairing the availability of the services provided by the components. If a component has a high availability requirement, runtime testing under this circumstances cannot be performed. In the second case, test invocations can be interleaved with production invocations and the component is able to distinguish between testing and production requests.

Test Isolation

Test isolation techniques are the means test engineers have of preventing test operations from interfering with the state or environment of the system, and of the production state and interactions of the system from influencing test results. Therefore, they represent the capability of a system to enable runtime testing by providing countermeasures for its test sensitivity.

A prerequisite for applying test isolation techniques is a prior study of test sensitivity in order to find the runtime testability issues that will make runtime testing to (partially) fail. In the following paragraphs we present some ideas for the development of test isolation techniques that could be evaluated in further work, by using our measurement in terms of testability gain and implementation effort.

State Separation

State separation techniques are the counterpart of state sensitivity. They aim to separate the in-service state of a component from the testing state. If the component has an observable state, Suliman et al. [Suliman et al., 2006] propose a solution based on three levels of sophistication. The first level consists of block-

ing the component operation while it is being tested. The second level proposes cloning the component with special support from the runtime environment. The last level relies on special testing sessions supported by the tested components to provide state isolation on components that cannot be cloned easily.

Interaction Separation

Interaction separation can be applied to component interactions that propagate through the system and affect other components, and, in particular, the external environment of the system. When interactions cross the system boundary two possible isolation solutions can be foreseen: omission of the output, or simulation. Omission of an output consists of suppressing the output, as if it had never occurred. This is possible only if the rest of the test does not depend at all on the effects that output might have. If a response is expected, in the form of an input or external event, then the external system will have to be replaced by a simulator, or a mock component.

Resource Monitoring

To prevent test cases from exhausting the resources of the system, resource monitoring techniques can be applied. A simple monitoring solution would be to deny or postpone the execution of the test case if this needs more resources than currently available, for example if system load grows over a certain threshold [Brenner et al., 2007b]. A more advanced possibility would be to allow components and tests to negotiate the resources needed for specific tests.

Scheduling

Tests can be scheduled to preserve the availability of the components, aiming to control how, in what number, and at what moment test cases are allowed to be executed in the system. For example, some test cases would only be executed when the component is less needed, akin to what was proposed for the reconfiguration phase in [Matevska and Hasselbring, 2007]. If a component is blocked by a test, but there is a certain service operation that cannot be put on hold anyway, the test case could be pre-empted from the system to satisfy the service call.

6.2 Runtime Testability Measurement

Runtime testability is the degree to which a system can be runtime tested. A numerical measurement for the runtime testability of a system can be defined in terms of what amount of fault sites in the system can be runtime tested, relative to the maximum test coverage attainable by the system testers under runtime testing conditions.

Runtime testability can be expressed empirically by the maximum coverage that

can be reached by runtime tests, defined as

$$ERT = |Cov| \tag{6.1}$$

where Cov represents the set of possible fault sites (according to a coverage criterion) which can be covered by runtime tests. The rationale behind this definition is that if a fault site cannot be exercised in testing, the number of faults that remain undetected in the system after runtime testing will be higher, i.e., the fault detection capability of tests will be higher.

Naturally, we need to predict the value of runtime testability before actually implementing the preparations if we are to use runtime testability in planning. The Runtime Testability Measurement (RTM), was defined with this purpose. It focuses on the preparation costs, predicting runtime testability independently of the nature of the runtime tests that may be performed. In this section we will present the formal definition of RTM and the prediction model used to obtain it.

Model of the System

Systems are formed by components bound together by their service interfaces, which describe the operations supported by the given service. Furthermore, we cannot assume that service interfaces work in isolation. Some of them will require other services in the system ("required" interfaces of a component). Therefore, our model has to consider *dependencies* both internal to a component (i.e., to offer a service, the component requires other), and external (i.e., binding to the actual provider of the required service). It must be taken into account the fact that operations can be used in tests only if the operations they depend on can be used, transitively.

In order to estimate the runtime testability of a system, we use a dependency graph, which captures the architecture of the system at the moment of testing. This way of representing dependencies is known as Component Interaction Graph (CIG) [Wu et al., 2001].

A CIG is defined as a directed graph with weighted vertices, $CIG = \langle V, V_0, E, c \rangle$, where

- $V \equiv V_P \cup V_R$: vertices in the graph, formed by the union of the sets of provided and required operations by the components' interfaces.
- $V_0 \subseteq V$: input operations to the system, i.e., operations directly accessible to test scripts.
- $E \subseteq V \times V$: edges in the graph, representing dependencies between operations in the system. E.g., if $(v_1, v_2) \in E$, v_1 depends on v_2.
- $c : V \rightarrow \mathbb{R}^+$: function that maps a specific operation to the preparation cost that is going to be optimized. This function could be n-dimensional but for

the sake of simplicity in the rest of the chapter we will assume a single dimension.

Although a very simple model, the runtime dependency graph has two main advantages. First, it is detailed enough to identify key runtime testability issues to the individual operations of components that cause them. Second, it is simple enough so that its derivation from the component's design and the system's runtime architecture is easy, and its composition is a tractable problem.

Because of the lack of state and control flow information in the CIG model, the following assumptions are made about the actual behavior of the system:

1. Control flow paths are independent.
2. When the control flow reaches an operation (vertex) it will always propagate to all dependencies (edges).

The first assumption is needed because the CIG does not reflect data dependencies between different operations, i.e., the fact that in order to be able to follow a certain path, some other paths have to be followed first. For example, if a user authentication path is not testable, paths requiring an authenticated user will not be testable either. This assumption then does not hold in that case.

The second assumption is necessary because we do not know which edges in the CIG will be traversed by a test case. In the extreme case, the interaction might propagate through all edges, affecting all reachable vertices.

These assumptions will be a source of imprecision, the impact of which will be evaluated in Section 6.4. In future work, the model will be extended with additional information to relax these two assumptions.

Definition of RTM

Following our model and assumptions, the total preparation cost needed to involve an operation in a runtime test, taking into account the individual preparation costs of all the operations it depends on, is defined as

$$C(v_i) = \sum_{v_j \in S_{v_i}} c(v_j) \tag{6.2}$$

where v_i and v_j are operations, and S_{v_i} is the set of successors of vertex v_i, i.e., all the vertices reachable from v_i including v_i itself.

The initial definition of RTM considered only those operations where $C(v_i) = 0$, defining RTM as

$$RTM_{old} = |\{v_i \in V : C(v_i) = 0\}| \tag{6.3}$$

This definition assumed that any operation on any interface of any component in the system can be invoked directly by test scripts, i.e., that every operation is an input operation. This is unrealistic in most cases. In this chapter we drop this assumption by means of the set of possible input vertices V_0, by requiring that operations can be reached from a testable input vertex $v_0 \in V_0$, i.e., that $v_i \in S_{v_0}$ and whose $C(v_0) = 0$. RTM can be then defined as

$$RTM = |\{v_i \in V : C(v_i) = 0 \ \wedge \ \exists v_0 \in V_0 : v_i \in S_{v_0} \wedge C(v_0) = 0\}| \qquad (6.4)$$

This value can be divided by $|V|$ in order to obtain a normalized metric that one can use to compare the runtime testabilities of systems with different number of vertices. However, such application has important implications on the theoretical requirements on the metric, as we will observe in Section 6.3.

6.3 Theoretical Validation

In this section, we establish the characteristics of the RTM measurement from a measurement-theoretical point of view. It allows us to identify what statements and mathematical operations involving the metric and the systems it measures are meaningful and consistent. We will concentrate (1) on RTM's fundamental properties, and (2) on its type of scale.

Fundamental Properties

In this section, we will study the properties required for any measurement. These properties determine whether RTM fulfills the minimal requirements of any measurement, i.e., whether it actually creates a mapping between the desired empirical property and the characteristics of the system, that can be used to classify and compare systems. The properties were described by Shepperd and Ince in [Shepperd and Ince, 1993] through an axiomatic approach.

1. Axiom. *It must be possible to describe the rules governing the measurement.*

This is satisfied by the formal definition of RTM and the CIG.

2. Axiom. *The measure must generate at least two equivalence classes.*

$$\exists p, q \in CIG : RTM(p) \neq RTM(q)$$

The two different CIGs in Figure 6.2 are assigned two different RTM values, therefore proving this axiom. A node marked with a $\times$ represents an operation where $c(v) > 0$.

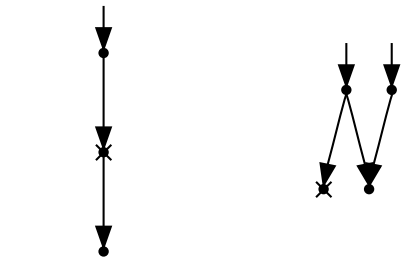


Figure 6.2: Two equivalence classes for RTM

3. AXIOM. *An equality relation is required.*

This axiom is satisfied given that our measurement is based on natural numbers, for which an equality relation is defined.

4. AXIOM. *There must exist two or more structures that will be assigned the same equivalence class.*

$$\exists p, q \in CIG : RTM(p) = RTM(q)$$

Figure 6.3 shows a collection of systems belonging to the same equivalence class, proving this axiom.

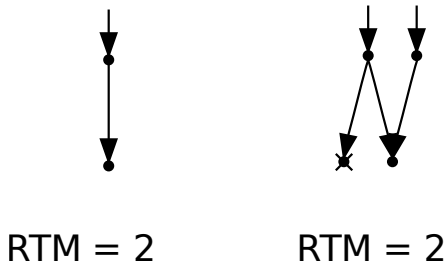


Figure 6.3: Equivalent CIGs for RTM

5. AXIOM. *The metric must preserve the order created by the empirical property it intends to measure. This axiom is also known as the Representation Theorem.*

$$\forall p, q \in CIG : p \underset{rt}{\succeq} q \Leftrightarrow RTM(p) \geq RTM(q)$$

where $\underset{rt}{\succeq}$ *represents the empirical relation 'more runtime testable than'.*

This last axiom means that for any two systems, the ordering produced by the empirical property "runtime testability" has to be preserved by RTM. It is possible

to find systems for which this axiom does not hold for RTM, because of the assumptions that had to be made (see Section 6.2). However, we can empirically assess the effect of these assumptions on the consistency and accuracy of RTM. An empirical study about the accuracy of RTM is presented in Section 6.4.

Type of Scale

The theoretical characterization of the metric's scale type (i.e., ordinal, interval, ratio, absolute) determines which mathematical and statistical operations are meaningful. This is important, because certain optimization algorithms require specific mathematical operations that might not be meaningful for RTM, for example for defining heuristics as we will see in Chapter 7.

The weakest type of scale is the ordinal scale, i.e., such that RTM creates an order of systems in terms of runtime testability. Assuming RTM satisfies Axiom 5, i.e., it preserves the empirical ordering of runtime testability, then, by definition, RTM defines a homomorphism from runtime testability to the Natural numbers. Therefore, by the ordered nature of Natural numbers, we can assert that RTM can be used as an ordinal scale of measurement. As will be noted in Section 6.4, in practice this is true only for systems in which our assumptions about control flow and dependencies hold.

Above ordinal scales, we can find ratio scales, which allow comparisons of the distances between values, and with an absolute zero value. In order to be able to use RTM as a ratio scale measurement, in addition to the requirements for the ordinal type of scale being satisfied, a concatenation operation with an additive combination rule [Zuse, 2000] must exist.

When defining such concatenation operation, our goal is to compare systems or solutions for the same system. It's not necessary to consider, thus, the influence of adding or removing connections between two disjoint systems when combined. Therefore, a meaningful concatenation operation for this goal is creating the union of both systems by disjoint union of their CIG models. This operation, $\cup : CIG \times CIG \rightarrow CIG$, can be defined as $A \cup B = \langle V, V_0, E, c \rangle$, where

- $V \equiv V_A \cup V_B$
- $V_0 \equiv V_{0A} \cup V_{0B}$
- $E \equiv E_A \cup E_B$
- $c(v) = \begin{cases} c_A(v) & \text{if } v \in V_A \\ c_B(v) & \text{if } v \in V_B \end{cases}$

For this concatenation rule, the additive combination rule

$$RTM(A \cup B) = RTM(A) + RTM(B) \tag{6.5}$$

can be used. Therefore, RTM can be used as a ratio scale with extensive structure (e.g., like mass or length), with respect to the disjoint union operation. The practical implications of these properties are discussed on Section 6.3.

Relative Values

If we divide the values of RTM by the total number of operations, $|V|$, we can obtain the relative runtime testability (rtm), to compare systems in relative terms. This transforms the measurement from a count into a ratio. Ratios, as percentages have an absolute scale [Zuse, 2000] and cannot be combined additively. For the runtime testability ratio and disjoint union concatenation operator, we can define the combination rule

$$rtm(A \cup B) = \alpha \cdot rtm(A) + (1 - \alpha) \cdot rtm(B) \tag{6.6}$$

where $\alpha = |V_A|/(|V_A| + |V_B|)$.

This combination rule is not additive, in order to state the effect of a combination of two systems we need more information than RTM, namely the size relation α.

Summary

We will highlight the most relevant practical implications of the theoretical properties of RTM obtained in the previous section.

Because we proved that RTM fulfills the minimal properties of any measurement, RTM can be used to *discriminate* and *equalize* systems. Therefore, the statements '*system A has a different runtime testability than B*', and '*systems A and B have the same runtime testability*', are meaningful. Moreover, as we proved RTM has an ordinal scale type, RTM can be used to *rank* systems. The statement '*system A has more runtime testable operations than B*' becomes meaningful, and this enables us to calculate the median of a sample of systems, and Spearman's rank correlation coefficient.

Furthermore, by proving the ratio scale for RTM, it can also be used to *rate* systems, making the statement '*system A has X times more runtime testable operations than B*' a meaningful one. This allows performing a broad range of statistic operations meaningfully, including mean, variance, and Pearson's correlation coefficient.

RTM can also be used alone to reason about the composition of two systems. Due to its additive combination rule, '*systems A and B composed, will be* $RTM(A) + RTM(B)$ *runtime testable*' is a meaningful statement, provided that A and B are disjoint. This is not true for the relative rtm, as additional information about the relationship between the systems (α, the size relation between the systems) is needed.

6.4 Empirical Validation

In this section we conduct a number of experiments in order to empirically determine how accurate RTM is with respect to the empirical property of "runtime testability" (ERT).

Two systems were used in the experiment: (1) AISPlot, a system-of-systems taken from a case study in the maritime safety and security domain, and (2) WifiLounge, an airport's wireless access-point system[1]. For the experiments in this chapter we will not use the real preparation costs c_i, but random costs, as we explain in the next subsection.

Experimental Setup

To obtain the value of RTM, vertices are first classified into testable and untestable by means of $C(v)$ (see Eq. 6.2). To obtain the value of $C(v)$, we assumed that execution always propagates through all branches. This will introduce cases where the cost will be overestimated, leading to a number of false positives (i.e., vertices classified as not testable when they are testable in reality). Conversely, the assumption that paths are independent will produce cases where cost is underestimated, introducing false positives (i.e., vertices classified as testable when they are not testable in reality). Therefore, our goal is to assess the influence of the assumptions made in order to keep the model simple, in the number of false positives and false negatives of this classification, and in the final value of RTM.

In order to have a baseline for comparison, the naive approach of just counting directly testable operations was used, in addition to RTM_{old} and RTM, This measurement is defined as:

$$NTES = |\{v_i \in V : c(v_i) = 0\}| \tag{6.7}$$

For this experiment we are not so much concerned about the preparation cost (as any $c(v_i) > 0$ will have the same effect regardless of the concrete value of $c(v_i)$), but about the fact that a vertex is testable or not. 500 variations of the AISPlot and WifiLounge systems with different RTM values were generated by choosing the untestable vertices by randomly sampling in groups of increasing size from 2 to 30 untestable vertices.

