Faculdade de Engenharia da Universidade do Porto

FEUP

Reducing Screened Program Points for Efficient Error Detection

Ricardo Manuel Nascimento Castilho

Master in Informatics and Computing Engineering

Supervisor: Rui Maranhão (PhD)

28th June, 2010

Reducing Screened Program Points for Efficient Error Detection

Ricardo Manuel Nascimento Castilho

Master in Informatics and Computing Engineering

Approved in oral examination by the committee:

31st July, 2010

Abstract

Despite extensive testing in the development phase, residual defects can be a great threat to dependability in the operational phase. Lately there have been some studies that proved the utility of generic invariants, often dubbed *fault screeners* as error detectors within a spectrum-based fault localization (SFL) approach aimed to diagnose program defects in the operational phase. The generic nature of this generic screeners allows them to be automatically instrumented without any programmer-effort, while training is straightforward given the test cases available in the development phase. Experiments based on the Siemens program set demonstrate diagnostic performance that is similar to the traditional, development-time application of SFL based on the program pass/fail information known before-hand.

In this document we present a study that researches the possibility of further improving the performance of this automatic error detection technique by reducing the number of generic invariants used to screen a program. Here we describe the adopted metrics in order to evaluate the diagnosis quality of each screened program point. In the same baseline, we investigate about the variables that have the highest diagnosis effectiveness, the so-called *collar variables*, and how the current false negative rate of the techniques used hindered their discovery in our benchmark set up.

Resumo

Por mais intensiva que seja a etapa de testes durante a fase de desenvolvimento, a existência de defeitos residuais (*bugs*) representa sempre uma grande ameaça à fiabilidade do sistema aquando da fase operacional. Recentemente têm sido levados a cabo alguns estudos que provaram a utilidade dos *generic invariants*, frequentemente apelidados de *fault screeners*, como detectores de erros associados a um método localização automática de falhas baseado no espectro do programa. Método este que tem como objectivo diagnosticar defeitos do programa na fase operacional. A natureza genérica destes *generic screeners* permite que a sua instrumentação seja feita automaticamente sem qualquer esforço por parte do programador, o seu treino durante a fase de implementação também representa um processo bastante simples, sendo apenas necessário o acesso uma bateria de casos de teste adequada. Experiências baseadas no conjunto de programas disponibilizados pelo *Siemens-Set* demonstram que a performance de diagnóstico obtida é similar à da abordagem tradicional, cuja execução é feita durante a fase operacional e depende de informação prévia de passagem ou falha do programa.

Neste documento apresentamos um estudo que investiga a possibilidade de aumentar a eficiência deste método de detecção automática de erros, reduzindo o número de *generic invariants* utilizados para monitorar o programa. Aqui descrevemos as métricas adoptadas por forma a avaliar a qualidade de diagnóstico de cada um dos pontos do programa monitorados. Seguindo o mesmo raciocínio, investigamos quais são as variáveis que têm a maior eficácia de diagnóstico, as chamadas *collar-variables*, e como a taxa de falsos negativos relativa às técnicas utilizadas se tornou um entrave à descoberta das mesmas.

Acknowledgements

In the long, strange and yet beautiful and fulfilling journey that have been this master thesis, I certainly have a lot to be grateful for. So I would like to take a moment to thank to all the people that in some way have helped me throughout this whole process.

First of all I would like to thank to Dr.Ir. Rui Maranhão who has given me this amazing opportunity of doing a master thesis in the outstanding country that is The Netherlands. Rui's guidance, even from afar, has been impeccable and I am really thankful for all his patience, support, and interest.

I would also like to express my gratitude to Prof.Dr.Ir Arjan J. C. van Gemund for his ever cheerful and truly inspiring supervision. It has been a real honor to work with such an incredible gentleman of science.

A very special thanks is in order to Alberto Gonzalez, a PhD student at the time of this report writing, although he could have easily been a firefighter since he have consistently put out all the "fires" that have insisted to appear at every step of the way. Seriously dude, if it wasn't for you I would probably still trying to figure out why my script was shooting rainbows instead of calculating the program points' *C_d*...

Concerning the non-academic side of my life in the Netherlands, I need to thank to all of the people I have met and made friends with as they are responsible for making my staying here so much more pleasuring.

Many thanks for all my friends back home who never stopped to support and motivate me. Specially to my little gang, the "IeS" from Mirandela whose members are, without doubts, some of my dearest friends.

A word of gratitude, in Portuguese, to my family: my mother Cândida, my grandmother Alzira and my little brother Nuno. Apesar dos tempos extremamente difíceis que estávamos a atravessar na altura e da falta que eu fiz em Portugal, vocês puseram o meu futuro acima do vosso bem estar e tudo o que fizeram foi apoiar-me e incentivar-me durante estes 5 longos meses e isso é algo que eu nunca esquecerei. Do fundo do meu coração: Muito Obrigado.

Finally I would like to dedicate this thesis to the memory of my father, Mário Marques Castilho, who has passed away in the beginning of the year. He was by far the greatest man that I had the privilege to meet and my main role-model for every aspect of my life. He lived a full life by the principles of pride, strength and honor, until the very end, and his only regret was to not have been able to see any of his kids' graduation, as such this thesis is my last (humble) tribute to him.

Delft
June 28, 2010

Contents

List of Figures

LIST OF FIGURES

List of Tables

LIST OF TABLES

Abbreviations

LLVM	Low Level Virtual Machine
C_d	Cost of diagnosis
SFL	Spectrum-based Fault Localization
EFSM	Extended finite state machine
FMS	Finite state machine

ABBREVIATIONS

Chapter 1

Introduction

This chapter introduces the main aspects that form the context, motivation and goals of this master thesis. Here we also describe the outline used in this document.

1.1 Context

This thesis research is inserted in *Software Engineering* theme, more properly, the *Software Testing and Quality* area. This work is a contribution to the field of *Automatic Error Detection Techniques Based on Program Invariants*. The full process of researching, results analysis and report writing have been carried out in the Embedded Software Department of the Faculty Electrical Engineering, Mathematics and Computer Science on Delft University of Technology in Delft, the Netherlands.

1.2 Motivation and Goals

Software has replaced hardware as the base of embedded systems during the last years. Tasks that used to be done by integrated circuits are now done by complex software. And more complexity implies a higher error rate.

The electronic market is always putting manufacturers under pressure. Releasing a product before the competition is mandatory if they don't want to be left behind. That situation usually affects the quality of the product, since a shorter product lifespan usually means a shorter development time.

With a shorter software development period, a high presence of bugs is more than likely, because testing phases last less time than it requires. That is why traditional techniques are becoming increasingly more obsoletes, since they are not prepared to deal with this shorter testing periods.

Software updating is not easy. Operating systems usually require some sort of network connection in order to get its updates, which is fairly easy nowadays. But for an MP3 player or a TV, software updating might be impossible, since they are usually isolated systems. Or even because the firmware they use would be stored in a ROM. If these kind of products, which currently rely increasingly on software, are more sensitive to bugs, and fixing them after the product is been release is almost impossible, we are forced to find new solutions to keep the availability of this kind of systems.

If only the own system could be able to detect its own bugs and launch recovery routines, its availability could be kept, instead of letting software fall into a failure.

Recently there have been undergoing research [AZvG08, AZvG] with the goal of achieving just that. That line of investigation has uncovered, so far, a way to detect system errors (bugs) during the operational phase. This have been attained by imbuing error detectors in all of the system's variables and train those detectors to accept ”correct” values (values that do not generate system failures). Thus enabling the system to detect potential harmful input values during its execution. However, the fact that this approach applies the mentioned error detectors to all the system's variables puts a quite significant overhead in the system. Moreover, its clear that some variables have no diagnosis value at all. Thus one needs to selectively instrument only the most important variables.

With this research we proposed to find if it is possible to increase the performance of the existing error detection techniques, by reducing the overhead introduced in the system by them. In other words, we try to find a way to diminish the amount of points of the program that need to be screened in order to increase the system performance when in error detection mode. This should be done without losing diagnosis quality.

1.3 Thesis Layout

This thesis is organized as follows. Besides the introduction, it comprises 6 more chapters:

Chapter 2 In this chapter we introduce the thesis' key concepts.

Chapter 3 This chapter is dedicated to the presentation of prior work in areas relevant to our research. Our investigation is partially based on some of their contributions.

Chapter 4 In chapter 4, we further detail the purpose of this research as well as the research question we have proposed to answer. Here we also describe the approach used to investigate these questions.

Chapter 5 Here we explain our experimental set up, the development decisions adopted and the actual implementation details of the approach described in chapter 4.

Chapter 6 In this chapter we report on the experiments' results and interpret them in their relevance to answer our research questions.

Chapter 7 In chapter 6, we present the final conclusions of the research, along the main contributions of this project and future lines of work.

Chapter 2

Preliminaries

In this chapter we describe the key concepts that have been used in our research.

2.1 Faults, errors and failures

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 is the cause of an error in the system. Since we focus on computer programs, faults are bugs in the program code, and failures occur when the output for a given input deviates from the specified output for that input.

2.2 Siemens Set

In our study, we used a set of test programs known as the Siemens set [HFGO94]. The Siemens set is composed of seven programs. Every single program has a correct version and a set of faulty versions of the same program. Each faulty version contains exactly one fault. Each program also has a set of inputs that ensures full code coverage. Table 2.1 provides more information about the programs in the package. Although the Siemens set was not assembled with the purpose of testing fault diagnosis techniques, it is typically used by the research community as the set of programs to test their techniques. In total the Siemens set provides 132 programs. However, as no failures are observed in two of these programs, namely version 9 of schedule2 and version 32 of replace, are discarded. Besides, we also discard versions 4 and 6 of print tokens because the faults in this versions are in global variables and the profiling tool used in our experiments does not log the execution of these statements. In summary, we discarded 4 versions out of 132 provided by the suite, using 128 versions in our experiments.

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

Table 2.1: Siemens Set benchmark

2.3 Fault Localization

[Abr09] The process of pinpointing the fault(s) that led to symptoms (failures/errors) is called fault localization, and has been an active area of research for the past decades. Based on a set of observations, automatic approaches to software fault localization yield a list of likely fault locations, which is subsequently used either by the developer to focus the software debugging process, or as an input to automatic recovery mechanisms [PBB$^+$02, SÖ9]. Depending on the amount of knowledge that is required about the system's internal component structure and behavior, the most predominant approaches can be classified as (1) statistical approaches or (2) reasoning approaches. The former approach uses an abstraction of program traces, dynamically collected at runtime, to produce a list of likely candidates to be at fault, whereas the latter combines a static model of the expected behavior with a set of observations to compute the diagnostic report. Statistical approaches yield a diagnostic report with the M components ordered in terms of statistical evidence of being faulty. Reasoning approaches yield a diagnostic report that comprise diagnostic candidates. Diagnosis candidate (possibly multiple-fault) that explain the observations, ordered in terms of probability. In this thesis we will focus solely on statistics-based approaches.

Statistics-based fault localization techniques use an abstraction of program traces, also known as program spectra, to find a statistical relationship with observed failures. Program spectra are collected at run-time, during the execution of the program, and many different forms exist [HRS$^+$00]. For example, component-hit spectra indicate whether a component was involved in the execution of the program or not. In contrast to model-based approaches, program spectra and pass/fail information are the only dynamic source of information used by statistics-based techniques. Well-known examples of such approaches are the Tarantula tool by Jones, Harrold, and Stasko [JHS02], the Nearest Neighbor technique by Renieris and Reiss [RR03], the Sober tool by Lui, Yan, Fei, Han, and Midkiff [LFY$^+$06], the work of Liu and Han [Liu06], PPDG by Baah, Podgurski, and Harrold [BPH08], CrossTab by Wong, Wei, Qi, and Zap [WWQZ08], the Cooperative Bug Isolation by Liblit and his colleagues [LNZ$^+$05, Lib08, ACRL07, ZLN06], the Pinpoint tool by Cheng and his colleagues [CKF+02], the AMPLE tool by Dallmeier, Lindig, and Zeller [DLZ05], the work by Steimann, Eichst'adt-Engelen, and Schaaf [SEES08], and the TimeWill Tell approach by Yilmaz, Paradkar, and Williams [YPW08].

2.4 Invariant based error detection

Generally we detect errors in a system by implementing tests (invariants) that typically trigger some exception handling process This so-called program invariants [ECGN01] can be program-specific (e.g., a user-programmed test to assert that two state variables in two different components are in sync) or generic (e.g., a compiler generated value range check). While program-specific invariants cover failures anticipated by the developer and a low rate of false positives and negatives, it's a costly and error-prone process because of its manual integration. In the other hand, generic program invariants are very simple and cost-effective but have high rates of false positives and negatives. This last type of detectors requires training to adapt to the application-specific program profile. This training is automatically performed during testing at development time.

Generic invariants are often subbed fault screeners and they have been extensively been studied for both software and hardware applications. There are several models proposed:

- Value screeners or bitmask invariants [HL02, RCMM07]. Very simple generic invariants, based in the value check of a program's variable. They have two components, the first observed value fst and a bitmask msk representing the activated bits (all bits start set to 1) every time the invariant is used the bits new are verified according to the following:

  $violation = (new \oplus fst) \wedge msk$

  If the violation is non-zero, an error is flagged, if in error detection mode (operational phase). During training mode (development phase) the invariant is updated according to:

  $msk = (new \oplus fst) \wedge msk$

- (Dynamic) Range screeners [HL02, RCMM07]: Generic program invariants which are used to bound a program's variable range. Whenever a new value v is found it is compared against the lower bound l and the upper bound u in the following way:

  $violation = \neg(l < v < u)$

If v is outside these bounds an error is flagged if in deployment phase or, if in training phase, the range of v is extended:

$$l := min(l,v)$$
$$u := max(l,v)$$

- Bloom filters [AZvG08, RCMM07, Blo70]: Generic program invariants composed by probabilistic data structures that verify if an element is member of a set. This screener collects and stores a compact representation of a variable's entire history. All variables share the same Bloom filter, which is nothing more than a bit array of 64KB (although Racunas et. al proved that the size of the filter could be reduced by using a backup filter to prevent saturation [RCMM07]). Each 32-bit value v and instruction address ia are merged into a single 32-bit number g:

  $$g = (v * 216) \vee (0xFFFF \wedge ia)$$

  where $\vee$ and $\wedge$ are bitwise operators, respectively. This number g is used as input to two hash functions ($h1$ and $h2$), which index into the Bloom filter b. During training mode, the outputs of the hash functions are used to update the Bloom filter according to the assignment

  $$b[h1(g)] := 1; b[h2(g)] := 1$$

  In detection mode an error is flagged according to $violation = \neg(b[h1(g)] \wedge [h2(g)])$

2.5 Collar Variables and State Clumps

The concept of collar variables is a very simple and intriguing one. It as been discovered and rediscovered in multiple AI researches and it can be summarized as regularities in the behavior of a system. Such regularities could be translated, in practical terms, as a small number of key variables that determine the behavior of the rest of the system. They have the following features:

- Software testing should quickly saturate—most program paths will be exercised early, with little further improvement seen as testing continues [MJ96].
- Random mutation will be more likely to affect the many non-collar variables compared to the few collar variables, so the net effect of those mutations would be small (a mutant of a program is a syntactically valid but randomly selected variation

to a program, such as swapping all plus signs to a minus sign). If so, then most random mutations of a program containing collars will not change the program's behavior [WWQZ08, Bud80, Acr80].

- Software states should clump so that only a small number of states will be reached at runtime. Collars imply clumps because the number of reachable states in an application will be quite small, containing just the number of possible settings to the collar [Pel04].

Chapter 3

Related Work

In this chapter we focus in the actual literature about this thesis theme. We present a small summary of the most relevant researches done in the field. Furthermore we make brief references to prior work whenever that is in order.

3.1 Collar Variables and State Clumps

Tim Menzies, David Owen and Julian Richardson [MOR07] have questioned themselves about the reason why the modern software works at all. Given their example of a simple model with 300 boolean variables which, doing the math, have $2^{300} = 10^{90}$ different states. Considering the astronomers estimate that the universe holds 10^{24} stars, that little model has more internal states then stars in the sky, they say. With this in mind, they tried to understand why the current, flawed and incomplete, test methods weren't missing too many critical errors. The answer seems to reside in recent AI research which revealed certain regularities in the behavior of AI systems. Such regularities could be translated, in practical terms, as a small number of key variables that determine the behavior of the rest of the system. These small set of variables, if present in conventional (non-AI software), could bring great benefits to the software engineering field. That is, ultimately, because when this variable set is present, the problem of controlling software reduces to just the problem of controlling the variables in that set. As such, a developer searching for bugs can narrow the test battery to only include the range of the variables in that set. Consequently much of we can find by using complex and costly methods it can be also found by random ones. This is of relevance because it is a fact that a random, incomplete algorithm can often be the simplest or the fastest method available (in some cases both!) [MR95]. These set of key variables were extensively studied by AI researchers being discovered and rediscovered multiple times and given various names [KJ97, DB86, CB94, WG03].

In the research area where this thesis is the common term used while referring to these set is collar variables [MR06]. Eventually this concept was tested on conventional software and the expected effects:

- Software testing should quickly saturate—most program paths will be exercised early, with little further improvement seen as testing continues [MJ96].
- Random mutation will be more likely to affect the many non-collar variables compared to the few collar variables, so the net effect of those mutations would be small (a mutant of a program is a syntactically valid but randomly selected variation to a program, such as swapping all plus signs to a minus sign). If so, then most random mutations of a program containing collars will not change the program's behavior [MW95, Bud80, Acr80].
- Software states should clump so that only a small number of states will be reached at runtime. Collars imply clumps because the number of reachable states in an application will be quite small, containing just the number of possible settings to the collar [Pel04].

were all verified without exception! These studies introduced the concepts of collar variables and state clumps in the software engineering field. Currently there are being developed two algorithms that explore this concepts, TAR3 and LURCH:

- TAR3 is a randomized version of the TAR2 data miner [MH03]. TAR3 inputs a set of scored examples and outputs contrast rules that distinguish highly scored examples from the others. The rule generation algorithm seeks the smallest set of rules that most select for the highest scoring examples. To find the collar variables, TAR3 assumes that if collars exist, they control software's behavior. So, a random selection of the software behaviors must, by definition, sample the collars. That is, we need not search for the collars—they'll find us.
- LURCH is a random testing and debugging tool for finite-state models [ODC06]. LURCH works by simulating the execution of the models, choosing randomly when more than one transition is possible. LURCH never backtracks; it simply runs until some termination condition (such as path end, depth limit, or error detected), and then starts over until a user-specified number of paths have been explored.

3.2 Application-Based Metrics for Strategic Placement of Detectors

Pattabirman and his co-workers have researched the strategic placement of error detectors within programs in order to to preemptively detect crashes due to errors in data values used in the program [PKI05]. For this paper they have developed a model for error propagation and crashes, from that model derived several metrics to guide the strategic placement of detectors and evaluates (using fault injection) the coverage provided by ideal detectors embedded at program locations selected using the computed metrics. It was shown that the strategic placement of detectors can increase crash coverage by an order-of-magnitude compared to random placement, with a low percentage of false-positives. This work is of relevance for this thesis, as well as for the previous research on this thesis theme. That is for it's contributions, when applied to automatic error detectors, helps to maximize the error coverage of systems in which such detectors are integrated while minimizing the overhead caused with the insertion of those.

3.3 Automatic Generation of Software Behavioral Models

Dynamic analysis of software systems produces behavioral models that are useful for analysis, verification and testing. The main techniques for extracting models of functional behavior generate either models of constraints on data, usually in the form of boolean expressions, or models of interactions between components. This models are usually in the form of finite state machines. Both data and interaction models are useful for analyzing and verifying different aspects of software behavior, but none of them captures the complex interplay between data values and components interactions. Thus related analysis and testing techniques can miss important information. Lorenzoli and his team attempted to research precisely on that. Their study has focused on the generation of models of relations between data values and component interactions [LMP08]. Moreover, they present GK-tail, a dynamic analysis technique that produces models of the behavior of software systems in the form of EFSMs. These models represent constraints on data values, properties of interaction patterns, as well as their interplay. The early experiences they execute, using third-party applications gained with a prototype implementation shows the usefulness and feasibility of the approach. The data collected during their study, using third-party applications and design patterns, indicates that relevant behaviors of software systems depend on the interplays between data and interaction patterns, and can be captured by EFSM models, but not by constraints on data and FSM produced separately. When GK-tail generates models of component interactions, the data collected confirm the hypothesis of scalability of the approach, since the complexity of GK-tail does not depend on the size of the analyzed systems, but rather on the complexity of the interactions between components. EFSM models, as all dynamic models, depend on the quality of the test suites used to produce them.
However the results obtained with this paper are not conclusive, further testing must be taken, namely using larger applications, in order to confirm this findings.

3.4 Automatic Software Fault Localization using Generic Program Invariants

In [AZvG08] it have been studied the utility of generic invariants as error detectors within a SFL approach applied to operational phase rather than just the development phase. The authors' motivation laid in the fact that merely testing during the operational phase is not enough nowadays because of the extreme complexity of the software, as such fault diagnosis during the operational is more and more desirable. Additionally there were no research regarding the use of program invariants to help locate the root cause of a failure, moreover SFL techniques were never used in congruence with automatic error detection.

During the compiling stage the program is instrumented (using the optimization pass for LLVM) to generate the statement spectra and execute the invariants. In the Training stage the program is run for the test cases which it passes, in during this phase the invariants work in training mode. In the operating stage the entire program test cases are executed, in which the invariants are executed in detection mode. Both errors and spectra are collected in data bases which are input to the SFL component. In order to prevent eventual corruptions of the logged data they located the program invariants and the spectra in an external component ("Screener"). The program points screened are all memory loads/stores, function arguments and return values. The programs and test cases used in this experiment are included in the Siemens Set [HFGO94]. The program invariants used in this experiment were the bitmask invariant and the range screener. This experiment's results shows that the diagnosis performance of SFL while using program invariants during the operational phase is similar to the one obtained in the development phase. As such it's possible to attain fault diagnosis with accuracy comparable to the likes of those done in the development phase at the operational phase, with no human interaction and with a reasonable overhead (14% with the Siemens Set). Also it was proved that this overhead can be reduced at the expense of a reasonable diagnosis performance penalty.

3.5 On the performance of fault screeners in software development and deployment

In [AZvG] have analyzed analytically and empirically the performance of generic program invariants, namely range and Bloom filter invariants as error detectors. Although there is a fairly large amount of research on this subject prior to this study [RCMM07, Blo70, HL02, EPG^{+}07] , the results of all that research were obtained empirically, as useful as that may be, it was important to create a analytical model to evaluate more accurately the fault screeners performance. This research had focused on that.

Their study has shown, with an empirical analysis, that near-“ideal” screeners, such as Bloom- filters, need more training than simpler screeners, such as range invariants, however their false negatives rate is lower. The analytical model created and used in this research has proved that the training required by near-“ideal” scales with the variable domain size, while simpler screeners need constant training time. Thus the model is in total agreement with the empirical findings. Additionally they used the analytical model to confirm the empirical findings on [AZvG08], also with positive results.

Further research must be taken in order to convert the metrics used to measure the performance in the screener-SFL to the more common T-Score [JH05, RR03] in order to allow a direct comparison with stand-alone screener performance, for the current metrics provide some comparison term but not a very good one. Moreover, additional studies should be taken in order to decrease the overhead created when using screeners during the operational phase.

Chapter 4

Research Questions and Approach

In this chapter we first present this thesis problem in a more detailed manner. After that we expose an abstract, high-level explanation of the work method adopted.