The value of ERT to perform the comparison was obtained by executing an exhaustive test suite in terms of vertices and execution paths. The set of operations covered when executing each test case was recorded. If a test case used any untestable operation, none of the operations covered by the test were counted. The validity of this approach is discussed in Section 6.4.

[1] `http://kraken.cs.cas.cz/`

Results

From each system and metric pair in this experiment, we recorded the following data:

- M_{set}: Set of operations classified as testable, as determined by the set definitions in Equations 6.3, 6.4 and 6.7.
- Cov: Set of operations covered, determined by running the test cases (we assume $|Cov| \approx ERT$).
- $f_p = (|M_{set} - Cov|)/|M_{set}|$: false positive rate, i.e., operations wrongly classified as testable.
- $f_n = (|Cov - M_{set}|)/|Cov|$: false negative rate, i.e., operations wrongly classified as untestable.
- $\bar{e} = ||M_{set}| - |Cov||$: absolute error between the predicted and empirical runtime testabilities.

	System	f_p	f_n	$\bar{e}$
NTES	AISPlot	0.942	0.000	83.487
	WifiLounge	0.713	0.000	57.093
RTM_{old}	AISPlot	0.882	0.107	15.012
	WifiLounge	0.577	0.079	27.664
RTM	AISPlot	0.411	0.111	2.418
	WifiLounge	0.306	0.128	9.101

Table 6.1: False positive/negative rate, and error

Table 6.1 shows the rates of false positives and false negatives, along with the absolute error, averaged over 500 runs. The deviation between RTM and the actual covered operations for each sample can be seen in the three plots in Figures 6.4, 6.5, and 6.6. The plots contain the combined results of the two systems with NTES, RTM_{old}, and RTM, respectively. The dashed line represents the ideal target. Any point above it, constitutes an overestimation error, and below it, an underestimation error.

NTES In Figure 6.4, it can be seen that NTES has an extremely high error caused by its high false positive rate (94% and 71%). NTES has no false negatives as it classifies as untestable only the vertices which are directly untestable, disregarding dependencies.

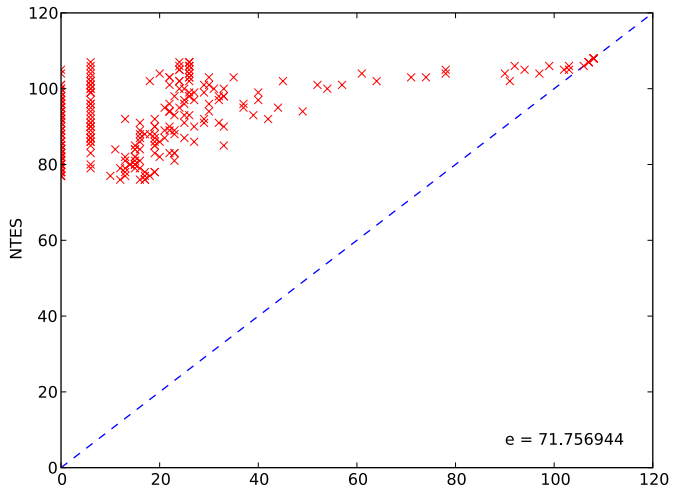


Figure 6.4: Accuracy of NTES

RTM$_{old}$ Figure 6.5 shows how, by taking control flow dependencies into account, the false positive rate of RTM$_{old}$ is lower than NTES, at the price of introducing a number of false negatives. False negatives appear because in some cases where the control flow does not propagate to all of the operations' dependencies, as we had assumed. Still, because of the assumptions that test interactions can start in any vertex, and that the test paths are independent, the number of false positives is considerable. The number of input vertices in WifiLounge is proportionally higher than for AISPlot. Hence, the number of false positives caused by this assumption is lower.

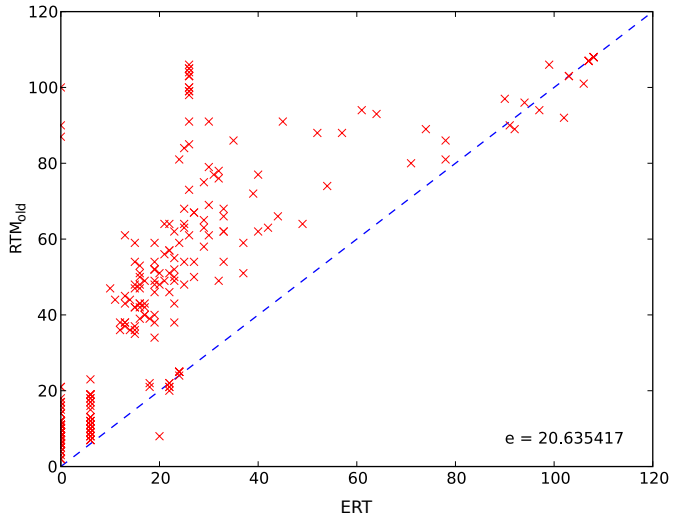


Figure 6.5: Accuracy of RTM$_{old}$

RTM Finally, in Figure 6.6 we show that by taking input vertices into account, the amount of overestimation decreases dramatically for both systems. Still, the false positive rate is significant. Therefore, we conclude that assuming that paths are not dependent is not very reasonable and needs to be addressed in future work.

The increase of false negatives makes more apparent the consequences of the assumption that control flow is always being transmitted to dependencies. The error caused by this assumption is augmented by the fact that it also applies to the input paths to reach the vertex being considered.

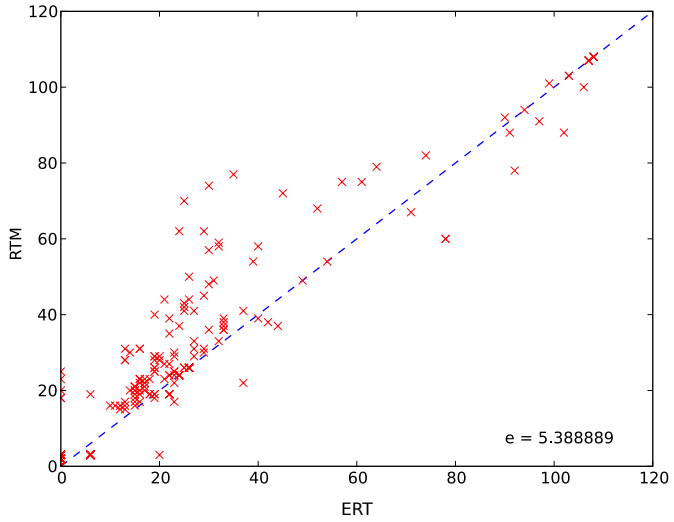


Figure 6.6: Accuracy of RTM

Threats to Validity

The main threat to the validity of our experiments is the representativeness of the used systems, given that we generated our sample from random variations of only two systems instead of from a larger population of different systems. Firstly, from the architectural point of view, Shaw and Garlan [Garlan and Shaw, 1994] mention as the most common styles, data-flow, call-and-return, independent components, and blackboard architectures. AISPlot is a representative of the data-flow architectural style, while WifiLounge is a representative of the call-and-return style. A problem we foresee for independent component and blackboard architectures, is that the derivation of the CIG can be hindered by the fact that dependencies and interacting components are never explicit as is the case for data-flow and call-and-return styles.

A second threat is the representativeness of the control-flow behaviors of both systems, with respect to the assumptions we made. Assuming that in AISPlot the interaction propagates through all dependencies means that we assume that the processing of an input will involve the entire pipeline. This is precisely the way

outputs are produced in a data-flow system. AISPlot contains also cases where the data-flow can fork (i.e., the data follows a different processing path in the pipeline, or it is dropped, depending on its content). Therefore AISPlot is a good representative of a data-flow system.

On the other hand, in a call-and-return system, taking a certain path depends on a multitude of conditional branches which make our assumption that all dependencies are used when invoking an operation seem rather weak. However, the low false negative rate and number of underestimated cases, suggest the opposite. In fact, examining the source code of WifiLounge we come to the conclusion that it has very few cases in which an operation does not use all its dependencies. Consequently, WifiLounge may not be the best representative of the call-and-return style.

A final consideration is whether our way to obtain the ERT value was adequate. To the greatest extent possible we ensured that the developed test suites covered both systems completely, and redundantly, by exercising every possible path in the CIG from every input operation of the system. This way it is ensured that if an operation is not covered, it is not because a test case is missing, but because there is either no test case that could cover it without requiring also an untestable operation, or because the operation is actually unreachable.

6.5 Related Work

Testability and testing effort was originally related to the probability of a fault causing an error, this error propagating to the output [Hamlet and Voas, 1993, Voas et al., 1991], and the error being actually detected by the test oracle [Bertolino and Strigini, 1995].

However, testability is affected by many other factors and its measurement has been studied extensively from various points of view. In this section we present the main related work in the measurement of testability.

A descriptive model of testability is presented in [Binder, 1994], consisting in six main factors: representation, test suites, implementation, test tools, built-in test, and process. Although no attempt is done at defining a universal testability metric. Some related metrics are cited for the implementation sub-factors without further consideration.

A second descriptive model of testability is presented in [Gao and Shih, 2005]. It defines a global measurement of testability based on an average of measurements for 5 individual factors. However, no evaluation, either theoretical nor empirical, is performed. Even the minimum requirement of how to obtain each of the values is missing. Hence, we cannot consider it as a software metric.

A number of potential quantitative metrics for testability, relating system structure and dependencies to regression test cost, were studied in [Jungmayr, 2002].

Their study lacks any theoretical characterization of the metrics they propose. Furthermore, a derived "reduction metric" to assess the impact of a dependency in testability is proposed without considering the scale type of the metric it is derived from. This reduction metric requires to be derived from a ratio scale metric to be meaningful. Consequently, their conclusions with regard to relating the impact of a dependency in testability should be taken with care.

A metric of testability related is proposed in [Freedman, 1991]. It is based on the idea that if a module's input and output domains are not completely defined by the module's interface contract, it will require a higher test effort (i.e., time needed to define test cases). This metric is proven to be an extensive, ratio-scale metric for the sequential and alternative concatenation rules. The metric is empirically evaluated as well.

None of the mentioned metrics addresses *runtime* testability, which motivated the introduction of RTM in previous work,and its usage to minimize preparation costs.The metric suffered a number of shortcomings and lacked validation. This chapter is the first where RTM has been empirically and theoretically validated, and where an efficient minimization algorithm has been proposed.

6.6 Summary

In this chapter we have performed an in-depth study, theoretical and empirical, of RTM, a metric designed to measure the effect that untestable operations will have on the runtime testability of a system. We have studied the theoretical characteristics of RTM, proving that it satisfies an accepted set of characteristics for every measure [Shepperd and Ince, 1993], and that it can be used as a ratio scale type. Our experiments have shown empirically that, although we are on the good path towards obtaining an accurate prediction of runtime testability, there is still some room for improvement.

7

Optimization of Runtime Testability

Runtime testing requires an extra cost to be invested in implementing appropriate measures, in order to enable runtime tests to be performed without affecting the running system or its environment. This chapter introduces a method for the improvement of the runtime testability of a system, which provides an optimal action plan considering the trade-off between testability and implementation cost. The computation of the action plan is driven by the runtime testability metric (RTM) introduced in the previous chapter.

This method is used to direct integration and test engineers in the implementation of improvement measures for the runtime testability during the integration of a system. Furthermore, it can also be applied earlier, on the design stage, providing a Design for Testability method for dynamic systems. By means of an example, we demonstrate how our method enables an optimal expenditure of resources towards runtime testability during the integration phase of a system.

7.1 A Method to Improve Runtime Testability **110**
7.2 Application Example **115**
7.3 Near-optimal Optimization **121**
7.4 Related Work **124**
7.5 Summary **125**

Once the theoretical and empirical properties of RTM have been established, RTM can be safely used for the elaboration of an optimal preparation plan with the objective of achieving the highest possible runtime testability within a given budget b. Mathematically, this problem is formulated as

$$\begin{aligned} &\text{maximise:} && RTM \\ &\text{subject to:} && \sum c(v_j) \cdot x_j \leq b, \quad x_j \in \{0,1\} \end{aligned}$$

where b is the maximum budget available, and x_j represents the decision of including vertex v_j in the preparation plan or not. In order to allow good runtime testability, integration engineers aim for a high RTM. However, not all preparation costs are equivalent, nor do they provide the same runtime testability improvement. A prerequisite for the optimization of runtime testability is the knowledge

about which items need additional preparation cost, what is that cost, and what is their influence in the rest of the system.

The main contribution of this chapter is RiTMO, a method to enhance the runtime testability of a system at design or deployment-time. RiTMO computes an action plan for the implementation of improvement measures of the system's runtime testability. For this purpose, it uses an estimated value of runtime testability based on the RTM metric introduced in Chapter 6, along with the cost of the remedial measures needed to reduce their impact. Our approach reflects the trade-off that engineers have to consider, between the improvement of the runtime testability of the system after some interferences are addressed, and the cost of the remedial measures that have to be applied. We present a detailed application example of RiTMO to a system taken from our industrial case study in maritime safety and security systems. Furthermore, since no computationally efficient algorithm exists to allow test engineers to efficiently generate a preparation plan for systems of realistic size, we introduce two low-cost greedy algorithms to be used in obtaining a near-optimal preparation plan.

The chapter is structured as follows. In Section 7.1, a method to improve runtime testability of a system is described. Section 7.2 evaluates the proposed method. Section 7.3 introduces our heuristics for sub-optimal optimization. In Section 7.4 related work is presented and compared to our research. Section 7.5 wraps up the chapter and introduces some ideas for further research.

7.1 A Method to Improve Runtime Testability

In this Section we introduce RiTMO (*RuntIme Testability Measurement and Optimisation*), a method for improving the runtime testability of a given system based on RTM. This method helps developers in identifying parts of a system where an effort has to be invested in the implementation of test isolation measures, in order to maximize the RTM for a given budget. It can also be used to compute the minimal budget required to reach a target RTM.

In a typical application scenario, a system with high availability and reliability requirements is being configured during run-time. In order to allow good testing coverage, integration engineers aim for a high RTM. By combining component design information with the system's architecture, the system integrator uses RiTMO in order to determine where the budget is best spent in order to increase runtime testability.

Another application scenario is a runtime system update. Then, the system integrator uses RiTMO to verify that sufficient runtime regression testing will be possible. If that is not the case, our method will point out which are the most worthy components of the system where where isolation measures will need to be applied before the testing of the update can be done.

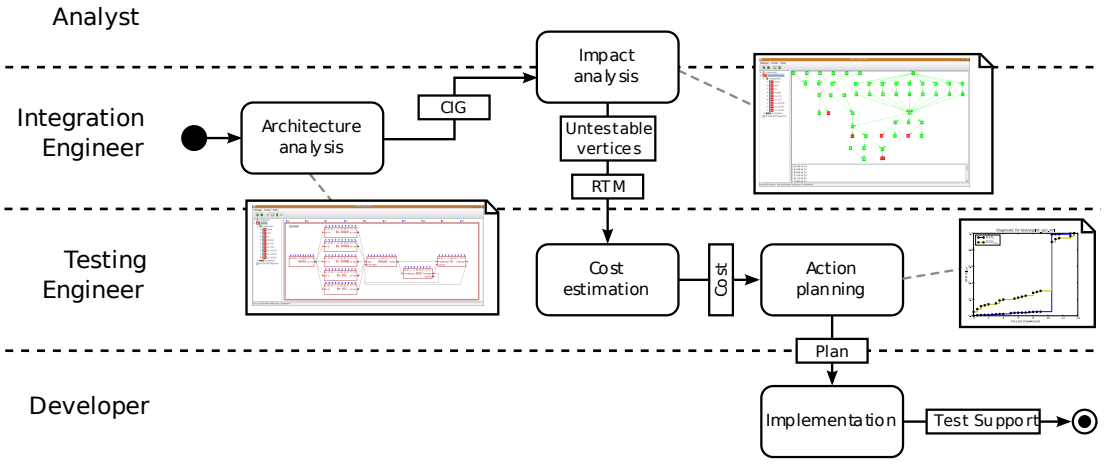


Figure 7.1: Complete workflow of RiTMO.