4.1 Motivation

SFL can be used with simple fault screeners [AZvG08, AZvG]. However, instrumenting every single program point is too expensive. Moreover, its clear that some program points have no diagnosis value at all [PKI05, MOR07]. Thus one needs to selectively instrument only the most important variables. As such, in this thesis, we try to find out which variables are important for a program, the, so-called, collar variables. The benefits of accomplishing this would greatly contribute for the development of the area, for such findings would certainly improve the performance of the invariant-SFL approach [AZvG08, AZvG]. This is because the interesting points during the error detection process would decrease from all the variables present in the system to only a small subset of those variables.

4.2 Problem

The main propose of this study was to answer the following academic question: "Is it possible to reduce the amount of program points that need to be screened in a program without losing diagnosis accuracy?". More specifically, we wanted to know if, given the exact same experimental environment and set-up used in [AZvG08, AZvG], we can obtain similar false positive and negative rates between the standard program instrumentation (instrumenting all program points) and a reduced instrumentation, featuring only a small set of the most relevant program points. This question can be divided in two other questions:

- "What criteria we will use to select this small set of program points? "
- "Is there some sort of pattern that we can detect to consistently and automatically find which ones are the "best" variables for each program?"

4.3 Approach

In order to fulfill the thesis objectives we start by determining which metrics could be used to measure the importance of each program point and then investigate which ones are the most important.

The first metric to be defined has been the ***cost of diagnosis***, this metric measures the effort needed to find the fault given, in our case, a certain program point. Such value can be obtained by taking a program point and consider it as the only diagnostically-relevant fragment of the program, deeming the pass/fail results for that program point, given set of test cases, as the global pass/fail results for the whole program and then feed this pass/fail information, as well as the standard program spectra's data, to a SFL tool and further processing the obtained data. Thus, this metric enables a qualitative measurement of a program point's diagnosis capabilities.

The second metric defined has been the ***number of violations***, this measures the number of times the test conditions defined by the fault screeners on a program point are violated, meaning, the number of *bugs* detected for that program point. Such measurements are obtained simply by counting the number of times a screener violation occurred for a program point, given a set of ran test-cases. This very plain and straightforward metric enables a quantitative measurement of a program point's diagnosis capabilities, also it provides some empirical knowledge on the value coverage of the ran test cases.

The third and last metric defined has been the ***distance to the fault***, this measures the distance (in number of statements) between the faulty statement of a program and a program point. This metric enables us to give some context to each program point as well as to assess if there is some correlation between a program point's diagnosis capabilities and its proximity to the fault.

Decided the metrics, we have developed a method to calculate them for each program point of a program, trying to pinpoint, this way, the "best" program points, and thus variables, of a program. The diagram depicted by figure 4.1 shows just that:

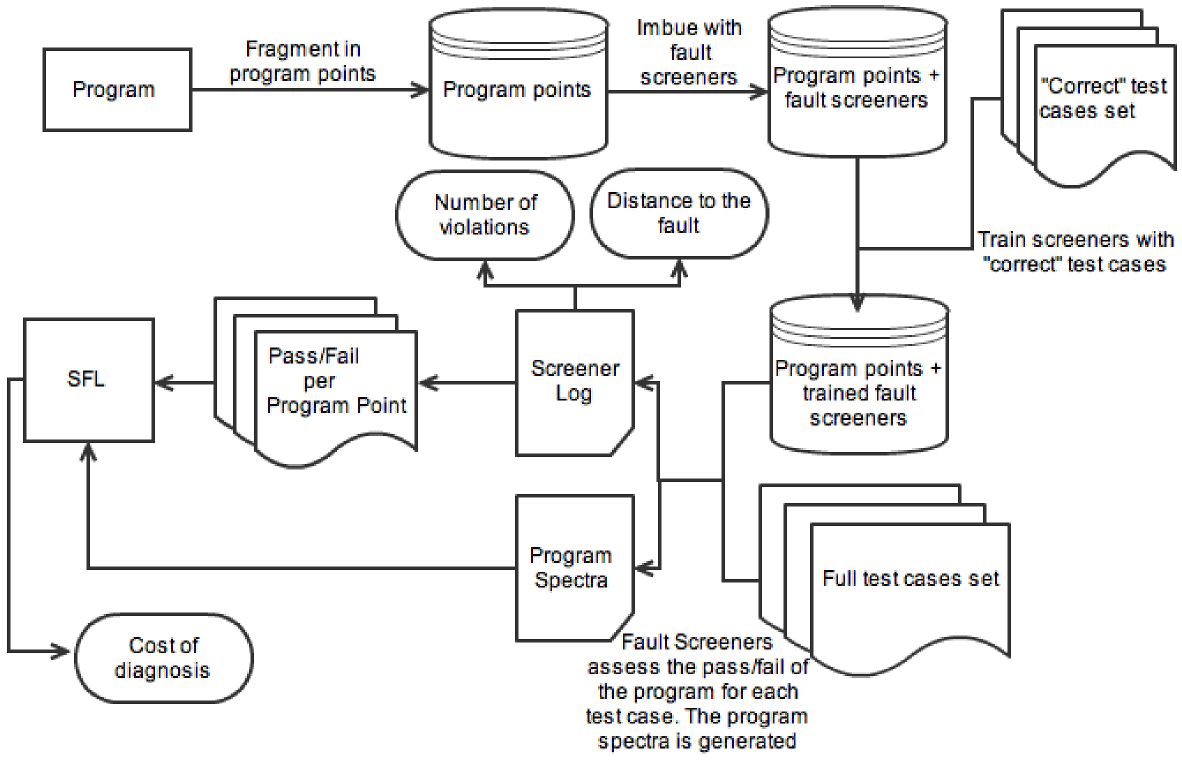

Program
Fragment in program points
Program points
Imbue with fault screeners
Program points + fault screeners
"Correct" test cases set
Train screeners with "correct" test cases
Number of violations
Distance to the fault
SFL
Pass/Fail per Program Point
Screener Log
Program points + trained fault screeners
Program Spectra
Full test cases set
Cost of diagnosis
Fault Screeners assess the pass/fail of the program for each test case. The program spectra is generated

Figure 4.1: Pinpoint Relevant Program Points Experiment

As we can see in figure 4.1, the process of obtaining our three metrics per program point comprises several steps. First the program is broke down in program points, then each of those program points is instrumented (i.e imbued with fault screeners) and their fault screeners are trained with test cases that cause the system to attain only faulty-free states, this enables the screeners to adapt to the program. Afterwards the program is ran with a more complete set of test cases, also featuring inputs that can activate the program's bugs and thus lead to faulty program states. During this step the fault screeners work as error detectors and its error report is stored n a global Screener Log. At the same time the program's spectrum is being generated for each program's run, creating a global matrix containing all of the Program Spectra for all the test cases. From the Screener Log we can directly extract the *distance to the fault* of each program point, as it is an information directly given in the log. We can also calculate the *number of violations* of each program point with ease as it can be done simply by counting the number of times a program point's screener detects an error. Posteriorly we filter the program's pass/fail assessment provided by each program point's screener for each of the test cases. We feed this information and the Program Spectra data to an SFL tool and from there we obtain the *cost of diagnosis* each program point.

4.4 Summary

Our study pretends to answer a very clear question: "Is it possible to reduce the amount of program points that need to be screened in a program without losing diagnosis accuracy?". To answer this we have adopted three diagnosis metrics in order to measure the diagnosis potential of each program point of a given program. Those metrics are: (1) *distance to the fault*, (2) *cost of diagnosis* and (3) *number of violations*. In our approach we take each program point as an oracle and make it detect program errors. By analyzing the errors' log we can determine both the program points' *distance to the fault* and its *number of violations*. The error log can also be fed to a diagnosis tool which will be able to calculate the program points' *cost of diagnosis*.

Chapter 5

Implementation

In this chapter we present, in detail, the approach we have used for our study, which is already summarily explained in chapter 4.3. It is divided in 6 sections:

Implementation decisions Enumerates the decisions that had to be made during the implementation of the experimental procedure.

Experimental setup Describes how we have prepared the ground for our experiment.

Invariant training and testing Shows how the invariant training and testing process and its respective results have been automated for our experimental setup.

Spectra generation Describes how we have obtained the program's spectra for all the adopted benchmark's test cases.

Program points' metrics calculation algorithm Presents the developed method to calculate both the number of violations and the cost of diagnosis of each program point over its distance to the fault.

Results processors Exposes the implemented automatic plot generator and programs' metrics calculator.

Summary A short, summarized description of the chapter

5.1 Implementation decisions

For our study experiments we have needed an automatic error detection tool capable of inserting program invariants in different points of the code, train them to accept a set of values and to further test them and report the results. The obvious tool to perform this role have been Zoltar [JAG09], an award winning tool set for Automatic Fault Localization. This is due to it is a state of the art tool in this matter and the only one capable to carry out the task at hands. Given the set of invariants that Zoltar toolset offers we have chosen to instrument each variable's load and store operations as they are the most frequent in everyone of the adopted benchmark's programs. In other words, the program points we consider in our research are all the operations of load and store of each program. In example, the following statement:

```
a=b;
```

is instrumented in two program points, one of the type store and associated to the variable *a* and another of the type load and associated to the variable *b*. In order to monitor and train the instrumented variables we have selected both bit mask and range screeners as those have been the only ones implemented in Zoltar at this reports's time of writing.

The benchmark used to feed Zoltar was the Siemens Set, although set was not assembled with the purpose of testing fault diagnosis techniques, it is the one typically used by the research community. Moreover it was already used with Zoltar in prior researches. So by adopting it we have had both the guaranty of instant tool-benchmark compatibility and the ability to foresee, to some extent, the behavior of the benchmark's programs.

5.2 Experimental setup

Before our research's experimental phase begun we have needed to set up our work environment. First we have had to select a solid and fast system as our main workplace due to the very large amount of information we have needed to process. As such, the adopted system was *apsthree*, a server from the Software Technology Department at Delft University of Technology. During our research the system had a Intel Xeon 2.8GHz processor and 32GB ram. It ran on the operating system Fedora Core 12 and provided a tc shell.

Selected the workplace we have installed the tools needed for the implementation of our study's approach. Besides the Zoltar toolset, we have had to install the 2.6 version of LLVM [LA04] framework since the mentioned toolset depends on it to work properly. The installation details of both tools are described in appendix A.1 and A.2.

The Benchmark adopted was the Siemens Set [HFGO94], which have not required any type of installation whatsoever since it essentially consists in series of simple files. In order to proceeded to the instrumentation of each variable involved in load or store operations for each version of each of the seven Siemens Set's programs we used a pre-made makefile system, from prior researches, which is able to propagate a single set of makefile commands throughout all benchmark's programs and versions. We have just had to change the actual *make* statement as it have been originally created to instrument the benchmark using gcov tool. Thus we altered if to issue a Zoltar instrumentation command instead:

```
...
7       all: $(PROGRAM).s
8               gcc $(PROGRAM).s -lpthread -linstrument -o \
                $(PROGRAM) -L/path/to/zoltar/lib
9
10      $(PROGRAM).s: $(PROGRAM).ibc
11              llc -f $(PROGRAM).ibc -o $(PROGRAM).s
12
13      $(PROGRAM).ibc: $(PROGRAM).bc
14              instrument -f -spstatement -invstore \
                 -invload -bypassmain $(PROGRAM).bc -o $(PROGRAM).ibc
15
16      $(PROGRAM).bc: $(PROGRAM).c
17              llvm-gcc -g -emit-llvm -c $(PROGRAM).c -o $(PROGRAM).bc
...
```

The statement in line 17 compiles the source code through LLVM which splits it in various program points. Then, in line 14, the variables are instrumented, per statement (*-spstatement* option), into invariants if they are involved in a load or store operation (*-invload* and *-invstore* options). The statements in lines 11 and 8 finish the instrumentation process. After this slight tune up all one needs to instrument the full Siemens Set, using Zoltar, is to execute *make* in a terminal, at the benchmark's top directory. This operation generates a file, named datafile.dat, in each instrumented program's version, which stores the data concerning that version's instrumented variables.

5.3 Invariant training and testing

In order to train and test the instrumented invariants we need to run each program version of the benchmark with different test cases. The Siemens Set programs' versions need hundreds of *correct test cases* in order to achieve an accurate invariant training. And some thousand of *untrained test cases* to test the invariants in order to obtain a satisfactory error detection rate. Although the Siemens Set itself supplies such test cases, it is unthinkable to run all those tests cases by hand. Therefore we developed a Python script that automatizes the process.

The script runs one time for each version of each program of the benchmark, meaning that its implementation is only version-wise. Thus, for each version, the script first reads a file containing the expected results for each input, provided by the benchmark. With that data it tags each test case as "correct" or "incorrect" as it should pass or should fail, respectively.

```
...
for i, res in enumerate(REFERENCE_RESULTS):
        if REFERENCE_RESULTS[i] == '+\n':
                CORRECT_TEST_CASES.append(INPUTS[i])
        else:
                INCORRECT_TEST_CASES.append(INPUTS[i])
        ...
```

Figure 5.1: Test cases tagging

The correct test cases are the ones applied to the instrumented invariants during training phase. While the remaining are ran over them during the test phase, thus enabling the fault screeners error detection. The *stderr* output of each test case ran during the test phase is capture to a file, as it contains a report of the screener violations that occur in that run.

```
...
#Trains the invariants with the correct test cases
os.system('zoltar --settrain')
for i, inp in enumerate(TRAINED_TEST_CASES):
        cmd = 'FAULT=%s %s/%s %s' % (FAULT, PATH, PROGRAM, inp)
        sut = subprocess.Popen(cmd, shell = True, cwd=TOP,
                                                stdout= subprocess.PIPE)
        sut.wait()

#Tests the invariants with all the untrained test cases
os.system('zoltar --settest')
for i, inp in enumerate(INCORRECT_TEST_CASES):
        #runs the test case and catches its output
        cmd = 'FAULT=%s %s/%s %s' % (FAULT, PATH, PROGRAM, inp)
        sut = subprocess.Popen(cmd, shell = True, cwd=TOP,
                                                 stderr= subprocess.PIPE)
                sutout = sut.communicate()
                open('stderr', 'a').write(sutout[1])
                ...
```

Figure 5.2: Training and testing invariants

5.4 Spectra generation

There was a script prior to our study that already extracted the spectra data of each test case ran for every versions of every program in the benchmark to individual files. We just have had to convert the format of the output file, which have been in *.numpy*, to *.txt*. Similarly to the script that trains and tests the invariants (see section 5.3) this one iterates over all test cases (inputs) and stores their output for further analysis. After getting the output inserts applies the *gcov* tool to generate the spectra of the program run.

```
...
for i, inp in enumerate(INPUTS):
    cmd = '%s/%s %s' % (PATH, PROGRAM, inp)
    refcmd = '%s/source/%s %s' % (TOP, PROGRAM, inp)

    tries = 4
    while tries > 0:
      try:
        past = time.time()
        sut = subprocess.Popen(cmd, shell=True, cwd=TOP,
        stdout=subprocess.PIPE)
        ret = sut.wait()
        now = time.time()

        sutout = sut.communicate()

        ora = subprocess.Popen(refcmd, shell=True, cwd=TOP,
                        stdout=subprocess.PIPE)
        ret2 = ora.wait()

        oraout = ora.communicate()

        error[i] = sutout != oraout

        subprocess.call('gcov %s' % PROGRAM, shell=True,
                stdout=file('/dev/null'))

        fill_spectrum('%s.c.gcov' % PROGRAM, spectrum, i)
        costs[i] = (now - past) * 1000.0
...
```

Figure 5.3: Generating program's spectra

5.5 Program points' metrics calculation algorithm

In this section we describe how we have obtained the values of *distance to the fault*, *number of violations* and *cost of diagnosis* for each instrumented program point in the benchmark.

5.5.1 *distance to the fault* calculation

This metric calculation have not been an immediate one. First we have tried a traditional procedure and tried to generate a Program Dependence Graph (*PDG*) directly from LLVM and compute the shortest path from the fault to each program point. However this is not supported for our source code fragmentation into program points by statement but rather, just for the standard division by the so-called *basic blocks*. As such we have attempted to use the opt tool and succeed. Still, the generated *PDG* format has not had an directly usable format and has been too data-intensive to edit. Moreover, the generated PDG data considered the *distance to the fault* of the program points contained in loops the same throughout the loops' iterations.Thus we have had to came up with a more creative solution. This have consisted in slightly tune up the actual Zoltar code in order to make it calculate and return the distance value at every fault screener violation reported. The adjustment made have been to basically include the *distance* parameter in the screener violation report issued by Zoltar. This parameter is calculated by comparing the position of the fault with the current statement position, during program's test case run, if the current statement is on or after the fault's location the distance starts to be incremented at each ran statement. At each screener violation the current value of *distance* is printed. In order to implement this algorithm we have started by adding a new attribute called *distance* to the structure that stores the invariant information and initialize it with -1 value:

```
void _initInstrumentationInfo(unsigned int timestamp) {
  ...
  _instrumentationInfo = (_InstrumentationInfo*)
                                calloc(1, sizeof(_InstrumentationInfo));
   ...
  _instrumentationInfo->training = 1;
  _instrumentationInfo->run = 0;
  _instrumentationInfo->distance= -1;
  ...
  }
```

Figure 5.4: Adding *distance* to the instrumentation structure

After this we have calibrated the spectra updating function to get the current version's fault location, which is passed to Zoltar through a environment variable as a test case is ran, and to compare, at each spectra update, the actual statement count with the fault location. If that number is lower than the fault it would mean that the program still have not reached the faulty statement, thus maintaining the distance value at is default, -1. However if the current statement count is equal or higher than the fault it would mean that the present program point is located either on the fault or near it. In this case the distance value is updated to the actual statement count and suffers further updates at each spectra updating.

```
 /* spectrum update function */
void _updateSpectrum(unsigned int spectrumIndex,
                     unsigned int componentIndex) {

  char *fault_s = getenv("FAULT");
  unsigned int fault = atoi(fault_s);
...
  SPECTRUMDATA(spectrumIndex)[componentIndex]++;

  if (_instrumentationInfo->distance >= 0)
        _instrumentationInfo->distance = _instrumentationInfo->distance + 1;

  if(componentIndex == fault)
        _instrumentationInfo->distance = 0;
...
}
```

Figure 5.5: *distance* calculations

Then we have modified the fault screener violations reporting functions to include the current distance count by the time of each detected violation. Thus making that information available at the *stderr* output obtained when running incorrect test cases during fault screeners test phase. (see section 5.3)

```
 void _handleInvariantBitmaskError(_Invariant *inv) {
  fprintf(stderr, "handle invariant bitmask error:\n");
  fprintf(stderr, "  val       = %08x\n", inv->range.u.val);
  fprintf(stderr, "  first     = %08x\n", inv->bitmask.first);
  fprintf(stderr, "  mask      = %08x\n", inv->bitmask.mask);
  fprintf(stderr, "  first^val = %08x\n\n", (inv->bitmask.first ^
        (unsigned int) inv->range.u.val));

...
  inv->nErrors++;
  fprintf(stderr, "  errors    = %d\n\n", inv->nErrors);
  fprintf(stderr, "  distance  = %d\n\n",
                _instrumentationInfo->distance);
...
  }
```

Figure 5.6: Adding *distance* information to the violation report

5.5.2 *cost of diagnosis* calculation

The *cost of diagnosis* of a program point is our study's core metric for the results evaluation. Also it had been the hardest to calculate. In order to compute this metric we use each program point has an oracle and produce a ranked list of likely faulty locations within that program point. After this data is obtained, the cost of diagnosis calculation is trivial. In order to generate the mentioned list we first read all the instrumented invariants data from the file *context.dat*, created during their training and testing phase, and store it into class objects of the type *Program_Point*. Each *Program_Point* contains information regarding its correspondent variable name, invariant index given by LLVM during instrumentation, invariant type (load or store), number of screener violations detected, cost of diagnosis, line of code number and distance to the fault:

```
class Program_Point:
        var        = ''   #variable name
        index      = -1   #index given to the program point by
                                #LLVM during instrumentation
        inv_type   = -1   #type of invariant (0 -store 1- load)
        vio_type   = ''   #type of violation (bm - bitmask r - range)
        violations = 0    #Number of times the invariant in this program point
                                #were violated
        C_d      = 0.0    #Cost of the diagnosis
        line       = ''   #line of code where it is
        distance   = -1   #distance from the fault
...
##### Gets all program points in the program #####
for i, line in enumerate(CONTEXT):
        if len(line.split(' ')) == 7:
                if int(word(line, 0)) == 1 and word(line, 6) !='-' and
                    word(line, 6).find('_addr') == -1:
                         pp = Program_Point()
                         pp.index = int(word(line, 2))
                         pp.inv_type= int(word(line, 1))

                         pp.line = int(word(line, 5))
                         pp.var  = word(line, 6)
                         PROGRAM_POINTS.append(pp)
```

Figure 5.7: Program points data structures and storage

Afterwards the script parses the *stderr* output file generated during the invariants testing. It registers, from this parsing, which program points' fault screeners have been violated and how many times.

```
...
for i, line in enumerate(STDERR):
        #checks if a new input is found
        if line.find('Writing datafile in') != -1 or i == len(STDERR) -1:
                #add the previous input to inputs list and resets the input
                if i > 0:
                        INPUTS.append(input_)
                        input_ = Input()
                        input_.pp_list = []
                        pp = Program_Point()

        #gets the pp's invariant type
        if line.find('invariant type index') != -1:
                #stores the previous program point
                if pp.index != -1:
                        if not input_.containsPP(pp):
                                input_.pp_list.append(pp)
                        pp = Program_Point()

                pp.inv_type = int(last_word(line))
        #gets the pp's index
        if line.find('invariant index') != -1:
                pp.index = int(last_word(line))
        #gets the pp's distance to the fault
        if line.find('distance') != -1:
                pp.distance = int(last_word(line))
                pos = pinpoint(pp,PROGRAM_POINTS)
                if pp.distance != -1 and pos != -1:
                        PROGRAM_POINTS[pos].distance = pp.distance

##### Counts the number of violations of each program point #####
for i, pp_ in enumerate(PROGRAM_POINTS):
        for j, inp in enumerate(INPUTS):
                for k, _pp_ in enumerate(inp.pp_list):
                        if pp_.isEqual(_pp_):
                                pp_.violations = pp_.violations + 1
...
```

Figure 5.8: Parsing *stderr* output

Finally, with the extracted information, the script is ready to generate an input file and feed it to *ochiai*, a fault diagnosis tool (part of Zoltar toolset [JAG09]) which is the responsible to produce mentioned list of likely fault locations. The referred input file consists in a matrix containing, per line, the program's spectra for each of that version's test cases. To this matrix's lines are appended +'s or -'s wether the program point was detected to have screener violations for that program run or not.

```
for i, pp_ in enumerate(PROGRAM_POINTS):
    isRankable = False
    matrix = []
    for j, inp in enumerate(INPUTS):
        if len(inp.pp_list) == 0:
                matrix.append(SPECTRA[j].strip('\n') + ' \t+\n')
        else:
                #The program point is on the list of violated program points
                #in one input it can be ranked
                if inp.containsPP(pp_):
                        matrix.append(SPECTRA[j].strip('\n') + ' \t-\n')
                        isRankable = True
                else:
                        matrix.append(SPECTRA[j].strip('\n') + ' \t+\n')
...
```

Figure 5.9: Assembling a spectra|pass/fail matrix

After the matrix is assembled the script runs *ochiai* and stores the resulting ranking file.