Figure 7.1 depicts the five main steps which compose the RiTMO method, associated to the engineering roles in charge of performing the task. It must be noted that a single person can have multiple roles. This figure is complemented by Table 7.1, in which details on the inputs and outputs of each activity are provided. The first step consists in analyzing the dependencies between each part of the system. The second step aims at determining which part cannot be tested at runtime. At this point, the RTM of the system in its current implementation can be computed. The third step determines the cost of "fixing" each of the untestable parts of the system by applying an adequate isolation technique. The fourth step uses our algorithm to obtain the optimal action plan for increasing the RTM. The fifth and final step consists in applying this action plan to effectively obtain a more runtime testable system. Each of these steps is described in the following subsections in more detail.

Step 1: Architecture Analysis

In order to compute the RTM of a system, the first step provides an architectural view of the system at the right level of granularity. In RiTMO, the architecture of the system is represented by a static dependency graph, the CIG, as depicted in Figure 7.2. This graph provides information on the call dependences between each method of each component in the system. A vertex of the graph represents a method, and an edge represents a call dependence of one method on another method.

Vertices in the CIG are obtained by examining the required and provided interfaces of each component instance. For each method of each interface, a vertex is created. If a method of a required interface is not used, i.e., no code of the component uses this part of the interface, the corresponding vertex is removed as it

Activity	Actors	Inputs	Outputs
Archit. analysis	Integration Engineer	Component designs Bindings	CIG
Impact analysis	Integration Engineer Analyst	CIG Domain knowledge Design knowledge	Untestable vertices Initial RTM value
Cost estimation	Test Engineer	CIG Untestable vertices Component designs Isolation techniques	Vertex fix proposal Vertex fix costs
Action planning	Test Engineer	CIG Untestable vertices Vertex fix costs Budget or RTM	Action plan Estimated cost Estimated RTM value
Implementation	Developer	Action plan Vertex fix proposal	Test support code

Table 7.1: Actors, inputs and outputs of each activity involved in RiTMO.

would add unreachable paths. It is important to note that when there exist multiple instances of a same component, each instance corresponds to different vertices, as their dependencies are different. Composite components are treated in a similar way: vertices are added for each of the methods in the interfaces.

Edges are generated in two independent steps. The first step of edge generation consists in finding the dependencies between the provided and required interfaces inside components. This task is challenging, because the very philosophy of component-based development hides the dependencies between interfaces: components are black boxes. There are, nevertheless, several ways to obtain this information:

- **From the specification.** If the models of the component behaviour are precise enough, it is possible to deduce for each provided functionality (interface method), which required method it depends on. For example, using sequence diagrams used to specify a component, the edge generation can be done automatically. The drawback is that even if the specification is precise enough, the implementation might be different due to optimisation (e.g. caching) or limitations (e.g. some functionalities are not implemented).

- **From the implementation, statically.** If the component's implementation is actually available, analyzing the source code will allow to precisely determine which required methods are used for every provided method.

- **From the implementation, dynamically.** Even if neither a model nor the component's source are available, it is still possible to deduce the dependencies between the methods using a tracing mechanism [Lorenzoli et al., 2008].

The component can be exercised either by using available unit test cases, or by observing it running in the actual system. The drawback of this method is that some dependencies might not be detected if the calls are not triggered by the operational profile used during tracing.

Secondly, the CIGs of individual components are composed, by adding edges from required to provided methods, which can be directly obtained from the bindings between the component instances in the system. For each connection of a required interface to a provided interface, an edge is added between the corresponding vertices. Similarly, all delegation dependencies between an interface of a composite component and the interface of the subcomponent in which it delegates, are also represented as edges.

Dependencies between components can be determined in two ways:

- **By introspection.** Runtime architecture reconstruction [Huang et al., 2006] can be applied if the component framework provides introspection functionality, such as EJB, CCM or Fractal. In this case, this step can be done entirely automatically.

- **From the specification.** If such functionality was not available, it is still possible to derive the edges either from a model of the system, such as a UML component model.

The CIG of an example composite component obtained by our method is depicted in Figure 7.2. The crossed out vertices and dashed edges represent elements that will be removed as they are unreachable (o_3 is never used, and $B.v$ cannot be used from outside C). Still, this way of deriving a CIG often leads to redundant vertices and edges, e.g., (o_1, u_1), or all the (i, m) pairs. This choice was made in order to keep the composition method of the CIG simple. Moreover, if there are modifications of the CIG due to an architectural reconfiguration, more paths will be kept compared optimised CIG, which will facilitate comparison with the old system.

Once this last step of the graph generation is done, the CIG model is complete, providing an architectural knowledge of the system with a fine granularity. The next step consists in determining which methods are not allowed to be executed during runtime testing due to their excessive impact on the running system or its environment.

Step 2: Impact Analysis and RTM

The goal of this step is to augment the CIG of the system with information on the impact of executing each method. In other words, it is necessary to identify whether each vertex of the graph requires preparation or not. The derivation of

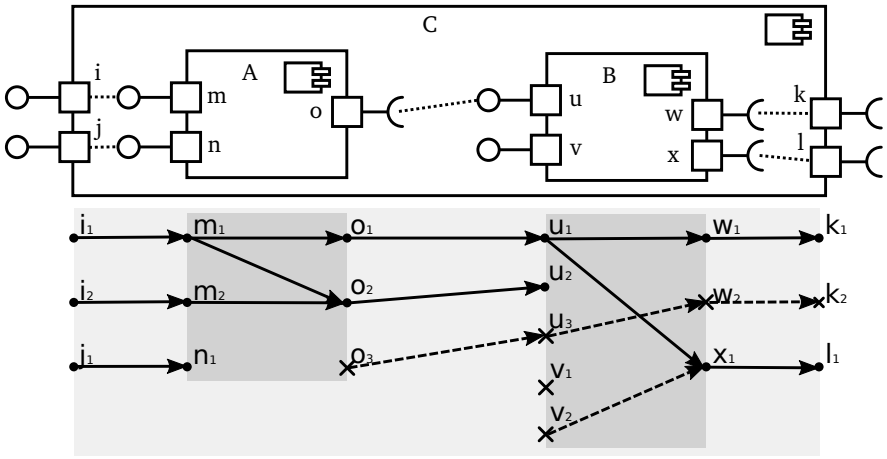


Figure 7.2: Example of a CIG representation of a system.

this information cannot be easily automated, and it is up to the integration engineer and analyst to apply their design and domain knowledge for deciding which method calls could disrupt the normal operation of the system. In the case no information is available for a certain vertex, a conservative approach should be taken.

Performing the impact analysis step requires inspecting each component specification or source code, checking if any activity in a testing context can have either an impact on the state of the system or affect the system's environment in an undesired way. In case the system is large, this can be a laborious task, however it can be greatly simplified if component vendors associate test impact meta-information to their components, so that this information is easily reusable, and provided by those stakeholders, who have the most precise knowledge about the component.

Using this annotated graph, it is possible to compute the value of RTM for the chosen coverage criteria, according to Equation 6.4. If this value is higher than the target coverage value for the system tests, the method can stop, as the system is sufficiently runtime testable. If this is not the case, improvements on the system must be performed.

Step 3: Cost Estimation

The third step of the RiTMO method consists in complementing the graph of the system with an estimation of the effort needed for the implementation of countermeasures to reduce the test impact of untestable vertices. For each vertex v_i that requires preparation, a cost, $c(v_i)$, which represents the estimated cost of test isolation, is associated. The cost unit is not fixed, but has to be uniform for the whole graph. Typically, the cost is expressed in terms of time (e.g., man-hours) or money.

Every untestable method is assessed by combining knowledge of the possible

countermeasures (e.g., making the method test-aware) with knowledge about the developer team capabilities, in order to obtain an estimation of the implementation effort. Although this step cannot be easily automated, the estimation does not need to be very precise as long as the estimations are relatively consistent with each other so that the computed plan is still valid in relative terms.

Step 4: Computation of Action Plan

The completely annotated CIG contains information about the untestable methods and the cost to make them testable. It is then possible to compute an action plan automatically for obtaining the optimal RTM given a maximum budget. Conversely, it is also possible to compute an action plan with the minimal cost to reach a fixed RTM.

A general exhaustive search algorithm could be used to obtain the optimal plan. However, in practice, this approach is not usable for systems of realistic size due to its exponential complexity. This is the reason why a heuristic near-optimal algorithm was introduced in [González et al., 2009b]. The resulting action plan in both cases consists of the list of methods to address, along with the final expected RTM value.

In order to have the most information available about the testability improvement, the optimal growth function of the system's RTM vs. fix cost can be calculated. From such graph it is easy to read what is the minimal budget required to reach a given RTM. An example is shown in Figure 7.5.

Step 5: Application of the Plan

The last step of the method consists in following the action plan by implementing the fixes for each of the method selected by the algorithm.

It is likely that the cost of addressing a vertex was under- (or over-) estimated. Therefore, it is advisable to address the methods in an order which will maximize the growth of RTM even when some of the vertices are not yet been made runtime testable, so that the most features have been made testable per unit of cost. The algorithm in Section 7.3 already provides this ordering, however, if the solution was obtained by the optimal algorithm, this order is still unknown Although obtaining the ordering is once again of exponential complexity, a heuristic algorithm could be applied to the solution's vertices to obtain a near-optimal ordering. We will present two of such heuristics in Section 7.3.

7.2 Application Example

In this section we will describe an application example of RiTMO to a component-based subsystem taken from our industrial case study.

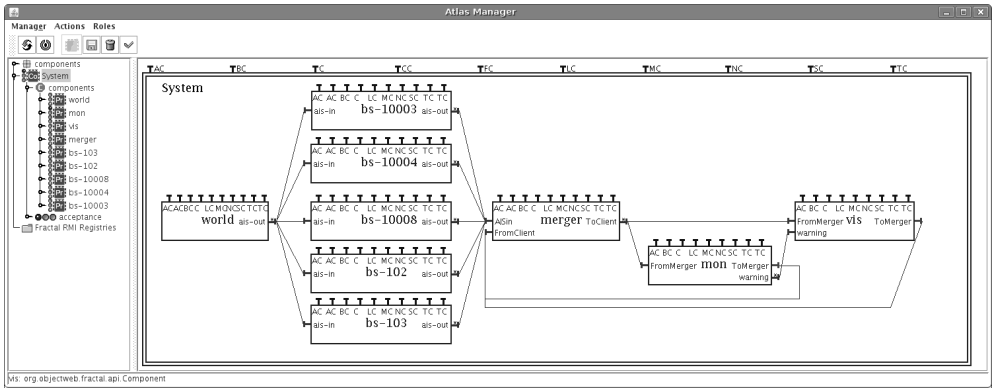


Figure 7.3: Runtime-reconstructed AISPlot Architecture

System Setup and Architecture

The system used in our integration experiment is a vessel tracking system taken from our industrial case study, codenamed AISPlot. It is part of a component-based system coming from the maritime safety and security domain. The architecture of the AISPlot system is shown in Figure 7.3.

The system is used to track the position of ships sailing a coastal area, detecting and managing potential dangerous situations. Position messages are broadcast through radio by ships (represented in our experiment by the `World` component), and received by a number of base stations (BS components) spread along the coast. If a message received from the simulator belongs to the area the base is covering, it is relayed to the `Merger` component through the `AISin` interface. `Merger` removes duplicates (some base stations cover overlapping areas), and offers a subscription service to client components for receiving updates on the status of vessels. Components interested in receiving these updates, can subscribe to receive notifications through the `Merger-Client` interface pair. The `Merger` interface offers the `Csubscribe` and `Cunsubscribe` operations for manging client subscriptions. The `Client` interface offers four operations for notification of ship status: `Cnew`, `Cpublish`, `Creply` and `Cdispose`. In the current system, there are two clients: a safety monitor, and a visualizer screen. The `Monitor` component scans all the received messages in search for inconsistencies in the data sent by the ships, in order to spot those which are less reliable and which therefore require more care from the operator. The `Visual` component draws the position of all ships on a screen in the control center, and also the warnings generated by the `Monitor` component, via the `Warning` interface.

Implementation

AISPlot is implemented in Java, on the ATLAS/FRACTAL runtime integration and testing research platform [González et al., 2008a]. It is a dynamic, reflective

component-based platform that allows the insertion, modification and removal of components at run-time. In addition, ATLAS/FRACTAL provides a way of enabling components to test the services they depend on, based on the principles of Built-In Testing (BIT) [Gross, 2005]. ATLAS provides components with the ability of requesting the execution of test cases associated with them, in order to check that the services on which they rely conform to their expectations, and of receiving a notification of when they are going to be involved in a test. At the time of this writing, ATLAS/FRACTAL does not provide means of interleaving test and nominal operations. Therefore, during a test, components only receive test invocations (this is referred to as *blocking* test execution context [González et al., 2009a]).

For the purpose of this experiment, a number of new features were introduced into ATLAS/FRACTAL:

- A CIG model can be associated to primitive components, and retrieved at runtime.
- Composite components can compose their own CIG from the CIG's of their contained components.
- Special *test support* code can be injected at run-time into any component during tests. This allows the deployment of test isolation code into the components that need it.

Application of RiTMO

Architecture analysis

Because of the reflective capabilities of ATLAS/FRACTAL, the derivation of the set of required and provided interfaces into vertices, and the creation of the edges from external connections between components in the system, were a straightforward and completely automated process. Figure 7.3 shows the reconstructed view of the system's architecture (components and run-time links) obtained by querying the reflection interfaces provided by FRACTAL.

ATLAS/FRACTAL does not provide a way of reverse-engineering primitive components to extract edges induced by internal dependencies. Therefore, this process was performed semi-automatically by source code inspection. The *call hierarchy* functionality of Netbeans[1] was used to automatically derive a call tree of each of the public operations in the implemented interfaces of each component, and locate calls to methods of required interfaces.

The CIG models obtained this way were stored as plain text files and associated to their components using the mentioned functionality of ATLAS/FRACTAL. During the system deployment, the management console of ATLAS/FRACTAL automatically

[1] `http://www.netbeans.org`

composes all the CIGs of the primitive components to obtain the complete CIG of AISPlot, which can be seen in Figure 7.4.

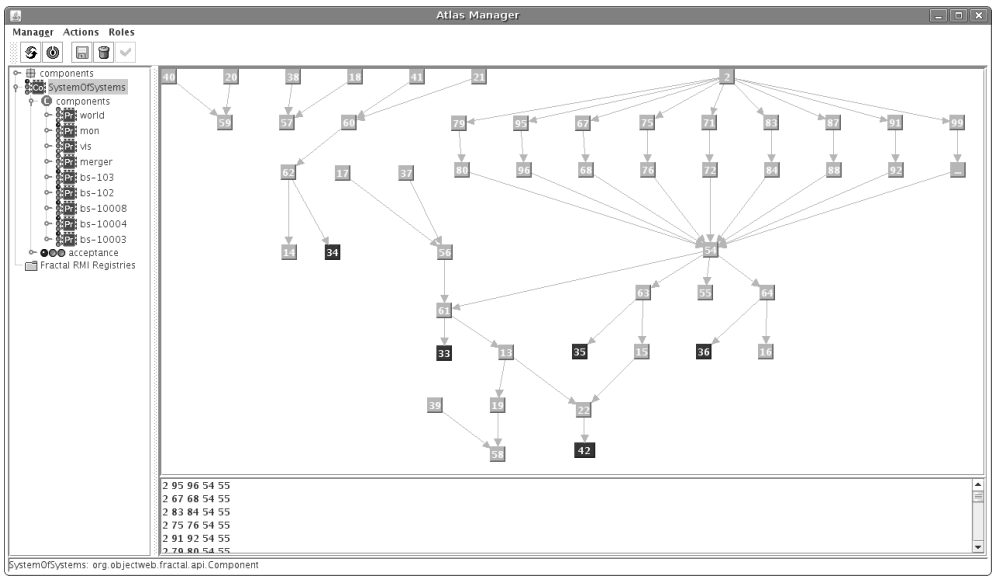


Figure 7.4: AISPlot CIG during impact analysis

Impact analysis

The `World` and `BS` components are stateless and have no interaction outside the system's boundaries. Therefore, no impact is caused by invoking their operations during a test.

On the other hand, `Merger` and `Monitor` are stateful components, as both of them store tables of received messages and vessel information internally. This information will be altered by test operations, and therefore these components are test sensitive. The merger component has to manage vessel subscriptions from clients as well. Because the set of subscribed ships will be altered during testing, this is also a source of test sensitivity.

The visualizer component stores internally a list of observed vessels (ships to which it has subscribed in `Merger`), which is already a source of test sensitivity. Moreover, the visualizer component is interacting with a real display through a socket connection. This interaction must be isolated in some way so that users of the system do not see the test vessels and warnings drawn on the display.

The preliminary estimation of the runtime testability of the system can be seen in Table 7.2. Due to the pipelined nature of the system, only a very low number of vertices and paths are testable, those which are not related to the main pipeline path, hence the extremely low RTM value.

	Total	Testable	RTM
Vertices	86	4	0.046

Table 7.2: RTM results of the impact analysis

Cost estimation and action planning

A number of ATLAS/FRACTAL runtime test support artifacts had to be developed in order to address the test sensitive operations of the system. The development cost was estimated as effort in man-hours required to program the needed isolation code.

The test sensitive operations in `Merger` and `Monitor` require maintaining separate tables for vessel accounting (and subscription management in the case of `Merger`). As both the vessel table and the subscription table are implemented inside the components, the solution we selected was to insert another instance of the component as "test support instance". In ATLAS/FRACTAL, the test support functionality automatically redirects all the calls happening during testing to this special instance.

The solution required for `Visual` is more involved, as it requires isolating the low-level drawing commands that the visualizer is sending to the display. Moreover, it was decided that drawing commands resulting from a test should not be discarded but sent to a different display so that the observability of the system under test is enhanced.

Given our previous knowledge in implementing a set of analogous measures for a system with components of similar characteristics, the cost of implementing test support for the untestable operations in `Merger` and `Monitor` was estimated to be approximately 0.5 man-hours, and the implementation of the isolation and observation code for each operation in `Visual` would have an estimated cost of 2 man-hours. Table 7.3 summarizes all the test sensitive operations in each component, along with suitable isolation techniques and their estimated costs.

Figure 7.5 shows the RTM as a function of the cost spent developing test support artefact, calculated by our testability optimisation tool. Because of the pipelined architecture of the system, all the untestable vertices in the pipeline have to be fixed to obtain a substantial improvement of testability. This can be seen on the big jump in testability in Figure 7.5 when 10.5 man-hours are dedicated to testability improvement. Without the information in the plot, it would have been difficult to find this issue.

Table 7.4 shows the final value for the RTM of the system once all the countermeasures have been applied to the untestable vertices, along with the vertices which are part of the plan.

v_i	Operation	Sensitivity	Isolation	Cost
13	monitor.Cnew	state	separate state	0.5
14	monitor.Creply	state	separate state	0.5
15	monitor.Cpublish	state	separate state	0.5
16	monitor.Cdispose	state	separate state	0.5
33	visual.Cnew	state, interaction	separate, redirect	2
34	visual.Creply	state, interaction	separate, redirect	2
35	visual.Cpublish	state, interaction	separate, redirect	2
36	visual.Cdispose	state, interaction	separate, redirect	2
42	visual.Warning	interaction	redirect output	2
54	merger.MessageIn	state	separate state	0.5
58	merger.Csubscribe	state	separate state	0.5
59	merger.Cunsubscribe	state	separate state	0.5

Table 7.3: Untestable operations in AISPlot

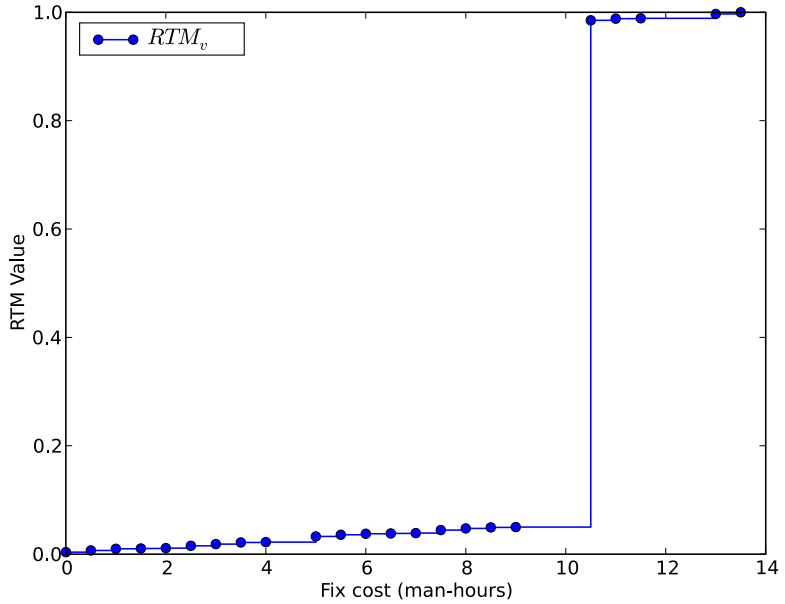


Figure 7.5: Evolution of RTM for different action plans in AISPlot

Discussion

In our example, we have shown how this method can be used in a component-based system, revealing a great testability gap (from 30% to 90% runtime coverage prediction) which would not have been easy to identify by simply looking at the architecture of the system.

Although in our example RiTMO is applied at a late stage of the system deployment, this is not the only possible application scenario. RiTMO can be applied

	Total	Testable	RTM
Vertices	86	77	0.895
Plan	13,15,16,33,35,36,42,54,58		
Cost	10.5 man-hours		

Table 7.4: RTM results of the action plan

to drive a *Design for Runtime Testability* phase in the earlier stages of the development life-cycle of a system, given that the knowledge about which components will be integrated, and their test sensitivities, are available. This way, the component's design can take into account runtime testability concerns from a much earlier stage, saving resources in the implementation of isolation measures later on the development life-cycle.

As final positive remark, mention that even though in our application example we have relied on specific support features provided by the ATLAS/FRACTAL platform, e.g., model meta-information, introspection and test code injection, RiTMO does not depend on these functionalities. It is still possible to apply it to systems where the platform does not provide runtime testing-specific features.

Nevertheless, there are number of issues that have to be considered concerning the applicability of the method. First, for the practical application of the method, it must be taken into account that the quality of the plan depends greatly on the quality of the predicted costs. Care should be taken in obtaining a development cost estimation model to obtain realistic (and therefore, useful) improvement plans. Second, as a system can have a very large number of components and vertices, a (semi-)automatic tool that supports all the steps of RiTMO is needed. Although Figures 7.3 and 7.4 show our preliminary implementation work, it is in a very early stage, especially with respect to finding the components that contain test sensitive features. Finally, for better cost estimation, the process for defining isolation costs could refined to allow dependencies between vertex fixes, as often several operations of a component have to be fixed as a whole.

7.3 Near-optimal Optimization

Unfortunately, combinatorial optimization problems, such as the optimization of runtime testability, are NP-Hard. Therefore, an optimal preparation plan cannot be calculated in polynomial time. The graph in Figure 7.6 shows the exponential time it takes to perform the calculation of the optimal preparation plan on our experiment platform as a function of the number of untestable operations $|U|$, averaged over 25 runs.

As can be seen, the time required to compute the optimal preparation plan grows exponentially with the number of untestable vertices $|U|$ and, therefore,

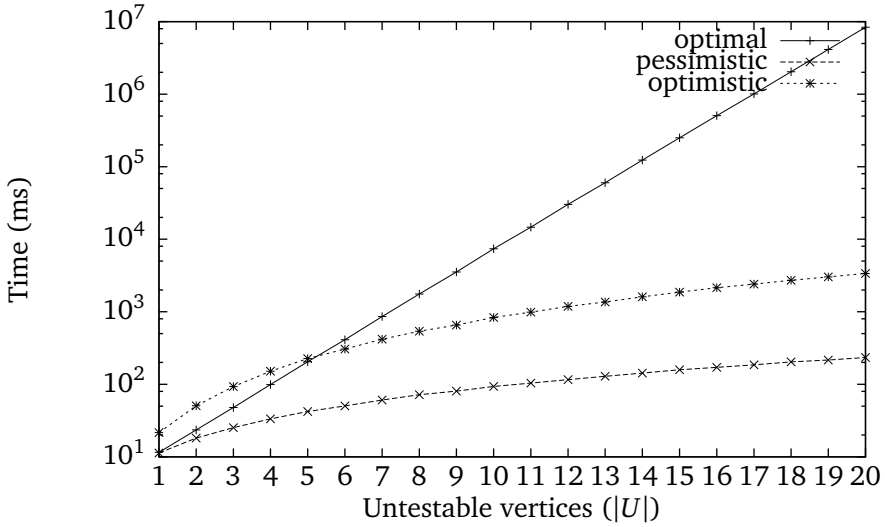


Figure 7.6: Time complexity of the optimization

its application to real systems with a large number of vertices is not practical, especially if we want the preparation plan to be available in reasonable time. An approximate solution with a smaller computational complexity is needed.

In this section we present an approach to generate an approximate preparation plan by using a greedy selection heuristic method. according to Algorithm 7, where CIG represents the interaction graph of the system, U is the set of untestable vertices, and $h(v)$ a heuristic function to be used. For each pass of the loop, the algorithm selects the vertex in U with the highest rank according to $h(v)$, and removes it from the set of untestable vertices. The rank is updated in each pass. This algorithm constructs its solution incrementally. The preparation plan at each step contains the plan of the previous steps.

Algorithm 7 Incremental Approximate Planning

```
function PreparationPlan(CIG, U, h(v))
    Sol ← ∅                                  ▷ List to hold the solution
    while U ≠ ∅ do
        v ← FindMax(U, h(v))
        Append(Sol, v)
        Remove(U, v)
    return Sol
```

The method relies on heuristics that benefit from partial knowledge about the structure of the solution space of the problem.

Heuristics

First, we consider a heuristic that takes a pessimistic approach: it ranks higher the vertices that provide the highest gain on testability in the next step. The count is divided by the cost to penalize expensive nodes. The heuristic is defined as

$$h_{pessimistic}(v_i) = \frac{1}{c_i}(RTM_{v_i} - RTM) \tag{7.1}$$

where RTM_{v_i} is the value of RTM after the costs of vertex v_i have been covered. Given the pessimistic nature of this heuristic, we expect this heuristic rank to perform well for low budgets, and poorly for higher ones.

Our second heuristic takes an optimistic approach. It ranks higher the vertices that appear in the highest number of P sets, i.e., the vertices that will fix the most uncoverable vertices assuming they only depend on the vertex being ranked. This value is also divided by the cost to penalize expensive nodes over cheaper ones. The heuristic is defined as

$$h_{optimistic}(v_i) = \frac{1}{c_i}|\{v_j \mid v_i \in S_{v_j}\}| \tag{7.2}$$

By ignoring the fact that an uncoverable vertex may be caused by more than one untestable vertex, and that the vertex may not be reachable through a testable path, this second heuristic will take very optimistic decisions on the first passes which affect the quality of results for proportionally low budgets, and yields a better performance for higher budgets.

The optimistic heuristic will skip many low-cost solutions (its curve being much lower than the optimum), while the pessimistic heuristic is more precise for low costs but completely misses good solutions with higher budgets. A simple solution to address these shortcoming is to combine both heuristic rankings by taking the best results of both.

Computational Complexity and Error

The time complexity of the preparation plan computation depends on the time complexity of the heuristic function. As in each pass there is one less vertex in U, the H function is evaluated $|U|, |U|-1, \ldots, 1$ times while searching for the maximum. In total, it is evaluated $\frac{|U|^2}{2}$ times.

Both heuristic functions perform a sum that depends on the total number of vertices in the system, the complexity of the PreparationPlan function is $O(|V| \cdot |U|^2)$, and therefore polynomial.

Although a polynomial complexity is much more appealing than the $O(2^{|U|})$ complexity of the exhaustive search, the approximation error has to be taken into account.

A number of experiments were conducted to evaluate the approximation error of our heuristic method. The plot in Figure 7.7 shows the evolution of the relative average approximation error of RTM for each of our heuristics, as a function of the number of untestable operations $|U|$. The optimal solution function is obtained by exhaustive search.

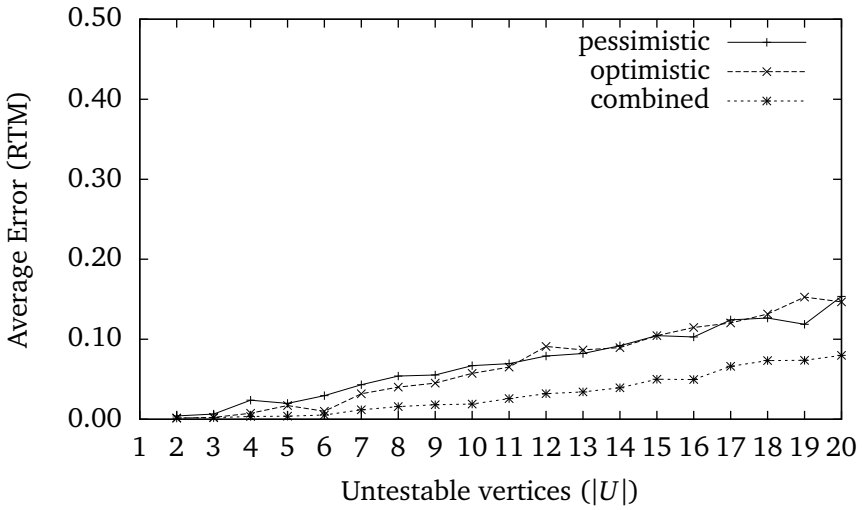


Figure 7.7: Performance of the approximate algorithms compared to the optimal

The average error incurred by our heuristics is very low considering the processing time required for their calculation. Combining the rankings created by both the long and short-term heuristics, by choosing the maximum of either solution, yields the best of both methods while maintaining the low computational complexity. This can be seen in the *combined* error plot in Figure 7.7.

7.4 Related Work

A number of research approaches have addressed testability from different angles. However, runtime testability has not been considered in depth so far. To the best of our knowledge, our work is the first to define a cost-driven method (RiTMO) to drive testability improvement efforts in terms of a testability/cost optimization problem.