With the compiled information for each program point its respective cost of diagnosis can be calculated with ease. The calculation is made by using one of two formulas:

1. If the fault location is ranked in the ranking file the script applies the formula:

$$C_d = \frac{n}{M-1}$$

2. If the fault location not ranked in the ranking file the script applies the formula:

$$C_d = \frac{M_r + \frac{M-M_r}{2}}{M-1}$$

M_r: Number of likely fault locations ranked M: Number of spectrum components n: Position of the fault in the ranking file

The first formula is applied when the actual faulty statement of the program is listed within the *ochiai*-generated file, ranking the most likely locations for a fault to occur. In this case, to calculate the *cost of diagnosis* (C_d) we just divide the position of the fault

```
for i, pp_ in enumerate(PROGRAM_POINTS):
...
        #ranks the program point if possible
        if isRankable:
                rank = calculate_rank(M)
                open('%s_%s.rank' %(pp_.inv_type, pp_.index), 'w').write(rank)
...

#uses ochiai to rank the matrix and returns the result
def calculate_rank(M):
        matrix_file= open('matrix', 'w')
        matrix_file.writelines(matrix)
        matrix_file.close()
        cmd = '/path/to//MFL/bayes -m 4 -o %s matrix' % M
        sut = subprocess.Popen(cmd, shell = True, stdout= subprocess.PIPE)
        sutout = sut.communicate()
        return sutout[0]
```

Figure 5.10: Calculating likely fault locations

in the file by the total number of spectrum components of the program. This will always generate a C_d value below 0.5, meaning that less than 50% of the code needs to be inspected in order to find the fault.

The second formula is applied when the faulty statement of the program is not listed within the *ochiai*-generated file, ranking the most likely locations for a fault to occur. In this case the calculation is made by taking the C_d the program point would have if the fault was in last place in the file (value of *M_r*) and further adding to it the average number of spectrum components needed to inspect after it. So this formula will always generate C_d values above 0.5, meaning that more than 50% of the code needs to be inspected in order to find the fault.

```
...
##### Calculates the C_d of each program point #####
for i, rank_file in enumerate(RANKING_FILES):

        pp     = PROGRAM_POINTS[pinpoint_pp(PROGRAM_POINTS, rank_file)]
        rank   = open(rank_file.strip('\n'), 'r').readlines()
        pp.C_d = round(calculateC_d(rank, M, FAULT), 2)
...
#Calculates the C_d value based on a ranking, the number of
#components M and the fault
def calculateC_d(rank_lines, M, fault):

        fault_is_ranked = False
        for i, line in enumerate(rank_lines):
#if the fault is ranked calculates the program point's C_d directly
                if int(line.split(' ')[0])-1 == fault:
                        return 0.5 - float(i)/float(M-1)
#if the fault is not ranked calculates the program point's C_d using
#an obscure formula
        Mr = len(rank)
        return 0.5 - float((Mr + (M -Mr)/2))/float((M-1))
```

Figure 5.11: *cost of diagnosis* calculation

5.5.3 Number of violations

The calculation of the number of violations per program point is trivial. We have just counted the number of detected errors per program point when we have parsed the *stderr* file obtained from the invariant testing step of the approach (see section 5.3)

5.6 Results processors

We implemented two automated results processors. A script that takes the metrics data of a program's versions and averages it and a automatic plot generator, that reads data files and plots two different kinds of charts: (1) Program points' *cost of diagnosis* over the *distance to the fault* and (2) Program points *number of violations* over the *distance to the fault*. Both of the plot generators support either data per program and per version. The tool used to plot the data is gnuplot

```
#script lines
gplot = []
if MODE == 0:
        gplot.append('set title \"%s_%s\"\n' % (PROGRAM, VERSION))
else:
        gplot.append('set title \"%s\"\n' % (PROGRAM))
gplot.append('set xlabel \"distance to the fault\"\n')
gplot.append('set ylabel \"0.5-C_d\"\n')
gplot.append('set parametric\n')

if MODE == 0:
        gplot.append('set trange [-2:12000]\n')
        gplot.append('set xrange [-2:12000]\n')
        gplot.append('set yrange [-1:1]\n')
        gplot.append('set term gif\n')
        gplot.append('set output \'%s/%s_%s_cd.gif\'\n' % (PATH, PROGRAM, VERSION))
        gplot.append('plot %s %s, t, %s %s' % (DATA, L1, ORACLE_C_d, L2))
else:
        gplot.append('set trange [-2:1800]\n')
        gplot.append('set xrange [-2:1800]\n')
        gplot.append('set yrange [-1:1]\n')
        gplot.append('set term gif\n')
        gplot.append('set output \'%s/%s_cd.gif\'\n' % (AVG_PATH, PROGRAM))
        gplot.append('plot %s %s, t, %s %s ' % (AVG_DATA, L1, ORACLE_C_d, L2))

#writes and executes the script
open('gplot.p', 'w').writelines(gplot)
os.system('gnuplot > load \'gplot.p\'')
```

Figure 5.12: gnuplot charts' automatic generation

```
pp= Program_Point(inv_type, index, var, distance, violations, C_d, line_)
location = whereIs(pp, program_points)
if location != -1:
        program_points[location].distance_acc       =
                program_points[location].distance_acc + pp.distance
        program_points[location].distance_counter   =
                program_points[location].distance_counter + 1

        program_points[location].violations_acc     =
                program_points[location].violations_acc + pp.violations
        program_points[location].violations_counter =
                program_points[location].violations_counter + 1

        program_points[location].C_d_acc     =
                program_points[location].C_d_acc + pp.C_d
        program_points[location].C_d_counter =
                program_points[location].C_d_counter + 1

else:
        program_points.append(pp)

for i, pp in enumerate(program_points):
        if pp.violations_counter!= 0:
             pp.avgViolations =
             round(float(pp.violations_acc)/float(pp.violations_counter), 2)
        if pp.distance_counter != 0:
             pp.avgDistance =
             round(float(pp.distance_acc)/float(pp.distance_counter), 2)
        if pp.C_d_counter!= 0:
             pp.avgC_d =
             round(float(pp.C_d_acc)/float(pp.C_d_counter), 2)

        if pp.avgC_d > 0.4: program_points.remove(pp)
```

Figure 5.13: Automatic average calculation per program

5.7 Summary

The approach adopted in our study comprises 4 main steps: (1) Invariant instrumentation, (2) Invariant training and testing, (3) Program spectra generation and (4) Metrics calculation.

The invariant instrumentation consists in compile the benchmark's programs' versions' source code with LLVM and splitting it, by statement, into various program points. This program points are instrumented with Zoltar, thus creating load and store invariants.

The invariants are monitored and trained by two types of fault screeneres: (1) *Range* and (2) *Bit Mask*.

The invariants testing and training can be automated by first dividing the program's test cases into *correct* and *incorrect* test cases. This can be achieved by following the expected pass/fail reference results for all test cases. Then we run each correct test case during the invariants training phase in order to teach them which values they should accept. Afterwards we run the remaining test cases during the invariants testing phase and store the resulting *stderr* output.

The program spectra generation is obtained by applying the *gcov* tool to the program's test cases.

In order to calculate the *distance to the fault* we have tweaked Zoltar to count and return the number of spectra updates that occur after the fault. To calculate the *number of violations* of each program point we need to parse the *stderr* output file obtained during the invariant testing phase and count the number of times that each program point has been reported has containing screener violations. To calculate the *cost of diagnosis* of each program point we need to merge the program's spectra and the program point's execution results (pass or fail) for each test case and to feed the resulting matrix to *ochiai* diagnosis tool in order to get a ranked file with possible locations for the fault. Given this ranking, the metric's calculation becomes trivial.

For easier results analysis we have developed a script that averages the metrics of each program point per program, in order to have a more generic view of the results. We also implemented an automatic plot generator that can create charts presenting either the *cost of diagnosis* or the *number of violations* over the *distance* .

Chapter 6

Experimental Results

In this chapter we present and discuss the results obtained during our research. It will be divided in three section: (1) metrics definition, (2) results analysis and (3) a summary of the findings described in this chapter.

In the metrics definition section we define each metric that we have used to evaluate the results obtained in our study.

In the results analysis section present and discuss the obtained results program-wise. First we present first two charts, showing the both the cost of diagnosis and number of violations of each program point over the distance to the fault. Then we present a table that further details the data presented in the charts and helps to identify the principal program points and respective variables of each program. After that we study that variable's error detection capabilities' consistency throughout the program's version. Then, if needed for further clarity, we present relevant code sections of the program's source code and explain the behavior of each variable based on the results obtained.

6.1 Evaluation Metrics

In this section we define the evaluation metrics used to assess the results obtained during our research.

0.5-C_d For representation proposes instead of a direct representation of the cost of diagnosis value (C_d) we decided to adopt a 0.5-C_d metric so the values range from -0.5 to 0.5 instead from 0 to 1. This way the neutral C_d values of 0.5, that do not add any useful information, are converted to 0, preventing them to hinder the observation of the other, more important, values as it would happen if we used the standard C_d value. Also, this way the C_d values below 0.5, the most interesting ones, are the positive values on the chart so the bar for a program point's C_d is as high as its C_d is low (i.e: C_ds of 0 will be represented with a value of 0.5). The same is true for C_d values above 0.5.

Distance The metric often referred in this chapter as *distance* stands for the distance of a program point to the fault present in the correspondent program's version. This distance represents the number of the program's statements ran by the time the compiler reaches the statement over which the program point is instrumented. As such this value should be always an integer bigger or equal then zero. The only exception is when the program point which distance we want to measure is before the fault, in this cases the distance value will always be -1.

Number of violations The metric often referred in this chapter by *violations* or *number of violations* measures the number of times that a program point's screeners detect a value, in the variable they scan, that they have not being trained to accept (*screener violation*).

Although the *distance* and *number of violations* values are obviously integers, this values often appear as floats in the current chapter. This happens due to most of the metrics' data presented in this chapter is averaged from among all of each program's versions. Although it would be possible to round this averaged values to the closest integer, we would lose information if we did that. For example, program point's *distance* value between -1 and 0 means that in the majority of that program point's locations throughout versions is behind the fault but there is some versions in which the program point is after it. This kind of information can be useful for results analysis.

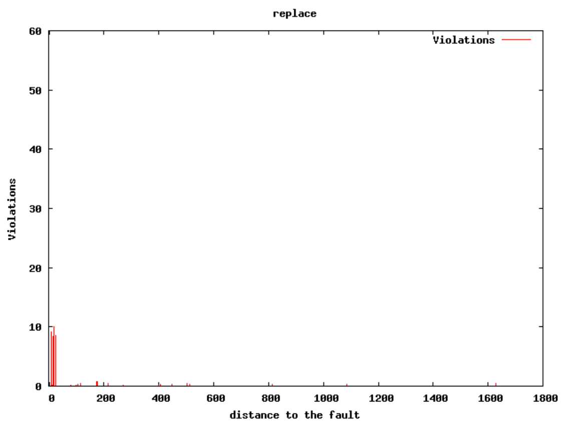
6.2 Results

In this section we analyze the most interesting variables amongst the Siemens Set' programs. The first program analysis is over *replace* program and it is the one described with deepest detail as it is the most data-intensive program of the programs set, excluding *tot_info*, and covers all the different behaviors found during this programs' set analysis. The remaining are described in a more summarized manner as the analysis method is roughly the same. This do not includes the *tot_info*'s analysis as its massive amount of screener violations require a slightly different analysis.

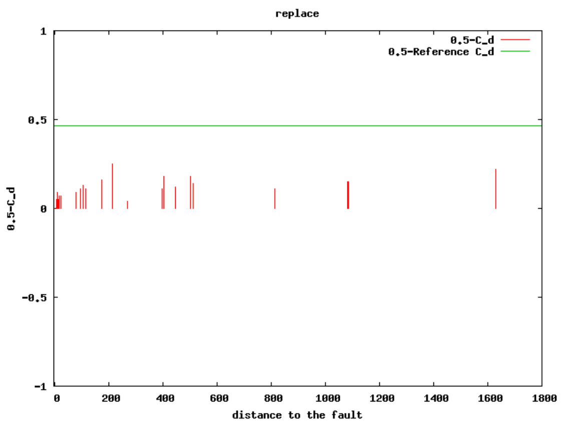
6.2.1 *replace*

The figure 6.1 presents the averaged values of the metrics obtained for each program point of *replace* throughout all of its versions. The chart depicted by figure 6.1a shows the number of violations detected by the program's screeners, located through all the code. This information is distributed according to the distance to the program's fault where that violations occur. The chart depicted by figure 6.1b presents the cost of diagnosis of each screened program point, meaning, the effort needed to find the failure based on that program point's information. This data is also distributed according to the distance that each program point is from the fault.