Two articles form the base for our approach to runtime testability, presented in [González et al., 2009a], and complemented with the definition of RiTMO in this chapter. In [Brenner et al., 2007b], Brenner et al. introduce the concepts of test sensitivity and isolation. However no mention to nor relation with the concept of runtime testability are done. In [Suliman et al., 2006], Suliman et al. discuss several test execution and sensitivity scenarios, for which different isolation strategies are advised. The factors that affect runtime testability cross-cut those in

Binder's Testability [Binder, 1994] model, as well as those in Gao's component-based adaptation [Gao and Shih, 2005].

Testability improvement efforts have focused on regression testing a well, selecting and prioritizing the test cases more prone to uncover faults (i.e., those which contribute more test coverage) with the least cost [Elbaum et al., 2002, Li et al., 2007, Smith and Kapfhammer, 2009]. Our approach to testability improvement is also cost-based. However, their approaches are focused on test cases, whereas our approach is focused towards the system.

Other prior testability-related approaches have focused on modeling statistically which characteristics of the source code of the system are more prone to uncovering faults [Freedman, 1991, Voas et al., 1991] for amplifying reliability model information [Bertolino and Strigini, 1995, Hamlet and Voas, 1993]. Jungmayr [Jungmayr, 2002] proposes a method based on a measurement of testability from the point of view the static structure of the system to assess the maintainability of the system. Our approach is similar in that runtime testability is influenced by the structure of the system under consideration.

7.5 Summary

In this chapter we have introduced an approach to the improvement of the system's runtime testability in terms of a RTM/cost optimization problem.

RiTMO enables integration engineers to identify critical situations of bad system runtime testability and to compute and execute an action plan to improve it. We have provided a low-cost approximation algorithm which computes near-optimal preparation plans, significantly reducing the computation time.

The next step in the development of our work is the integration of RTM and RiTMO into a CASE tool, enabling system engineers to receive timely feedback about the system they are designing.

8

A Diagnostic View on Runtime Testability

In order to perform runtime tests, an additional expense in the test preparation phase is required, both in software development and in hardware. Preparation cost prioritization methods have been based on runtime testability (i.e, coverage) and do not consider whether a good runtime testability is sufficient for a good runtime diagnosis quality in case faults are detected, and whether this diagnosis will be obtained efficiently (i.e., with a low number of test cases).

In this chapter we show (1) the direct relationship between testability and diagnosis quality, that (2) these two properties do not guarantee an efficient diagnosis, and (3) a measurement that ensures better prediction of efficiency.

8.1 Efficiency: Taking Testing Cost into Account **129**
8.2 Experiments **131**
8.3 Related Work **135**
8.4 Summary **136**

In practice many parts of a system cannot be runtime tested, because the tests would affect the system in critical ways that are difficult to control or impossible to recover from. For instance, firing a missile while testing part of a combat system. In order to test those parts, an additional expense in the test *preparation* phase is required, both in software development, (e.g., test wrappers, testable components, simulators) and in hardware (e.g., an extra processor, more memory, higher capacity data links). Therefore, not only the execution costs of testing, but also the preparation costs, need to be optimized, i.e., prioritized.

We can represent test cases and their coverage in a matrix. Columns correspond to components, and rows to tests. This is shown in Figure 8.1. If test case t_i covers component c_j, this is represented with element $a_{ij} = 1$ in the test matrix. Otherwise, $a_{ij} = 0$. The execution costs are represented by the C_t column, whereas the preparation costs associated to components are represented by the C_p row.

There exists extensive literature on prioritizing the execution costs of a test suite. Test execution prioritization techniques operate on the vertical dimension of

	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	C_t	o_i
t_1	1	0	1	0	0	1	1	0	10	1
t_2	0	1	0	1	0	0	0	0	10	0
t_3	0	0	1	0	1	0	1	0	10	0
t_4	0	1	0	0	0	0	0	1	10	0
t_5	1	0	0	0	1	0	0	0	10	1
t_6	0	0	0	0	0	1	1	0	10	0
t_7	0	1	0	1	0	1	0	1	10	0
t_8	0	1	1	0	1	0	1	1	10	0
t_9	1	0	1	0	0	1	0	1	10	1
C_p	0	0	0	0	0	0	10	10		

Table 8.1: Example coverage matrix, execution, and preparation costs

the matrix, selecting test cases based on their execution cost and their utility. This utility can be a measurement of either fault detection [Elbaum et al., 2002, Jiang et al., 2009a, Li et al., 2007, Rothermel et al., 2001, Smith and Kapfhammer, 2009, Wong et al., 1997] or fault localization [Gonzalez-Sanchez et al., 2011e] potential. On the other hand, preparation cost optimization methods operate on the horizontal dimension, and select components to prepare based on runtime testability measurements [Gonzalez-Sanchez et al., 2009, Gonzalez-Sanchez et al., 2010b].

However, preparation cost prioritization methods fall short in two respects. First, one must consider that testing is very often paired to a diagnosis phase. Since runtime testability is optimized for coverage, a question left unanswered is whether a good runtime testability is sufficient for a good runtime diagnosis quality. Second, different preparations will enable the execution of different test cases, which means that, in practice, test preparation prioritization should take into account the test execution cost of the tests it is 'enabling'. As important as the final coverage or diagnosis quality of the test cases, is the rate at which such coverage or diagnosis quality evolves.

In this chapter we address these issues, showing (1) the direct relationship between testability and diagnosability, (2) that these two measurements do not guarantee an efficient diagnosis, and (3) a third measurement that ensures better prediction of efficiency.

The chapter is organized as follows. Section 8.1 introduces the concept of diagnosis efficiency and measurement. Section 8.2 presents the systems used in our evaluation and our empirical results. Section 8.3 positions this work with respect to related work. Finally, Section 8.4 wraps up the chapter and presents direction for further research.

8.1 Efficiency: Taking Testing Cost into Account

Each test that is executed provides new diagnostic information. As important as the final diagnostic effort, C_d, is the rate at which diagnostic effort decreases per unit of test cost. This is known as efficiency, S, and is calculated as

$$S = \sum_{i=1}^{N} C_d(i) - C_d(N) \tag{8.1}$$

where $C_d(N)$ is the diagnostic effort once all the test cases have been applied.

So far, the prioritization of test preparation costs has been driven exclusively by the trade-off between preparation cost and runtime testability/diagnosability. However, when considering fault localization, one should take into account not only the *final* residual effort of the diagnosis C_d, but also the testing cost involved. In other words, one should prioritize preparation operations that not only 'enable' tests that provide a good RTM and final C_d, but also that have a low S value.

The following example matrix demonstrates how just RTM is not sufficient for this purpose.

$$A = \left[\begin{array}{cccccccc} c_1 & c_2 & c_3 & c_4 & c_5 & c_6 & c_7 & c_8 \\ \hline 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \end{array}\right]$$

Components c_7 and c_8 are not testable. If we prepare either of them, $RTM = 7$. However, if we take into account diagnostic efficiency, S, that each of the preparations yield, there is a clear difference. Figure 8.1 shows a simulation of how preparing c_7 will produce a much slower decay of C_d than if we had prepared c_8, even though the RTM values are identical for both options.

This example highlights the necessity of a third measurement that can predict the efficiency of the tests enabled by the preparations.

Runtime Diagnostic Efficiency

To estimate runtime diagnostic efficiency, RSM, we will use a simulation based on the evolution of the diagnostic effort when combined with a greedy optimization algorithm based on information gain [Gonzalez-Sanchez et al., 2011e]. The algo-

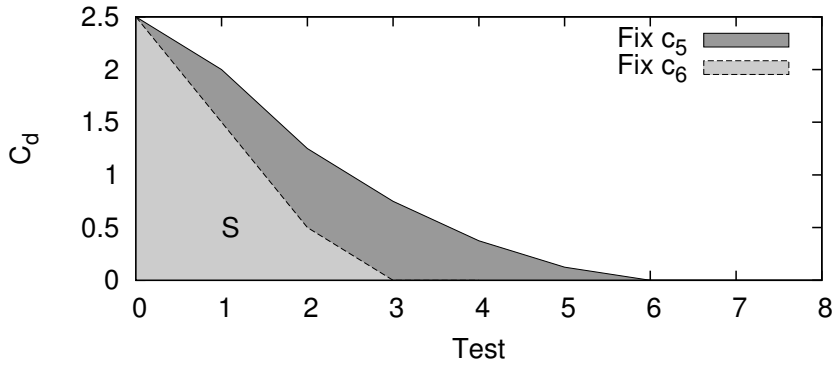


Figure 8.1: Difference in S of two preparations with identical RTM and RDM

rithm uses the tests that have $C_p(i_i) = 0$ to calculate the value of RSM, as was done with RTM.

Since simulating a diagnosis process can be very costly and requires many runs, we will use an approximation based on diagnosability. Every test matrix A decomposes a system into a partition of components $G = g_1, g_2, \ldots, g_L$, Each subset g_i contains all components with identical columns in A. Each of those subsets is called an *ambiguity group*.

The *diagnosability* of a test matrix is the ability of distinguishing between components uniquely, by decomposing the system into the smallest possible ambiguity groups. For example the matrix

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}$$

has a lower diagnostic ability than

$$A' = \begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

since the later generates a finer partition of ambiguity groups $\{\{1,3,6,9\},\{2\},\{5,8\},\{4\},\{7,10\}\}$ whereas the first one only $\{\{1,3,6,9\},\{2,5,8\},\{4,7,10\}\}$.

The probability of group g_i occurring is $\Pr(g_i) = \frac{|g_i|}{M}$, and the residual diagnostic effort in such group corresponds to $C_d = \frac{|g_i|-1}{2}$. The formula for the diagnosability is then

$$\mathrm{G}(A) = \frac{1}{M} \sum_{i=1}^{L} \frac{|g_i|}{M} \cdot \frac{|g_i| - 1}{2} \tag{8.2}$$

and represents the expected (i.e., average) value of the diagnostic effort once the test in A have been applied.

The first example matrix in Table 8.1 has a $G(A) = \frac{1}{10}(\frac{4}{10} \cdot \frac{3}{2} + \frac{3}{10} \cdot \frac{2}{2} + \frac{3}{10} \cdot \frac{2}{2}) = 0.12$. On the other hand the second example matrix, which can distinguish more components uniquely, has a $G(A') = \frac{1}{10}(\frac{4}{10} \cdot \frac{3}{2} + 0 + \frac{2}{10} \cdot \frac{1}{2} + 0 + \frac{2}{10} \cdot \frac{1}{2}) = 0.08$.

Algorithm

This greedy simulation algorithm provides a good estimation of the evolution of C_d when using the information gain algorithm [Gonzalez-Sanchez et al., 2011e] to prioritize test cases, at a lower cost without having to resort to simulations of diagnoses.

Algorithm 8 RSM Estimation

$A \leftarrow \emptyset$
$T \leftarrow \{t_i : C_p(t_i) = 0\}$
$RSM \leftarrow \frac{M-1}{2}$
for $l \leftarrow 1, |T|$ **do**
  $i = \arg\min_{t_i \in T} \left(G(A||t_i)\right)$
  $A \leftarrow A||t_i$
  $RSM \leftarrow RSM + G(A)$
return RSM

Algorithm 8 computes the evolution of $G(A)$ in a greedy way. At each step, it selects the test that minimizes the ambiguity of the matrix and appends the test to the matrix (operator $||$) . Initially the diagnosis is totally ambiguous ($C_d = \frac{M-1}{2}$). At every iteration, a test t_i is selected, such that, when concatenated to the matrix, it minimizes its ambiguity.

Obtaining the partition G has $O(M)$ time complexity when done incrementally. This has to be repeated N times per choice, for N choices. The complexity of Algorithm 8 is $O(MN^2)$, similar to IG but with a much lower constant, and it must be done once instead of being repeated a large number of times.

8.2 Experiments

Our evaluation study aims to answer the main research question concerning the utility of each of our metrics when considering the planning of preparation costs for runtime testing.

Systems

Example System I: WifiLounge

We diagnose the runtime testability and diagnosability of a wireless hotspot at an airport lounge [Bures,]. The component architecture of the system is depicted

in Figure 8.2. Clients authenticate themselves as either business class passengers, loyalty program members, or prepaid service clients.

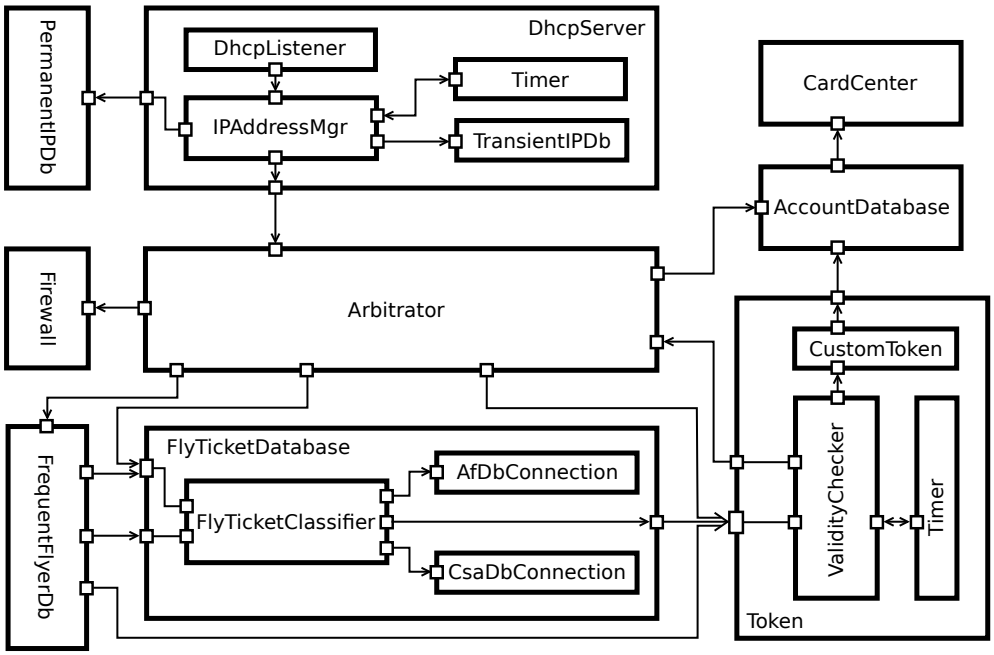


Figure 8.2: Wifi Lounge Component Architecture

When a computer connects to the network, the `DhcpListener` component generates an event indicating the assigned IP address. All communications are blocked by the firewall until the authentication is validated. Passengers of business class are authenticated in the ticket databases of a number of airlines. Passengers from a frequent flier program are authenticated against the program's database, and the ticket databases to check that they are actually entitled to free access. Passengers using the prepaid method must create an account in the system, linked to a credit card that is used for the payments. Once the authentication has succeeded, the port block in the firewall is disabled so that the client can use the connection. The session ends when the user disconnects, or the authentication token becomes invalid. If the user is using a prepaid account, its remaining prepaid time will be updated.

Siemens & SIR Programs

Even though these programs and their test suites were not designed with runtime testing in mind, to strengthen the validity of our study, we use the test coverage matrices of a set of seven test programs known as the Siemens set [Hutchins et al., 1994], and five programs from the Software Infrastructure Repository (SIR) [Do et al., 2005]. Table 8.2 provides more information about the programs (for more detailed information refer to [Hutchins et al., 1994] and [Do et al., 2005]).

Program	LOC	Tests	Description
print_tokens	563	4130	Lexical Analyzer
print_tokens2	509	4115	Lexical Analyzer
replace	563	5542	Pattern Matcher
schedule	412	2650	Priority Scheduler
schedule2	307	2710	Priority Scheduler
tcas	173	1608	Aircraft Control
tot_info	406	1052	Information Measure
sed	7125	370	Stream Editor
space	9126	150	ADL Compiler
grep	13287	809	String Matching
gzip	7933	210	Data Compression
flex	14194	107	Lexer Generator

Table 8.2: Set of programs and versions used in the experiments

The coverage matrix A of each program is obtained by instrumenting each of the programs with Zoltar [Janssen et al., 2009] to obtain the statements covered by each test case. Type and variable declarations and other static code, which are not instrumented by Zoltar, are excluded from diagnostic rankings and effort calculations.

Relationship between RTM and C_d

Before considering test execution costs, the first question we want to answer is: what is the relationship between RTM and diagnostic effort, C_d, if any? To this end we randomly chose sets of 1 to 15 untestable components in each of the systems, and recorded the value of RTM, and C_d for a set of simulated faults. These faults were obtained by simulating random faults in one of the program components producing a failure for every test case that covers the faulty component.

Figure 8.3 shows the correlation between RTM and C_d. It can be clearly seen that there is a correlation between them, with a linear shape, or very slightly negative-exponential towards high values of RTM.

The reason for this direct relationship is that a testable (i.e., coverable) system, although not enough to guarantee a good C_d, is a prerequisite. By diagnosability, a good C_d is obtained by covering components in a way that makes them be in different ambiguity groups. The test matrices used in our study correspond to fairly diagnosable systems. Therefore, if a component is testable, it will be very likely also diagnosable, causing this direct relationship between RTM and C_d.

The main conclusion that can be drawn from this relationship is that an investment in runtime testability will cause a direct improvement on C_d. Since the

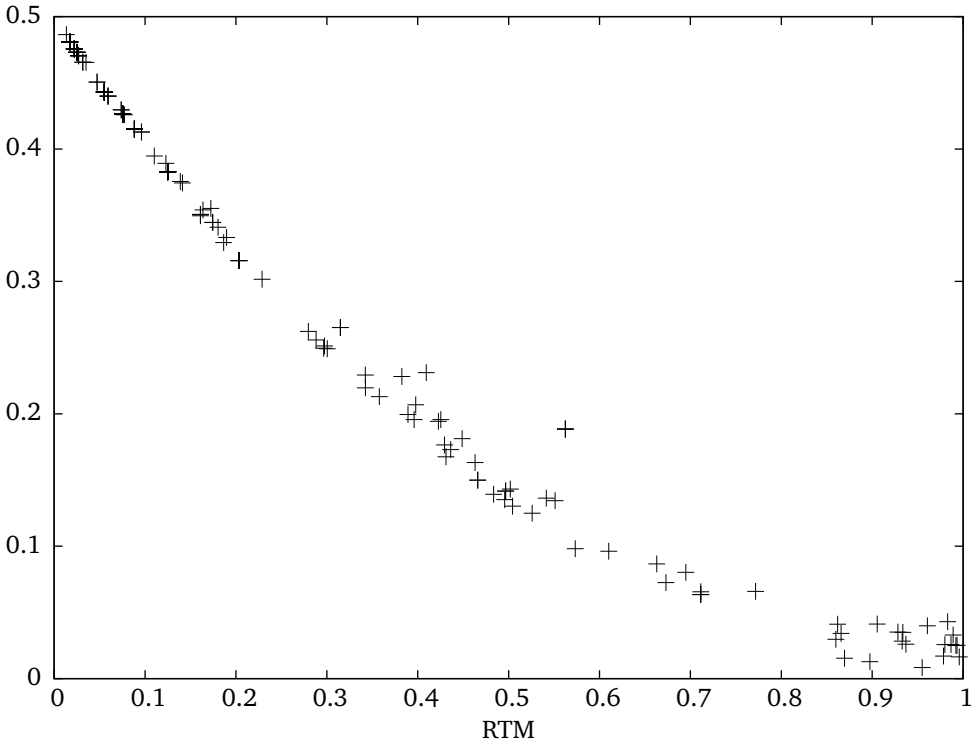


Figure 8.3: Correlation between RTM and C_d

relationship is very linear, we can be almost certain that our investment in testability has an equivalent return on the diagnosability of the system. The exception to this rule are systems with a large number of large ambiguity groups.

Relationship with Efficiency

A second question we would like to answer is the relationship with diagnostic efficiency. We have already presented an example in Section 8.1 that illustrates the fact that good runtime testability and diagnosability are no guarantee to an efficient diagnosis. This is clearly proven in the first of the two plots of Figure 8.4 that show that there is no correlation whatsoever between RTM, and diagnostic efficiency, S.

On the other hand, the RSM measurement, obtained by Algorithm 6 provides a much better correlation, and indicator of the efficiency that can be achieved. Moreover, RSM being based on $G(A)$ takes both into account the improvement rate of C_d, and the final C_d, making it a much more complete measurement.

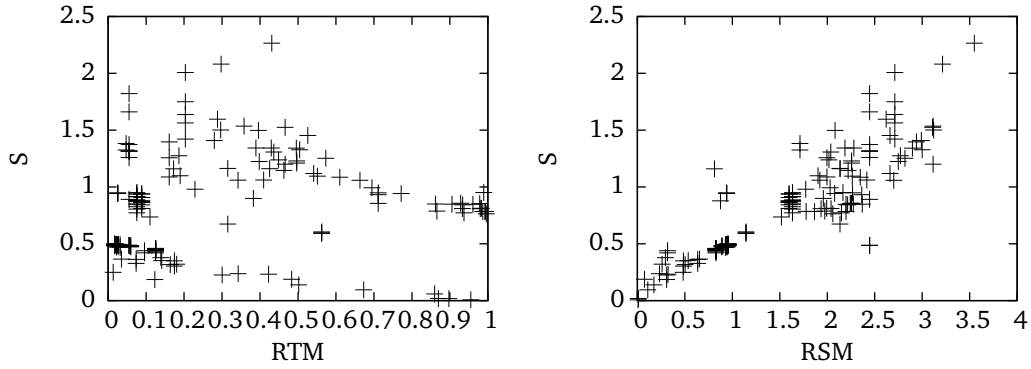


Figure 8.4: Correlation of RTM, and RSM with Efficiency

Threats to Validity

The main threat to the validity of our experiments is the representativeness of the used systems. Adding a broader variety of systems where runtime testing is performed will enhance the validity of our results both because of the large sample of architectures but also because it would include more variety in the sources of test sensitivity.

A second threat stems from the fact that we used simulated faults in our validation. This could be improved by using mutation faults, since there is greater consensus that mutation faults are similar in nature to faults found during unit testing.

8.3 Related Work

Testability and testing effort were originally related to the probability of a fault causing an error, this error propagating to the output [Hamlet and Voas, 1993, Voas et al., 1991], and the error being actually detected by the test oracle [Bertolino and Strigini, 1995].

However, testability is affected by many other factors and its measurement has been studied extensively from a qualitative [Binder, 1994, Gao and Shih, 2005] and quantitative [Jungmayr, 2002, Freedman, 1991] way. Runtime testability and RTM were first introduced in [González et al., 2009a] and refined in [Gonzalez-Sanchez et al., 2010b]. A method for the improvement of runtime testability of component based systems was described in [Gonzalez-Sanchez et al., 2009]. In contrast with the previous work, in this chapter we focus in the diagnostic perspective of runtime testing to optimize the runtime testing preparation costs.

Cost prioritization during testing has been extensively studied, from the point of view of test execution costs. Test prioritization techniques usually operate by selecting test cases that provided the highest fault detection potential[Elbaum

et al., 2002, Jiang et al., 2009a, Li et al., 2007, Rothermel et al., 2001, Smith and Kapfhammer, 2009, Wong et al., 1997]. Lately, test prioritization which selects tests by their potential to locate faults has been proposed [Gonzalez-Sanchez et al., 2011e] potential. None of these works take into account preparation cost.

8.4 Summary

Existing literature on runtime testability has considered only the implications of testability in the prioritization of preparation costs for runtime testing. In this chapter we have studied runtime testing from the point of view of diagnosis. The main conclusions are (1) that can be drawn are that runtime testability and diagnosability are related strongly to each other, (2) that RTM and RDM have no relationship with the efficiency of the diagnosis. We have introduced a third measurement, RSM, based on a greedy algorithm that can predict the efficiency of runtime diagnosis with a good accuracy.

9 Conclusions

Runtime testing and diagnosis are a developer's main means of performing the quality assurance process of dynamically evolving and high-availability systems, such as Systems of Systems or Service Oriented Architectures. As part of this quality assurance process, we have identified the three most relevant characteristics for runtime testing: (1) test preparation, (2) fault detection, (3) fault localization; and their associated costs: (1) preparation cost, (2) testing cost, and (3) inspection cost.

At the core of this thesis, we have investigated the relationship and trade-off between fault detection and fault localization. We started from the prior knowledge that boosting the fault detection capability of a test suite (thus, lowering testing cost) the fault localization capability becomes severely affected. Our approach is a departure from the commonplace fault detection-centric paradigm towards a fault localization-centric one, where during testing test cases are not selected by their fault detection potential, but for their potential to improve the current diagnosis. In particular, we have investigated an on-line test selection method based on the information gain heuristic. Although this method provides theoretically the most optimal solution, we showed that in practice the method's performance is not always good due to estimation errors in the input parameters. Motivated by this fact, we investigated two alternative approaches. First, we investigated a fall-back mechanism that detects and prevents errors in parameter estimations to adversely affect the information gain heuristic by switching to a safe mode. Second, we investigate a simpler, off-line test selection heuristic that achieves comparable performance and does not suffer from these problems. Besides its comparable performance, the off-line algorithm does not incur in additional overhead during testing. Furthermore, its computational complexity is much lower than the on-line algorithm.

Furthermore, we explored the novel concept of runtime testability, with an emphasis on the cost of the preparation actions needed to achieve testability, and the trade-off with the fault detection capabilities of runtime tests. As a first step, we proposed and validated a model and a metric to quantify and predict the value of runtime testability. Being able to measure runtime testability is the first step towards its optimization. We provided an integrated method to obtain this measurement from system specifications, and together with the preparation costs, ob-

tain an optimal sequence of preparations. Our work is not restricted only to the trade-off between test preparation and fault detection. We also studied the relationship between test preparation and fault localization, and proposed a metric to effectively prioritize preparation costs in terms of the fault localization capability of the system's test suite.

9.1 Summary of Contributions

The main contributions of this thesis are:

- From our study on diagnostic test prioritization, we conclude that, except in some cases, current test prioritization algorithms are not designed to optimize fault localization. We show how the concept of information gain is the key to an optimal trade-off between fault detection and fault localization.

- We introduce a novel, low-cost, on-line diagnostic test prioritization algorithm, named Sequoia. The main novelty of Sequoia is the fact that it can perform efficient test sequencing for diagnosis in systems with multiple faults and with test cases that can produce false negatives, by using a first order expansion of the set of minimal diagnostic candidates. Previous approaches only work in systems where only one fault is present, or in a false negative-free context. Our experiments with synthetic matrices and simulations with real program matrices show that Sequoia outperforms every other technique in terms of diagnostic effectiveness, with effectiveness improvements of up to 80%, while hardly sacrificing any fault detection performance. However, when applied in practice, we show that due to large errors in Sequoia's input parameters, its performance can be severely affected.

- We introduce a second low-cost diagnostic test prioritization algorithm, under the name Raptor. Unlike Sequoia, Raptor performs off-line prioritization and does not rely so heavily on the accuracy of its input parameters to work. Raptor exploits the fact that every spectrum-based fault localization algorithm relies on quickly differentiating the execution signatures of components and comparing them with the observed failures. By enhancing the rate at which component execution signatures differentiate, Raptor is able to obtain comparable performance to Sequoia, even outperforming it in many cases.

Furthermore, this thesis also makes the following contributions. First, as Sequoia, and Bayesian diagnosis in general, require the estimation of the false negative rate of tests before the diagnosis can commence, we research the possibility of using a mutation study to obtain such information. Our results show that this

information can be used to obtain a good diagnosis, even though the assumption of failure independence in the multiple-fault case does not hold in many cases.

Second, the fact that RAPTOR outperforms SEQUOIA in many cases, shows that the optimal solution in theory is not in practice. RAPTOR not only produces good results, but is also a simpler algorithm, requires less input parameters, and works off-line (i.e., does not introduce an overhead in the test process). Whereas from a scientific point of view SEQUOIA's value is out of question, from an engineering point of view, RAPTOR is a much more attractive alternative.

Third, we obtained an analytical performance model that shows how the characteristics of test suites have a great influence in diagnostic effectiveness. Our model highlights that current practice, where test suites are designed for very sparse coverage per test, is not optimized for fault detection nor fault localization. Fortunately, thanks to our model we know that there is ample room for improvement of test suites. This opens the door, for example, to the definition of new test generation algorithms that optimize test suites based on their potential for an effective fault localization.

Fourth, we presented runtime testability and the Runtime Testability Metric (RTM), that can be used to find the optimal investment in preparation costs to enhance the fault detection capability of our tests during runtime testing. RTM is based on a system dependency model and provides information independent of the nature of the test suite. The simplicity of RTM's model allows easy computation in the dynamic environment of a SoS.

9.2 Recommendations for Future Work

A multitude of interesting open issues are worth investigating. In the following we suggest several recommendations for future work:

1. Derivation of FNR Through Static Analysis

We have shown how the knowledge of FNR information can reduce both testing cost and diagnostic cost. However, these savings do not take into account that obtaining the FNR, although automated, is a very expensive process. Future work should investigate alternative FNR estimation techniques, e.g. by applying static analysis techniques that try to obtain results similar to the probabilistic approach needed for PIE.

2. Fall-back Strategies for Noisy FNR

Our work with diagnostic prioritization has highlighted the fact that the information gain heuristic, although optimal in theory, it can be affected by erroneous estimations of FNR in practice. This calls for

the definition of strategies to cope with noisy FNR, for example by detecting potentially dangerous situations like high variance in the FNR estimations, and falling back to a more robust test prioritization algorithm, or simply, to a random test ordering.

3. Staccato Optimization

The SEQUOIA algorithm relies in the STACCATO algorithm to generate a new diagnosis D. The results in our evaluation of diagnostic prioritization has shown that STACCATO's performance is very good. There are, however, a number of possible improvements. First, at each iteration of the SEQUOIA algorithm, STACCATO has to compute D from scratch, creating an important performance bottleneck. It would be very beneficial for SEQUOIA to develop an incremental variant of STACCATO. Second, currently the expansion of D to the first subsumption is done in no particular order. A 1st order expansion policy that expands first the candidate with higher probability would help to improve the results of SEQUOIA.

4. Effectiveness Modeling

We have developed analytical models to explain the differences in diagnostic effectiveness between programs. Our diagnostic effectiveness model provides very reliable predictions for uniform test matrices. However, there is ample room for improvement in the case of non-uniform matrices. In this case, a model that considers the probability density of test coverage can be useful. A second issue left for future work is the extension to test matrices with duplicated columns (components with identical signatures), where residual diagnostic effort cannot be totally eliminated.

5. Generation of New Test Suites

The analytical model of diagnostic effectiveness was aimed at understanding the differences in performance between different programs. Our results have shown what characteristics of test suites are the most desirable for diagnostic prioritization, and opens the door to the development of test suite generation techniques that use our model as test criterion, e.g., as fitness function in a search algorithm.

7. Improvement of RTM

The Component Interaction Graph (CIG), the dependency model used to predict which operations can be exercised during runtime testing, has a number of shortcomings that affect the prediction power of RTM.

The definition of RTM assumes independence between paths, which is a source of imprecision in the presence of data dependencies (not modeled). This can be solved by adding explicit data dependencies to the CIG. A second assumption that when the test interaction reaches a node, it will be propagated to all its dependencies, which is not always true. For example, propagation probabilities to each node can be added to each of the dependencies in the CIG.