By analyzing picture 6.1 we can see that the number of program points with actual diagnosis value is reduced. Also, the mentioned program points C_d value is only median, as none of them comes close to the reference C_d value, represented in chart 6.1b as the horizontal green line. Another observation that can be made is that there is, in chart 6.1a, an apparent aggregation of program invariant violations near the fault, while as we go farther away from it their numbers decrease abruptly. Interestingly enough this do not seem to lower their program point's *0.5-C_d* values as expected.



(a) Average number of violations per program point



(b) Average cost of diagnosis of each program point

Figure 6.1: *replace*'s averaged metrics

The data presented in the table 6.1 shows the obtained *distance*, *number of violations* and *0.5-C_d* values of the program's variables reported as containing screener violations, as well as the number of the variable's violated program points. The maximum (Max), minimum (Min) and average (Avg) values displayed are based on the average values, through all *replace*'s versions, obtained for each of the variable's screened program points. Some variables in the table have the same name, in this cases they are indexed by a number within *()*. So, as we can see, only 12 of the entire program's variables have been reported, by its screeners, as containing errors. In theory, this means that just that 12 of *replace*'s variables need to be monitored in order to have an accurate error detection. However, both table 6.1 and figure 6.1 show that the variables' cost of diagnosis for this program are, at most, mediocre as its value ranges from 0.04 to 0.25. Thus its maximum value only represents 50% of the value obtained by using the reference program (*oracle*). The same reasoning is valid for the number of violations, as it remains very low, indicating a deficient test coverage and a consequent lost of diagnosis accuracy. The violations' location tend to be near the fault, although in 3 out of the 12 analyzed variables the violations are detected before the fault and in the variable *i(3)* there is a violation very far away from it.

	Distance			***Violations***			***0.5 - C_d***			***Program Points***
	Max	**Min**	**Avg**	**Max**	**Min**	**Avg**	**Max**	**Min**	**Avg**	
escjunk	-1	-1	-1	1.42	0.92	1.17	0.18	0.18	0.18	2
retval(1)	-1	-1	-1	0.2	0.18	0.19	0.1	0.09	0.95	2
makeres	-1	-1	-1	0.22	0.18	0.2	0.11	0.09	0.1	2
i(1)	-0.92	-1	-1	5.45	5.17	5.31	0.12	0.04	0.08	2
result	-0.91	-1	-0.95	0.18	0.18	0.18	0.09	0.09	0.09	2
i(2)	24.14	-0.91	7.685	9.42	8.57	8.76	0.08	0.05	0.07	4
m	17.3	7.25	12.28	10	8.33	9.17	0.05	0.04	0.05	2
size	17	11.43	14.22	0.17	0.14	0.16	0.08	0.07	0.08	2
retval(2)	17	12.45	14.73	0.18	0.14	0.16	0.09	0.07	0.08	2
lastm	14.25	14.25	14.25	8.33	8.33	8.33	0.04	0.04	0.04	1
k	175.25	115.5	154.96	0.75	0.5	0.67	0.12	0.10	0.11	3
i(3)	1084.33	213.5	550.70	0.5	0.22	0.32	0.25	0.11	0.15	9

Table 6.1: Averaged metrics per *replace*'s variable

The table 6.2 shows in which versions the variables' screeners are detecting violations. Although 26 out of the 32 versions of *replace* were instrumented for our study, only 12 of them actually present screener violations. The remaining are disregarded for this table analysis. The underlying cause for the apparently low values of *0.5-C_d* and *number of violations* obtained can be explained by the data in this table. Table 6.2 shows that, contrarily from what was expected, the variables' violations are not consistent throughout the program's versions. From the 12 variables of the program that we have identified has having some diagnosis potential, as much as 9 only have violation reports for two or less of the versions. Thus explaining why the averaged metrics' values in figure 6.1 and table 6.1 are so low. So in spite of the individual cost of diagnosis value of a variable usually be very near the *oracle*'s value for a single version, that value is not maintained through the program's versions. By analyzing the table by version, we can see that some variables are correlated, as they only have violation reports for versions 14 and 28. This indicates that they are generating screener violations solely because one of them have its value corrupted by this versions' particular faults' nature, triggering a chain effect that have also tainted the remaining. Thus this variables have absolutely no diagnosis potential whatsoever. The variables that present only violations for single version follow the same reasoning so they can also be disregarded. So this leaves us with only 2 possible *collar variables* for *replace* program: *escjunk* and *i(3)*.

	escjunk	***makeres***	***retval (1)***	***i (1)***	***result***	***lastm***	***retval (2)***	***size***	***m***	***i (2)***	***k***	***i (3)***
Version 3	x										x	
Version 4	x										x	
Version 5	x											x
Version 13	x					x			x	x		
Version 14		x	x	x	x		x	x				x
Version 15				x								
Version 18												x
Version 22												x
Version 26		x	x	x	x		x	x				
Version 28	x											
Version 30	x											
Version 31												x

Table 6.2: variable violations throughout *replace*'s versions

The *escjunk* variable is located before the fault and has a average *0.5-C_d* value of 0.18 (see table 6.1) although in its cost of diagnosis per version is comparable to the *oracle's*. Its screeners are able to detect errors in its value for 6 of the 26 instrumented program's versions. Figure 6.2 shows the code snippet where the variable is inserted. There are two program points that have screeners monitoring the variable: (1) a store statement in line 108 which depends of the *esc* function and its arguments and (2) a load statement in line 109 which depends on the statement on line 108. As neither *dodash* function nor *esc* are directly affected by any version's fault, the values of *escjunk* are entirely controlled by *src* and *i* variables. Thus discarding the possibility of *escjunk* being a *collar variable*. The reason for this variable to have violations before the fault is because its value is completely fault-independent.

```
93      dodash(delim, src, i, dest, j, maxset)
...
100     {
   ...
103         char        escjunk;
104
105         while ((src[*i] != delim) && (src[*i] != ENDSTR))
106         {
107             if (src[*i - 1] == ESCAPE) {
108                 escjunk = esc(src, i);
109                 junk = addstr(escjunk, dest, j, maxset);
110             }
        ...
```

Figure 6.2: *escjunk* location on the source code

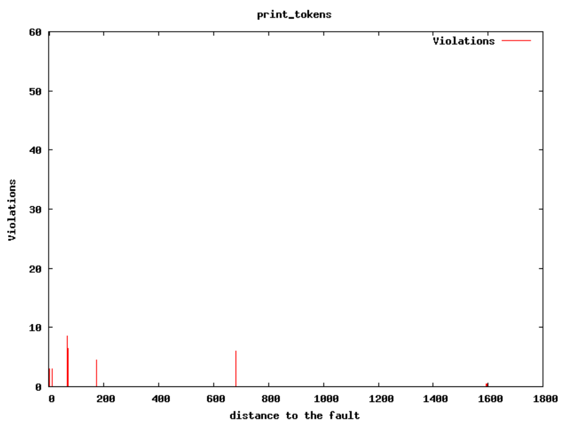
The variable *i(3)* is the variable with the highest *0.5-C_d* value of all *replace*'s variables. Its behavior is uncommon and worth to investigate, as its violations occur far away from the fault and there are significant *distance* gaps between them. This variable's violations data is represented by all the red bars in the [213.5; 1084.33] *distance* interval on figure 6.1. Statements containing an *i* variable in the code snippet depicted by figure 6.3 are actually the program points instrumented for *i(3)*. In the mentioned code snippet we can easily see that the value of *i* solely depends on the *sub* variable's value, this rules out the possibility of *i(3)* being a *collar varable*. Thus the mentioned behavior adopted by *i(3)* can also be explained by *sub* variable's value: As the distance to the fault is calculated based on the number of statements that the program's execution already ran, the variable is only being violated when it continues to be incremented over the maximum value accepted by its screener, which happens in average after 213 statements ran.

```
void
putsub(lin, s1, s2, sub)
...
{
...
    while ((sub[i] != ENDSTR)) {
        if ((sub[i] == DITTO))
            for (j = s1; j < s2; j++)
            {
        fputc(lin[j],stdout);
            }
        else
        {
            fputc(sub[i],stdout);
        }
        i = i + 1;
    }
}
```

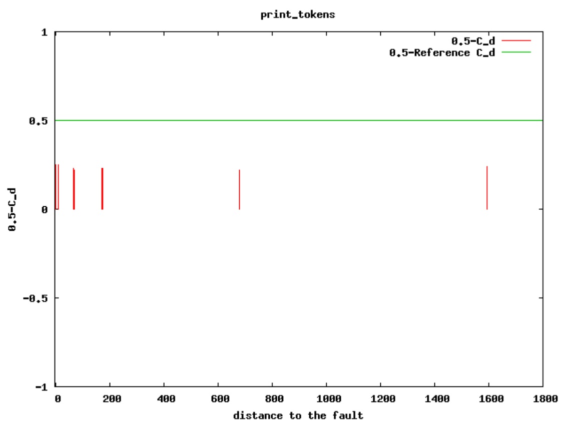
Figure 6.3: *i(3)* location on the source code

6.2.2 *print_tokens*, *print_tokens2*, *schedule*, *schedule2* and *tcas*

The figures 6.4, 6.5, 6.6, 6.7 and 6.8 presents the averaged values of the metrics obtained for each program point of *print_tokens*, *print_tokens2* and *schedule2* throughout all of their versions. The chart depicted by the left subfigures, of the mentioned figures, shows the number of violations detected by the program's screeners, located through all programs' code. This information is distributed according the distance to the program's fault where that violations occur. The chart depicted by the right subfigures presents the cost of diagnosis of each screened program point, meaning, the effort needed to find the failure based on that program point's information. This data is also distributed according to the distance that each program point is from the fault. By analyzing the mentioned charts we can see that the number of program points with actual diagnosis value is reduced. Also, the mentioned program points *0.5-C_d* value is fairly low, as none of them comes close to the reference *0.5-C_d* value, represented in the right subfigures as the horizontal green line.