Bibliography

[Abreu and van Gemund, 2009] Abreu, R. and van Gemund, A. (2009). A low-cost approximate minimal hitting set algorithm and its application to model-based diagnosis.

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

[Abreu et al., 2009b] Abreu, R., Zoeteweij, P., Golsteijn, R., and van Gemund, A. J. C. (2009b). A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, 82(11):1780–1792.

[Abreu et al., 2007] Abreu, R., Zoeteweij, P., and van Gemund, A. J. C. (2007). On the accuracy of spectrum-based fault localization. In *TAICPART-MUTATION '07: Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, pages 89–98, Washington, DC, USA. IEEE Computer Society.

[Abreu et al., 2008] Abreu, R., Zoeteweij, P., and van Gemund, A. J. C. (2008). An observation-based model for fault localization. In *WODA '08: Proceedings of the 2008 international workshop on dynamic analysis*, pages 64–70, New York, NY, USA. ACM.

[Abreu et al., 2009c] Abreu, R., Zoeteweij, P., and Van Gemund, A. J. C. (2009c). A new bayesian approach to multiple intermittent fault diagnosis. In *Proceedings of the 21st international jont conference on Artifical intelligence*, pages 653–658, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

[Andrews et al., 2005] Andrews, J. H., Briand, L. C., and Labiche, Y. (2005). Is mutation an appropriate tool for testing experiments? In *ICSE '05: Proceedings*

of the 27th international conference on Software engineering, pages 402–411, New York, NY, USA. ACM.

[Avizienis et al., 2004] Avizienis, A., Laprie, J.-C., Randell, B., and Landwehr, C. (2004). Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33.

[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 *ISSTA '08: Proceedings of the 2008 international symposium on Software testing and analysis*, pages 189–200, New York, NY, USA. ACM.

[Baudry et al., 2006] Baudry, B., Fleurey, F., and Le Traon, Y. (2006). Improving test suites for efficient fault localization. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 82–91, New York, NY, USA. ACM.

[Bertolino and Strigini, 1995] Bertolino, A. and Strigini, L. (1995). Using testability measures for dependability assessment. In *ICSE '95: Proceedings of the 17th international conference on Software engineering*, pages 61–70, New York, NY, USA. ACM.

[Binder, 1994] Binder, R. V. (1994). Design for testability in object-oriented systems. *Communications of the ACM*, 37(9):87–101.

[Brenner et al., 2007a] Brenner, D., Atkinson, C., Hummel, O., and Stoll, D. (2007a). Strategies for the run-time testing of third party web services. In *SOCA '07: Proceedings of the IEEE International Conference on Service-Oriented Computing and Applications*, pages 114–121, Washington, DC, USA. IEEE Computer Society.

[Brenner et al., 2007b] Brenner, D., Atkinson, C., Malaka, R., Merdes, M., Paech, B., and Suliman, D. (2007b). Reducing verification effort in component-based software engineering through built-in testing. *Information Systems Frontiers*, 9(2-3):151–162.

[Brodie et al., 2002] Brodie, M., Rish, I., and Ma, S. (2002). Intelligent probing: A cost-effective approach to fault diagnosis in computer networks. *IBM Systems Journal*, 41(3):372–385.

[Bruntink and van Deursen, 2006] Bruntink, M. and van Deursen, A. (2006). An empirical study into class testability. *J. Syst. Softw.*, 79(9):1219–1232.

[Bures,] Bures, T. Fractal bpc demo.

[David, 1970] David, H. A. (1970). *Order Statistics*. John Wiley & Sons.

[De Kleer, 2009] De Kleer, J. (2009). Diagnosing multiple persistent and intermittent faults. In *IJCAI'09: Proceedings of the 21st international jont conference on Artifical intelligence*, pages 733–738, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

[de Kleer et al., 1992] de Kleer, J., Mackworth, A. K., and Reiter, R. (1992). Characterizing diagnoses and systems. *Artificial Intelligence*, 56:197–222.

[Do et al., 2005] Do, H., Elbaum, S., and Rothermel, G. (2005). Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Softw. Engg.*, 10(4):405–435.

[Dumitras et al., 2009] Dumitras, T., Eliassen, F., Geihs, K., Muccini, H., Polini, A., and Ungerer, T. (2009). Testing run-time evolving systems. Number 09201 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany. Schloss Dagstuhl.

[E. U. Commission, Maritime Affairs, 2007] E. U. Commission, Maritime Affairs (2007). An integrated maritime policy for the european union.

[Elbaum et al., 2001] Elbaum, S., Malishevsky, A., and Rothermel, G. (2001). Incorporating varying test costs and fault severities into test case prioritization. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 329–338, Washington, DC, USA. IEEE Computer Society.

[Elbaum et al., 2002] Elbaum, S., Malishevsky, A. G., and Rothermel, G. (2002). Test case prioritization: A family of empirical studies. *IEEE Transactions on Software Engineering*, 28(2):159–182.

[Elbaum et al., 2004] Elbaum, S., Rothermel, G., Kanduri, S., and Malishevsky, A. G. (2004). Selecting a cost-effective test case prioritization technique. *Software Quality Control*, 12(3):185–210.

[Embedded Systems Institute, 2007] Embedded Systems Institute (2007). The poseidon project.

[Freedman, 1991] Freedman, R. S. (1991). Testability of software components. *IEEE Transactions on Software Engineering*, 17(6):553–564.

[Gao and Shih, 2005] Gao, J. and Shih, M.-C. (2005). A component testability model for verification and measurement. In *COMPSAC '05: Proceedings of the 29th Annual International Computer Software and Applications Conference*, volume 2, pages 211–218, Washington, DC, USA. IEEE Computer Society.

[Garlan and Shaw, 1994] Garlan, D. and Shaw, M. (1994). An introduction to software architecture. Technical report, Pittsburgh, PA, USA.

[González et al., 2008a] González, A., Piel, É., and Gross, H.-G. (2008a). Architecture support for runtime integration and verification of component-based systems of systems. In *1st International Workshop on Automated Engineering of Autonomous and run-time evolving Systems (ARAMIS 2008)*, pages 41–48, L'Aquila, Italy. IEEE Computer Society.

[González et al., 2009a] González, A., Piel, E., and Gross, H.-G. (2009a). A model for the measurement of the runtime testability of component-based systems. In *Software Testing Verification and Validation Workshop, IEEE International Conference on*, pages 19–28, Denver, CO, USA. IEEE Computer Society.

[González et al., 2008b] González, A., Piel, É., Gross, H.-G., and Glandrup, M. (2008b). Testing challenges of maritime safety and security systems-of-systems. In *Testing: Academic and Industry Conference - Practice And Research Techniques*, pages 35–39, Windsor, United Kingdom. IEEE Computer Society.

[González et al., 2009b] González, A., Piel, E., Gross, H.-G., and van Gemund, A. (2009b). Runtime testability in dynamic highly available component-based systems. Technical Report TUD-SERG-2009-009, Delft University of Technolgy, Software Engineering Research Group.

[Gonzalez-Sanchez et al., 2010a] Gonzalez-Sanchez, A., Abreu, R., Gross, H.-G., and van Gemund, A. J. (2010a). Spectrum-based sequential diagnosis. In Poll, S., de Kleer, J., Roychoudhury, I., and Daigle, M., editors, *21st International Workshop on the Principles of Diagnosis (DX'10)*, pages 55–62. PHM Society.

[Gonzalez-Sanchez et al., 2011a] Gonzalez-Sanchez, A., Abreu, R., Gross, H.-G., and van Gemund, A. J. (2011a). An empirical study on the usage of testability information to fault localization in software. In *Proceedings of the 26th Symposium On Applied Computing (SAC'11)*, pages 1398–1403.

[Gonzalez-Sanchez et al., 2011b] Gonzalez-Sanchez, A., Abreu, R., Gross, H.-G., and van Gemund, A. J. (2011b). Prioritizing tests for fault localization through ambiguity group reduction. In *Proceedings of the 26th International Conference on Automated Software Engineering (ASE'11)*, pages xx–xx. (To appear).

[Gonzalez-Sanchez et al., 2011c] Gonzalez-Sanchez, A., Abreu, R., Gross, H.-G., and van Gemund, A. J. (2011c). Spectrum-based sequential diagnosis. In *Proceedings of the 25th International Conference on Artificial Intelligence (AAAI'11)*, pages xx–xx. (To appear).

[Gonzalez-Sanchez et al., 2011d] Gonzalez-Sanchez, A., Gross, H.-G., and van Gemund, A. J. (2011d). Modeling the diagnostic efficiency of regression test suites. In *Proc. TEBUG'11 (ICST'11 Workshop)*.

[Gonzalez-Sanchez et al., 2011e] Gonzalez-Sanchez, A., Piel, E., Abreu, R., Gross, H.-G., and van Gemund, A. J. (2011e). Prioritizing tests for software fault localization. *Software: Practice and Experience*. Accepted for publication.

[Gonzalez-Sanchez et al., 2009] Gonzalez-Sanchez, A., Piel, É., and Gross, H.-G. (2009). RiTMO: A method for runtime testability measurement and optimisation. In *Quality Software, 9th International Conference on*, Jeju, South Korea. IEEE Reliability Society.

[Gonzalez-Sanchez et al., 2010b] Gonzalez-Sanchez, A., Piel, É., Gross, H.-G., and van Gemund, A. J. (2010b). Minimising the preparation cost of runtime testing based on testability metrics. In *34th IEEE Software and Applications Conference*, Seoul, South Korea. (to appear).

[Gonzalez-Sanchez et al., 2010c] Gonzalez-Sanchez, A., Piel, E., Gross, H.-G., and van Gemund, A. J. (2010c). Prioritizing tests for software fault localization. volume 0, pages 42–51, Los Alamitos, CA, USA. IEEE Computer Society.

[Gonzalez-Sanchez et al., 2010d] Gonzalez-Sanchez, A., Piel, É., Gross, H.-G., and van Gemund, A. J. (2010d). Runtime testability in dynamic high-availability component-based systems. In *The Second International Conference on Advances in System Testing and Validation Lifecycle*, Nice, France. (to appear).

[Gonzalez-Sanchez et al., 2011f] Gonzalez-Sanchez, A., Piel, E., Gross, H.-G., and van Gemund, A. J. (2011f). A diagnostic point of view for the optimization of preparation costs in runtime testing. In *Proc. TEBUG'11 (ICST'11 Workshop)*.

[Gross, 2005] Gross, H.-G. (2005). *Component-Based Software Testing with UML*. Springer, Heidelberg.

[Hamlet and Voas, 1993] Hamlet, D. and Voas, J. (1993). Faults on its sleeve: amplifying software reliability testing. *SIGSOFT Software Engineering Notes*, 18(3):89–98.

[Hao et al., 2005] Hao, D., Zhang, L., Zhong, H., Mei, H., and Sun, J. (2005). Eliminating harmful redundancy for testing-based fault localization using test suite reduction: An experimental study. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 683–686, Washington, DC, USA. IEEE Computer Society.

[Harrold et al., 1993a] Harrold, M., Gupta, R., and Soffa, M. L. (1993a). A methodology for controlling the size of a test suite. *ACM TOSEM*, 2:270–285.

[Harrold et al., 1993b] Harrold, M. J., Gupta, R., and Soffa, M. L. (1993b). A methodology for controlling the size of a test suite. *ACM Transactions on Software Engineering Methodology*, 2(3):270–285.

[Heimdahl and George, 2004] Heimdahl, M. and George, D. (2004). Test-suite reduction for model based tests: effects on test quality and implications for testing. In *ASE'04*, pages 176–185.

[Huang and Lyu, 2005] Huang, C. Y. and Lyu, M. R. (2005). Optimal release time for software systems considering cost, testing-effort, and test efficiency. *IEEE Transactions on Reliability*, 54(4):583–591.

[Huang et al., 2006] Huang, G., Mei, H., and Yang, F.-Q. (2006). Runtime recovery and manipulation of software architecture of component-based systems. *Automated Software Engg.*, 13(2):257–281.

[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 *ICSE '94: Proceedings of the 16th international conference on Software engineering*, pages 191–200, Los Alamitos, CA, USA. IEEE Computer Society Press.

[Janssen et al., 2009] Janssen, T., Abreu, R., and Gemund, A. J. C. v. (2009). Zoltar: A toolset for automatic fault localization. In *ASE '09: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 662–664, Washington, DC, USA. IEEE Computer Society.

[Jiang et al., 2009a] Jiang, B., Zhang, Z., Chan, W. K., and Tse, T. H. (2009a). Adaptive random test case prioritization. In *ASE '09: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 233–244, Washington, DC, USA. IEEE Computer Society.

[Jiang et al., 2009b] Jiang, B., Zhang, Z., Tse, T. H., and Chen, T. Y. (2009b). How well do test case prioritization techniques support statistical fault localization. In *COMPSAC '09: Proceedings of the 2009 33rd Annual IEEE International Computer Software and Applications Conference*, pages 99–106, Washington, DC, USA. IEEE Computer Society.

[Johnson, 1960] Johnson, R. (1960). An information theory approach to diagnosis. In *Symposium on Reliability and Quality Control*.

[Jones et al., 2002] Jones, J. A., Harrold, M. J., and Stasko, J. (2002). Visualization of test information to assist fault localization. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 467–477, New York, NY, USA. ACM.

[Jungmayr, 2002] Jungmayr, S. (2002). Identifying test-critical dependencies. In *ICSM '02: Proceedings of the International Conference on Software Maintenance (ICSM'02)*, pages 404–413, Washington, DC, USA. IEEE Computer Society.

[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*, 33(4):225–237.

[Liblit, 2008] Liblit, B. (2008). Cooperative debugging with five hundred million test cases. In *ISSTA '08: Proceedings of the 2008 international symposium on Software testing and analysis*, pages 119–120, New York, NY, USA. ACM.

[Liu et al., 2005] Liu, C., Yan, X., Fei, L., Han, J., and Midkiff, S. P. (2005). Sober: statistical model-based bug localization. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 286–295, New York, NY, USA. ACM.

[Lorenzoli et al., 2008] Lorenzoli, D., Mariani, L., and Pezzè, M. (2008). Automatic generation of software behavioral models. In *Proc. 30th Int'l. Conf. on Softw. Eng. (ICSE'08)*, pages 501–510, Leipzig, Germany.

[Maier, 1998] Maier, M. W. (1998). Architecting principles for systems-of-systems. *Systems Engineering*, 1(4):267–284.

[Matevska and Hasselbring, 2007] Matevska, J. and Hasselbring, W. (2007). A scenario-based approach to increasing service availability at runtime reconfiguration of component-based systems. In *EUROMICRO '07: Proceedings of the 33rd EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 137–148, Washington, DC, USA. IEEE Computer Society.

[Murpy et al., 2009] Murpy, C., Kaiser, G., Vo, I., and Chu, M. (2009). Quality assurance of software applications using the in vivo testing approach. In *ICST '09: Proceedings of the 2nd international Conference on Software Testing*. IEEE Computer Society.

[Offutt and Hayes, 1996] Offutt, A. J. and Hayes, J. H. (1996). A semantic model of program faults. *SIGSOFT Software Engineering Notes*, 21(3):195–200.

[Offutt et al., 1996] Offutt, A. J., Lee, A., Rothermel, G., Untch, R. H., and Zapf, C. (1996). An experimental determination of sufficient mutant operators. *ACM TOSEM*, 5:99–118.

[Okumoto and Goel, 1980] Okumoto, K. and Goel, A. (1980). Optimum release time for software systems based on reliability and cost criteria. *Journal of Systems and Software*, 1:315–318.

[Raghavan et al., 1999] Raghavan, V., Shakeri, M., and Pattipati, K. (1999). Test sequencing algorithms with unreliable tests. *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, 29(4):347 –357.

[Reiter, 1987] Reiter, R. (1987). A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95.

[Renieris and Reiss, 2003] Renieris, M. and Reiss, S. P. (2003). Fault localization with nearest neighbor queries. *Automated Software Engineering, International Conference on*, 0:30.

[Rothermel and Harrold, 1993] Rothermel, G. and Harrold, M. (1993). A safe, efficient algorithm for regression test selection. In *ICSM'93*, pages 358 –367.

[Rothermel and Harrold, 1996] Rothermel, G. and Harrold, M. (1996). Analyzing regression test selection techniques. *IEEE TSE*, 22(8):529 –551.

[Rothermel and Harrold, 1997] Rothermel, G. and Harrold, M. (1997). A safe, efficient regression test selection technique. *ACM TOSEM*, 6:173–210.

[Rothermel et al., 1998] Rothermel, G., Harrold, M., Ostrin, J., and Hong, C. (1998). An empirical study of the effects of minimization on the fault detection capabilities of test suites. In *ICSM'98*, pages 34–43.

[Rothermel et al., 2001] Rothermel, G., Untch, R. J., and Chu, C. (2001). Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27(10):929–948.

[Santelices et al., 2009] Santelices, R., Jones, J. A., Yu, Y., and Harrold, M. J. (2009). Lightweight fault-localization using multiple coverage types. In *ICSE '09: Proceedings of the 31st International Conference on Software Engineering*, pages 56–66, Washington, DC, USA. IEEE Computer Society.

[Shakeri et al., 2000] Shakeri, M., Raghavan, V., Pattipati, K., and Patterson-Hine, A. (2000). Sequential testing algorithms for multiple fault diagnosis. *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, 30(1):1 –14.

[Shepperd and Ince, 1993] Shepperd, M. and Ince, D. (1993). *Derivation and Validation of Software Metrics*. Oxford University Press.

[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 *24th Annual ACM Symposium on Applied Computing (SAC'09)*, pages 461–467. ACM Press.

[Stenbakken et al., 1989] Stenbakken, G., Souders, T., and Stewart, G. (1989). Ambiguity groups and testability. *IEEE Trans. on Instr. and Meas.*, 38(5):941 –947.

[Suliman et al., 2006] Suliman, D., Paech, B., Borner, L., Atkinson, C., Brenner, D., Merdes, M., and Malaka, R. (2006). The MORABIT approach to runtime component testing. In *30th Annual International Computer Software and Applications Conference*, pages 171–176.

[Voas et al., 1991] Voas, J., Morrel, L., and Miller, K. (1991). Predicting where faults can hide from testing. *IEEE Software*, 8(2):41–48.

[Voas, 1992] Voas, J. M. (1992). Pie: A dynamic failure-based technique. *IEEE Transactions in Software Engineering*, 18(8):717–727.

[Voas and Miller, 1993] Voas, J. M. and Miller, K. W. (1993). Semantic metrics for software testability. *Journal of Systems and Software*, 20(3):207–216.

[Voas and Miller, 1995] Voas, J. M. and Miller, K. W. (1995). Software testability: The new verification. *IEEE Software*, 12(3):17–28.

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

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

[Wong et al., 1997] Wong, W. E., Horgan, J. R., London, S., and Bellcore, H. A. (1997). A study of effective regression testing in practice. In *ISSRE '97: Proceedings of the Eighth International Symposium on Software Reliability Engineering*, page 264, Washington, DC, USA. IEEE Computer Society.

[Woodward and Al-Khanjari, 2000] Woodward, M. R. and Al-Khanjari, Z. A. (2000). Testability, fault size and the domain-to-range ratio: An eternal triangle. *SIGSOFT Software Engineering Notes*, 25(5):168–172.

[Wotawa, 2010] Wotawa, F. (2010). Fault localization based on dynamic slicing and hitting-set computation. In *10th International Conference on Quality Software (QSIC'10)*, pages 161–170, Zhangjiajie, China. IEEE Computer Society.

[Wu et al., 2001] Wu, Y., Pan, D., and Chen, M.-H. (2001). Techniques for testing component-based software. In *Proceedings of the IEEE International Conference on Engineering of Complex Computer Systems*, pages 222–232, Los Alamitos, CA, USA. IEEE Computer Society.

[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 *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 201–210, New York, NY, USA. ACM.

[Zhang et al., 2009] Zhang, L., Hou, S.-S., Guo, C., Xie, T., and Mei, H. (2009). Time-aware test-case prioritization using integer linear programming. In *ISSTA '09: Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 213–224, New York, NY, USA. ACM.

[Zhao, 2006] Zhao, L. (2006). A new approach for software testability analysis. In *ICSE '06*, pages 985–988, New York, NY, USA. ACM.

[Zuse, 2000] Zuse, H. (2000). *A Framework of software measurement*. de Gruyter.

Summary

Cost Optimizations in Runtime Testing and Diagnosis

– Alberto González-Sánchez –

Software testing and diagnosis (debugging) is a time-consuming but rather important task for improving software reliability. Hence, testing and debugging are an integral part of the quality assurance process of large-scale Systems of Systems. It is therefore necessary to devise an appropriate verification strategy that not only achieves this reliability goal, but also does this at a *minimum* cost. Testing must be as little disruptive as possible for the running configuration.

The ultimate goal of testing is to prove the absence of faults for all possible inputs. Since exhaustive testing is hardly ever possible, testing typically aims at detecting the presence of faults, by trying to produce a failure (the external manifestation of a fault). The longer the program does not produce any failures, the more certain testers are that the program contains no faults. Literature comprises a myriad of techniques to efficiently produce failures, e.g., by improving coverage and/or reducing the testing cost. Once failures have been detected, the faults that produced them have to be localized. Fault localization is a whole research area on its own that has recently attracted great interest. The goal of (automated) fault localization is to reduce the effort that the software developers have to invest in manually localizing the fault.

Unfortunately for developers, it has recently been shown that effective fault detection and effective fault localization are contradictory goals. In particular, it has been shown that focusing exclusively on fault detection has a detrimental effect on fault localization, canceling the savings in testing cost by an increase in diagnostic cost. In this thesis we explore the causes for this cost increase, and propose a shift in the goal of testing, focusing on techniques that achieve much better fault localization performance, while maintaining fault detection performance.

Furthermore, since runtime tests will be carried out in the production system itself, we dedicate a part of this thesis to exploring further implications of runtime testing and diagnosis. In particular, the system's components have to be prepared for the execution of tests while also remaining functional, i.e., to be made runtime testable. Runtime testability requires an investment in preparation cost to ensure the tests maintain their fault detection and localization performance. In this thesis we will focus on the preparation costs required for runtime testability, and their trade-off with fault detection and localization costs.

From our study in diagnostic test prioritization, we concluded that, except in some cases, current test prioritization algorithms are not designed to optimize fault localization. We showed how the concept of information gain is the key to an optimal trade-off between fault detection and fault localization. As a solution, we introduced a novel, low-cost, on-line diagnostic test prioritization algorithm, named SEQUOIA. The main novelty of SEQUOIA is the fact that it can perform efficient test sequencing for diagnosis in systems with multiple faults and with test cases that can produce false negatives, by using a first order expansion of the set of minimal diagnostic candidates. However, when applied in practice, we showed that in certain circumstances derived from a large error in SEQUOIA's input parameters, its performance can be severely affected. Furthermore, we introduced a second low-cost diagnostic test prioritization algorithm, named RAPTOR. Unlike SEQUOIA, RAPTOR performs off-line prioritization and does not require precise input parameters to work. RAPTOR is able to obtain comparable performance to SEQUOIA, even outperforming it in many cases.

This thesis also makes the following contributions. First, as SEQUOIA, and Bayesian diagnosis in general, require the estimation of the false negative rate of tests before the diagnosis can commence, we researched the possibility of using a mutation study to obtain such information. Our results showed that this information can be used to obtain a good diagnosis, although the assumption of failure independence in the multiple-fault case does not hold in many cases. Second, as our experiments with SEQUOIA and RAPTOR showed clear differences in the improvements that can be obtained between programs, we investigated the cause. We concluded that the characteristics of test suites have a great influence, especially the coverage density and distribution of tests. Third, we presented runtime testability and the Runtime Testability Metric (RTM), that can be used to find the optimal investment in preparation costs to enhance the fault detection capability of our tests during runtime testing.

Samenvatting

Kostenoptimalisatie in executietijdtesten en -diagnose

– Alberto González-Sánchez –

Software testen en diagnose (debugging) is een tijdrovende maar belangrijke taak om de betrouwbaarheid van software te verbeteren. Vandaar date testen en debugging een integraal onderdeel vormen in het proces van de kwaliteitsgarantie van grote schaal Systemen van Systemen. Het is daarom nodig om een passende verificatiestrategie te ontwikkelen die niet alleen dit betrouwbaarheidsdoel behaalt, maar dit ook doet tegen een minimale kost. Het testen moet zo min mogelijk de lopende configuratie verstoren.

Het ultieme doel van deze testen is om de afwezigheid van fouten voor alle mogelijke inputs te bewijzen. Aangezien grondig testen nauwelijks mogelijk is, is het testen meestal gericht op het opsporen van de aanwezigheid van fouten, door te proberen om een foutmelding (de externe manifestatie van een fout) te veroorzaken. Hoe langer het programma geen foutmeldingen veroorzaakt, hoe zekerder ingenieurs zouden zijn dat het programma geen fouten bevat. De literatuur beschrijft meerdere methoden om eficiënt foutmeldingen te genereren, bijvoorbeeld door broncodedekking te verbeteren en/of door de kosten van het testen te verminderen. Zodra storingen zijn opgespoord, moeten de fouten die deze hebben veroorzaakt worden gelokaliseerd. Het lokaliseren van fouten is een breed onderzoeksgebied op zichzelf dat recent grote belangstelling heeft gekregen. Het doel van (geautomatiseerde) foutlokalisatie is het verminderen van de inspanningen die de software-ontwikkelaars moeten leveren om handmatig de fout te lokaliseren.

Helaas voor de ontwikkelaars, werd het onlangs bewezen dat effectieve foutopsporing en effectieve foutlokalisatie tegenstrijdige doelen zijn. In het bijzonder werd het bewezen dat uitsluitend focussen op foutopsporing een nadelig effect heeft op foutlokalisatie, wat de besparingen in het testen van de kosten teniet doet

door een toename in diagnostische kosten. In dit proefschrift onderzoeken we de oorzaken voor deze stijging van de kosten, en stellen we een verschuiving voor van het doel van deze testen, met de nadruk op technieken die veel betere prestaties voor foutlokalisatie kunnen leveren, terwijl ze de prestaties voor foutopsporing behouden.

Aangezien executietijdtesten zullen worden uitgevoerd in het productiesysteem zelf, wijden we bovendien een deel van dit proefschrift aan het verkennen van verdere implicaties van executietijdtesten en -diagnose. In het bijzonder moeten onderdelen van het systeem worden voorbereid voor de uitvoering van de testen, terwijl ze ook functioneel moeten blijven, i.e., om executietijdtestbaar gemaakt te worden. Executietijdtestbaarheid vereist een investering in voorbereidingskosten om ervoor te zorgen dat de tests hun prestaties voor foutopsporing en -lokalisatie behouden. In dit proefschrift zullen we ons richten op de voorbereidingskosten van de executietijdtestbaarheid en hun wisselwerking met de kosten voor foutdetectie en -lokalisatie.

Uit onze studie van diagnostische testprioritering, concludeerden we dat, behalve in sommige gevallen, de huidige testprioriteringsalgoritmen niet ontworpen zijn voor het optimaliseren van foutlokalisatie. We tonen aan hoe het concept van informatieverzameling de oplossing is voor een optimale wisselwerking tussen foutdetectie en foutlokalisatie. Als oplossing introduceren we een nieuw, low-cost, on-line, diagnostisch testprioriteringsalgoritme, genaamd SEQUOIA. De belangrijkste nieuwigheid van SEQUOIA is het feit dat het efficiënt test sequencing kan uitvoeren voor diagnose in systemen met meerdere fouten en met testcases die kunnen leiden tot valse negatieven, met behulp van een eerste orde uitbreiding van de set van minimale diagnostische kandidaten. We tonen aan dat in de praktijk bij bepaalde omstandigheden (afkomstig van een grote afwijking in de inputparameters van SEQUOIA) de prestaties zwaar worden beïnvloed. Verder introduceren we RAPTOR, een tweede low-cost diagnostisch testprioriteringsalgoritme. In tegenstelling tot SEQUOIA, voert RAPTOR off-line prioritering uit en vereist het geen nauwkeurige input parameters. RAPTOR is in staat om vergelijkbare en in vele gevallen zelfs betere prestaties te leveren als SEQUOIA.

Dit proefschrift behandelt ook de volgende onderwerpen. Ten eerste aangezien SEQUOIA, en bij uitbreding Bayesiaanse diagnose in het algemeen, de schatting van het percentage vals negatieve testen vereisen voordat de diagnose kan beginnen, hebben we onderzoek gedaan naar de mogelijkheid om een mutatie onderzoek te gebruiken om dergelijke informatie te verkrijgen. Onze resultaten tonen aan dat deze informatie kan gebruikt worden voor het verkrijgen van een goede diagnose, hoewel de veronderstelling van de onafhankelijkheid voor mislukking in het geval van meervoudige fouten in veel gevallen niet geldt. Ten tweede hebben we onderzoek gedaan naar de oorzaak van het verschil in verbeteringen die kunnen worden verkregen met diagnostische testprioriteringsalgoritmen aangezien onze experimenten duidelijke verschillen aantoonden. We concluderen dat de kenmerken van

de testreeksen een grote invloed hebben, met name de broncodedekkingdichtheid en de distributie van de tests. Ten derde presenteren we executietijdtestbaarheid en de Runtime Testability Metric (RTM), die kunnen worden gebruikt om de optimale investeringen in voorbereidingskosten te vinden om het foutdetectie vermogen van onze tests tijdens het executietijdtesten te verbeteren.

Curriculum Vitae

Personal Data

Full name: Alberto González Sánchez

Born: March 27^{th}, 1984 in Valladolid, Spain.

Education

2007 – 2011: Ph.D., Computer Science
Delft University of Technology, Delft, The Netherlands. Under the supervision of prof.dr.ir. A.J.C. van Gemund.

2005 – 2007: Ingeniero en Informática (*M.Sc., Computer Science*)
Universidad de Valladolid, Valladolid, Spain. Master Thesis performed at the Delft University of Techonlogy, Delft, The Netherlands.

2002 – 2005: Ingeniero Técnico en Informática de Sistemas (*B.Sc., Computer Engineering*)
Universidad de Valladolid, Valladolid, Spain. Graduation Project in collaboration with the IT department of the Parliament of Castile and Leon.

Work Experience

October 2007 – October 2011: Assistant in Opleiding (AIO). *Research Trainee*
Software Technology Department, Delft University of Technology. Mekelweg 4, 2628CD Delft, The Netherlands.

April 2007 – June 2007: Student Assistant
Faculty of Aerospace Engineering, Delft University of Technology. Kluyverweg 1, 2629HS Delft, The Netherlands.

July 2006 – September 2006: Internship
Faculty of Informatics, University of Valladolid. Campus Miguel Delibes s/n, 47011 Valladolid, Spain.

September 2005 – June 2006: Internship
Servicio de Informática, Cortes de Castilla y León. Fuensaldaña, Spain.