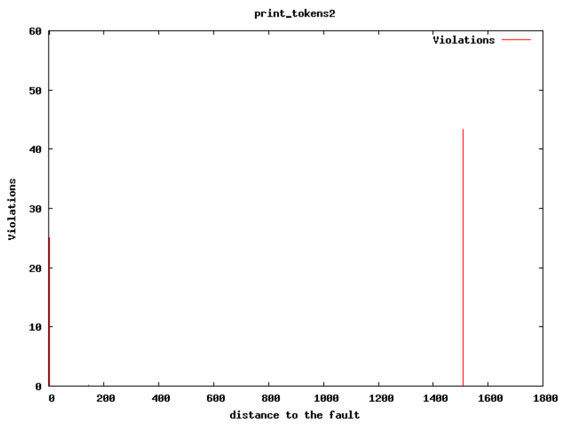


(a) Average number of violations per program point

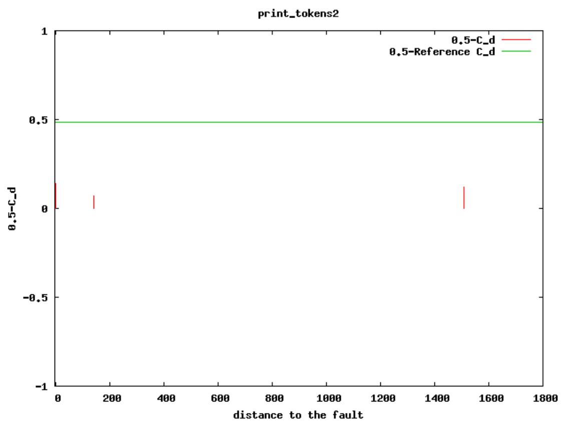


(b) Average cost of diagnosis of each program point

Figure 6.4: *print_tokens* averaged metrics

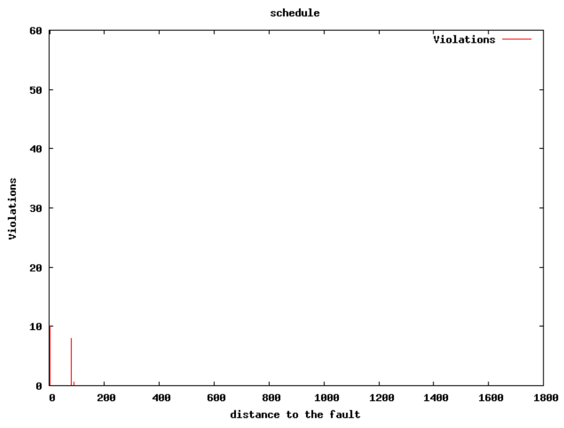


(a) Average number of violations per program point

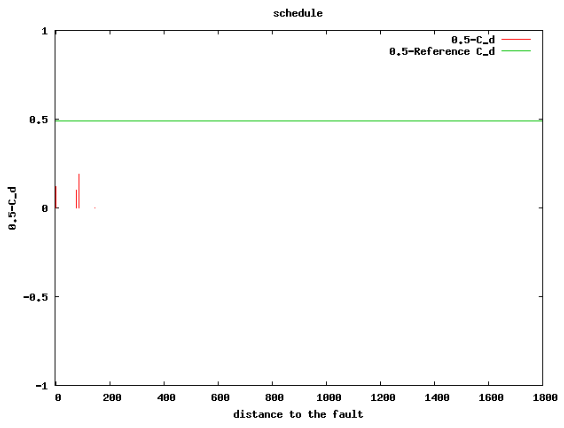


(b) Average cost of diagnosis of each program point

Figure 6.5: *print_tokens2* averaged metrics

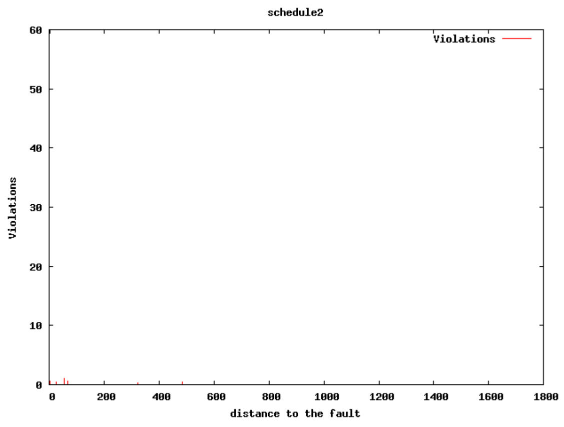


(a) Average number of violations per program point

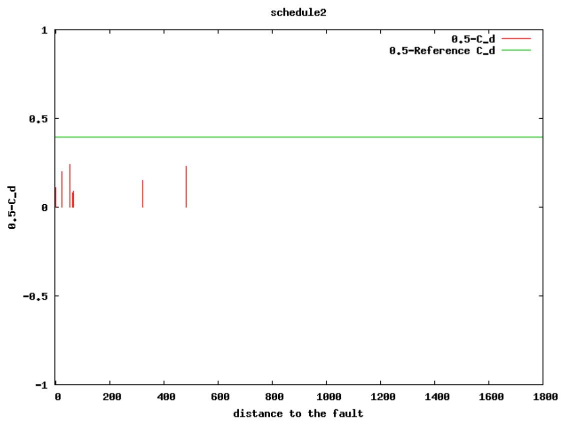


(b) Average cost of diagnosis of each program point

Figure 6.6: *schedule*'s averaged metrics

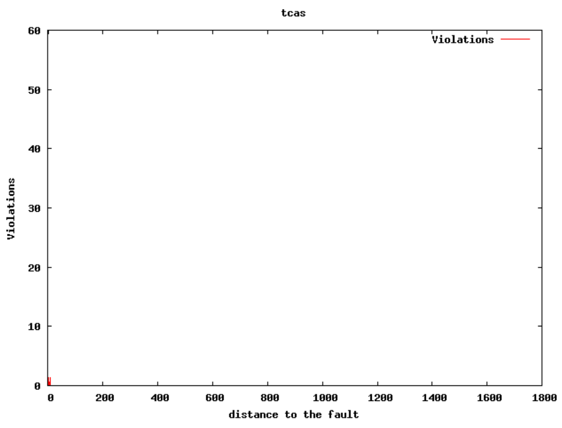


(a) Average number of violations per program point

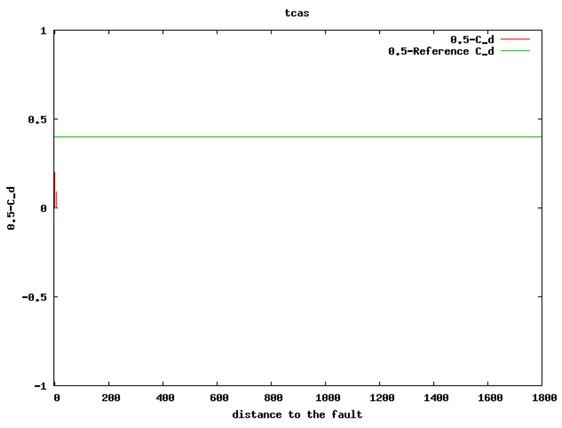


(b) Average cost of diagnosis of each program point

Figure 6.7: *schedule2*'s averaged metrics



(a) Average number of violations per program point



(b) Average cost of diagnosis of each program point

Figure 6.8: *tcas*' averaged metrics

The data presented in the tables 6.3, 6.4, 6.5, 6.6 and 6.7 shows the obtained *distance*, *number of violations* and *0.5-C_d* values of the programs' variables reported as containing screener violations, as well as the number of the variable's violated program points. So, as we can see, the number of important variables in all the 3 mentioned programs is indeed very low, as expected. However, figures 6.4, 6.5, 6.6, 6.7 and 6.8 and tables 6.3, 6.4, 6.5, 6.6 and 6.7 show that the variables' cost of diagnosis for the program are, at most, mediocre as its value never go further than 0.25, which only represents 50% of the value obtained by using the reference program (*oracle*).

	Distance			Violations			0.5 - C_d			Program Points
	Max	Min	Avg	Max	Min	Avg	Max	Min	Avg	
cu_state	12	2	6.67	3	3	3	0.25	0.25	0.25	3
ind	174	67.5	120.75	8.5	4.5	6	0.23	0.22	0.23	4
token_ind	1594.5	680	1365.88	6	0.5	1.88	0.24	0.22	0.23	4

Table 6.3: Averaged metrics per *print_tokens*' variable

	Distance			Violations			0.5 - C_d			Program Points
	Max	Min	Avg	Max	Min	Avg	Max	Min	Avg	
buffer	-1	-1	-1	23.67	23.67	23.67	0.08	0.08	0.08	1
ch	1510	1.57	551.38	43.29	0.14	22.81	0.14	0.07	0.12	3
fname	-1	-1	-1	0.57	0.14	0.33	0.21	0.07	0.12	3
fp	-1	-1	-1	0.14	0.14	0.14	0.06	0.06	0.06	3
i(1)	-0.25	-0.67	0.47	0.25	0.17	0.22	0.12	0.08	0.11	3
i(2)	-1	-1	-1	0.25	0.17	0.23	0.13	0.08	0.13	5

Table 6.4: Averaged metrics per *print_tokens2*'s variable

	Distance			Violations			0.5 - C_d			Program Points
	Max	Min	Avg	Max	Min	Avg	Max	Min	Avg	
n(1)	-1	-1	-1	0.4	0.2	0.3	0.18	0.09	0.14	2
n(2)	-1	-1	-1	1.2	0.4	0.8	0.28	0.09	0.19	2
src_queue	0.2	-1	-0.4	8	0.8	4.4	0.19	0.1	0.15	2
proc	78	-0.8	38.6	8	8	8	0.1	0.1	0.1	2
retval	0.25	0	0.13	10	8	9	0.12	0.1	0.11	2
count	87.4	87.4	87.4	0.6	0.6	0.6	0.19	0.19	0.19	1
ratio	1.6	1.6	1.6	0.4	0.4	0.4	0.1	0.1	0.1	1

Table 6.5: Averaged metrics per *schedule*'s variable

	Distance			Violations			0.5 - C_d			Program Points
	Max	Min	Avg	Max	Min	Avg	Max	Min	Avg	
next	-0.33	-0.67	-0.56	0.67	0.67	0.67	0.08	0.08	0.08	3
prio_queue	0.5	0	0.25	0.5	0.33	0.42	0.11	0.07	0.09	2
next_pid	482	321	397	0.5	0.33	0.42	0.25	0.15	0.19	2
buf	54	24.5	39.25	1	0.5	0.75	0.24	0.2	0.22	2
prio	66	64.67	65.335	0.67	0.67	0.67	0.08	0.09	0.09	2

Table 6.6: Averaged metrics per *schedule2*'s variable

	Distance			Violations			0.5 - C_d			Program Points
	Max	Min	Avg	Max	Min	Avg	Max	Min	Avg	
retval(1)	-1	-1	-1	0.2	0.2	0.2	0.02	0.02	0.02	1
retval(2)	-1	-1	-1	0.1	0.1	0.1	-0.02	-0.02	-0.02	1
Up_Separation(1)	-1	-1	-1	0.11	0.11	0.11	-0.02	-0.02	-0.02	1
Up_Separation(2)	2.89	2.89	2.89	0.67	0.67	0.67	0.2	0.2	0.2	1
iftmp.1	-1	-1	-1	0.11	0.11	0.11	-0.02	-0.02	-0.02	1
Positive_RA_Alt_Thresh	-1	-1	-1	0.2	0.2	0.2	-0.02	-0.02	-0.02	1
Alt_Layer_Value	-0.6	-0.6	-0.6	0.2	0.2	0.2	0.1	0.1	0.1	1
Other_Tracked_Alt	0.2	0.2	0.2	1.4	1.4	1.4	0.09	0.09	0.09	1

Table 6.7: Averaged metrics per *tcas*' variable

The tables 6.8, 6.9, 6.10, 6.11 and 6.15 shows in which versions the variables' screeners are detecting violations. Although we do not make use of the full set of programs' versions in our approach there are still a good percentage of the remaining versions on which there are not detected any violation. This versions are disregarded for the current tables' analysis. The underlying cause for the apparently low values of *0.5-C_d* and *number of violations* obtained can be explained by the data in these tables. They show that, contrarily from what was expected, the variables' violations are not consistent throughout the programs' versions. None of the programs have a variable for which it's screeners can systematically detect violations throughout all the programs versions. Actually most of the mentioned variables only have reported violations for two or less of their program's versions. And this happens in spite of the individual cost of diagnosis value of a variable usually be very near the *oracle*'s value for a single version. that value is not maintained through the program's versions.

	cu_state	***ind***	***token_ind***
Version 1	x		
Version 5		x	x

Table 6.8: variable violations throughout *print_tokens*'s versions

	buffer	***ch***	***fname***	***fp***	***i(1)***	***i(2)***
Version 2		x				
Version 3	x	x	x			
Version 4				x		
Version 5	x	x	x			
Version 6			x		x	x
Version 7			x			
Version 8		x				

Table 6.9: variable violations throughout *print_tokens2*'s versions

	n(1)	***n(2)***	***src_queue***	***proc***	***retval***	***count***	***ratio***
Version 1	x	x					x
Version 3						x	
Version 4						x	
Version 5	x	x	x	x	x		
Version 6	x	x					x

Table 6.10: variable violations throughout *schedule*'s versions

	next	*prio_queue*	*next_pid*	*buf*	*prio*
Version 1			x		
Version 2				x	
Version 8	x	x		x	x
Version 10					x

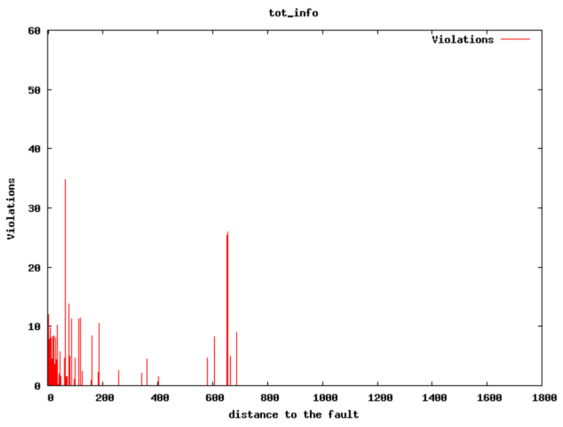
Table 6.11: variable violations throughout *schedule2*'s versions

	retval(1)	*retval(2)*	*Up_Separation(1)*	*Up_Separation(2)*	*iftmp.1*	*Positive_RA_Alt_Thresh*	*Alt_Layer_Value*	*Other_Tracked_Alt*
Version 1								x
Version 4							x	
Version 16								x
Version 28				x				
Version 29				x				
Version 35				x				
Version 36				x				
Version 38	x	x	x		x	x		
Version 40				x				
Version 41							x	

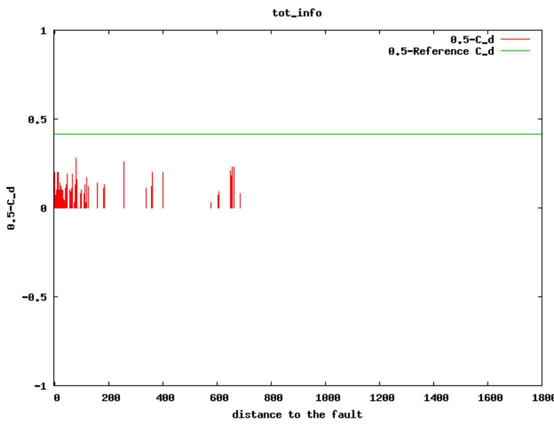
Table 6.12: variable violations throughout *tcas*'s versions

6.2.3 tot_info

The figure 6.9 presents the averaged values of the metrics obtained for each program point of the program throughout all of its versions. The chart depicted by figure 6.9a shows the number of violations detected by the program's screeners, located through all the code. This information is distributed according to the distance to the program's fault where that violations occur. The chart depicted by figure 6.9b presents the cost of diagnosis of each screened program point, meaning, the effort needed to find the failure based on that program point's information. This data is also distributed according to the distance that each program point is from the fault. As we inspect the charts, we notice that, although the number of instrumented program points is roughly the same as all the other programs', there are much more program points' screeners detecting violations. Also, the error coverage on each program point has increased, as we can see in chart 6.9a, since the number of violations per program point is much higher. Moreover, the program points' cost of diagnosis is also slightly better, as their *0.5-C_d* value is somewhat increased and the program's reference *0.5-C_d* value is lowered.



(a) Average number of violations per program point



(b) Average cost of diagnosis of each program point

Figure 6.9: *tot_info*'s averaged metrics

Due to the massive amount of program points reported to have violations, we analyze the data obtained by value intervals of both *distance* and *0.5-C_d* rather than by variable, as done in section 6.2.2. Table 6.13 shows just that.

Although there is a slight increase in the *0.5-C_d*, as observed in the chart depicted by figure 6.9, the table 6.13 shows that its biggest value concentration is still between 0 and 0.15 with 84.94% of the total program points reported to have violations which automatically renders them virtually useless for our study as their diagnosis value is too little to be considered. The remaining 15.66% are the most interesting ones, as their *0.5-C_d* values fall in the interval between 0.15 and 0.28, which are much more acceptable values.

Table 6.13 also show us that there is a clear aggregation of program points reported to have violations near the fault, as 77.5% of them is between 0 and 200 first program's executed statements. The remaining are quite disperse as their *distance* value varies between 200 and 600 without further significant gathering in a specific location . However there is another considerable aggregation in the *distance* interval between 600 and 700, this represents 7.03% of the considered program points.

Table 6.14 presents *tot_info*'s program points with the best cost of diagnosis. The numbers between *()* after each variable's name are meant to difference the program points within the same variable. The presented program points are illustrated in the chart depicted by figure 6.9b as: (1) the highest spike in the first group, between 0 and 200 distance, (2) the isolated, highest spike in [200 ; 400] distance interval and (3) the group in [600 ; 800]. By observing the mentioned table, we can see that the 3 chart points mentioned only correspond to 4 distinct variables. In fact, the first two pointed sections in the chart are actually generated by the same variable. Moreover that one variable (*totdf*) has the highest *0.5-C_d* values in the program, although its error coverage is very low. Both *ap* and *del* variables present exactly the same metrics, for all their program points,

	0.5-C_d				
	[0 ; 0.05]	**[0.05 ; 0.1]**	**[0.1 ; 0.15]**	**[0.15 ; 0.28]**	
[-1 : 0]	9.02%	2.21%	0%	0.6%	11.85%
[0; 1]	0%	7.63%	0%	0.6%	8.23%
[1;10]	8.84%	4.62%	0.60%	0%	14.06%
[10; 20]	0.6%	11.24%	2.21%	1.61%	15.66%
[20; 30]	5.42%	1.61%	0%	0%	7.03%
[30; 40]	3.01%	0.6%	1.2%	0%	4.82%
[40; 50]	0%	0.6%	0.6%	2.21%	3.41%
[50; 60]	0.6%	0%	0.6%	0%	1.2%
[60; 70]	1.81%	2.21%	1.2%	1.2%	6.43%
[70; 80]	0.6%	3.61%	1.2%	0.6%	6.02%
[80; 90]	0%	0.6%	0%	0.6%	1.20%
[90; 100]	0%	1%	0.6%	0%	1.61%
[100; 200]	0.6%	1.2%	5.42%	0.6%	7.83%
[200; 300]	0%	0%	0%	0.6%	0.60%
[300; 400]	0%	0%	1.2%	1.2%	2.41%
[400; 500]	0%	0%	0%	0.6%	0.60%
[500; 600]	0.6%	0%	0%	0%	0.60%
[600; 700]	0%	0%	1.81%	5.22%	7.03%
	31.12%	37.15%	16.67%	15.66%	**100%**

Table 6.13: Frequency table relating the *distance* to the fault with the *0.5-C_d*

and even between each other. The variable *sum* presents variations between its program points in all of its metrics. Its *distance* variations are minor, ranging from 651.59 to 664.35, their *0.5-C_d* values oscillate between 0.21 and 0.23 as its *distance* vary and its *number of violations* suddenly drops from around 25 to 5.

Variable	***Distance***	***Violations***	***0.5-C_d***
ap(1)	664.35	5	0.23
ap(2)	664.35	5	0.23
del	664.35	5	0.23
sum(1)	651.59	25.35	0.21
sum(2)	655.06	25.94	0.23
sum(3)	655.41	25.35	0.21
sum(4)	664.12	4.82	0.23
sum(6)	664.35	5	0.23
totdf(1)	255.29	2.59	0.26
totdf(2)	78.65	1.76	0.28

Table 6.14: *tot_info*'s program points with best cost of diagnosis

The table 6.15 shows in which versions the program points' screeners are detecting violations. Although we have instrumented 20 of the 23 program's version to use in this study, in 3 of those versions have not been detected any violations whatsoever, as such we disregard them for the current analysis. From the versions with violations, versions 1, 8 and 9 do not have reported violations of any of the program points in study. However all the mentioned program points have violations reported in versions 8, 11, 12, 15, 17 and 23, roughly a third of the studied versions. Both of the mentioned *(*ap)'s program points have violations exactly in the same versions as the *del*'s program points, which actually makes sense as in table 6.14 they have the exact same metrics value. Program points 3 through 5 of *sum* also only have violations in the exact same versions as *ap* and *del*. The *totdf* variable's program points are the ones which show the most consistency throughout the versions having violations reports in 11 and 12 out of the 17 versions in study.

	ap(1)	***ap(2)***	***del***	***sum(1)***	***sum(2)***	***sum(3)***	***sum(4)***	***sum(5)***	***totdf(1)***	***totdf(2)***
Version 1										
Version 2	x	x	x	x	x	x	x	x		x
Version 4	x	x	x	x	x	x	x	x	x	
Version 5	x	x	x			x	x	x		
Version 7									x	x
Version 8	x	x	x	x	x	x	x	x	x	x
Version 9										
Version 10										
Version 11	x	x	x	x	x	x	x	x	x	x
Version 12	x	x	x	x	x	x	x	x	x	x
Version 13									x	x
Version 15	x	x	x	x	x	x	x	x	x	x
Version 16									x	x
Version 17	x	x	x	x	x	x	x	x	x	x
Version 18									x	x
Version 20										x
Version 23	x	x	x	x	x	x	x	x	x	x

Table 6.15: program points' violations throughout *tot_info*'s versions

The code snippet depicted by figure 6.10 show the code section of *tot_info*'s source code in which *ap*'s, *del*'s and *sum*'s studied program points are. Both *del*'s and *sum*'s program points except sum(5) are instrumented over the statement in line 194. *ap(1)*, *ap(2)* and *sum(5)* are instrumented over the statement in line 198. As we can see, the studied variables' values are deeply connected, since the program point's statements are no more than load and store operations made over the 3 variables, making their value codependent. Also we can say that their value is, ultimately, fully dependent from the value of *a* variable, a function's argument as such ruling out the possibility of any of them being a *collar variable*. (see line 194 in figure 6.10) Thus explaining the fact that these 3 variables are reported to have errors in exactly the same program's versions, as presented in table 6.15. Following these reasoning we can also say that there are a false negatives both in *sum(1)* and *sum(2)* in version 5 as there is no other apparent reason for these program points to not have violations reports in that version.

The code snippet depicted by figure 6.10 show the code section of *tot_info*'s source code in which *totdf*'s studied program points are. Both *totdf(1)* and *totdf(2)* are instru-

```
184     gser( a, x )
185             double          a, x;
186             {
187             double          ap, del, sum;
...
194             del = sum = 1.0 / (ap = a);
195
196             for ( n = 1; n <= ITMAX; ++n )
197                     {
198                     sum += del *= x / ++ap;
199
...
```

Figure 6.10: *ap*, *del* and *sum* location on the source code

mented over the statement in line 106. The function presented in the code snippet is actually the *main()* function of the code and *totdf* is only used on it two times, in line 106 and in a *printf* instruction. However, its value is totally dependent from *infodf*, thus excluding the possibility of being a collar variable.

```
40      int
41      main( argc, argv )
        ...
99              if ( info >= 0.0 )
100                     {
101                     (void)printf( "2info = %5.2f\tdf = %2d\tq = %7.4f\n",
102                             info, infodf,
103                             QChiSq( info, infodf )
104                             );
105                     totinfo += info;
106                     totdf += infodf;
107                     }
..
```

Figure 6.11: *ap*, *del* and *sum* location on the source code

6.3 Summary

The results analysis that we perform in this chapter confirms that there are indeed very few variables that have some kind of diagnosis potential. Moreover these variables tend to aggregate near the fault, thus confirming that the location actually has some impact in the variables' diagnosis capabilities. In spite of this, there are still variables with error diagnosis potential that are located far away from the fault or before it. The first case occurs when the variable is caught in a loop that periodically triggers a screener violation, thus reporting the variable as containing errors at increasingly values of distance. The second case occurs mainly when the violations is triggered by an incorrect test case before the fault, most of the variables in which this happens tend to be somewhat fault-independent. Although we can verify that there is only a small set of variables per program in which errors can be detected and that these variables are generally near the fault, the actual values of cost of diagnosis found are much worse than expected. Additionally, the number of violations per program point is also very low, which indicates a deficient test coverage, further decreasing the variable's diagnosis quality. These behaviors are justifiable by the apparent variables' incapability of maintaining its diagnosis capabilities throughout their program's versions. Most of the variables that appear to have diagnosis capabilities, actually only have them for less than 20% percent of the program's version, which obviously renders them useless for our study. However there are some variables that can have their screeners to consistently detect violations for over 50% of the versions. Although this is not enough to consider them as possible *collar variables*, and most of them have been ruled out as such, there are strong evidences of false negatives in their screeners error detection. Thus indicating that even if we have not been able to find any *collar variables* in our study, that does not mean that they do not exist.

Chapter 7

Conclusions and Future Work

In Section 7.1, it is presented our contributions as well as a summary of our findings. Finally, in Section 7.2, we present the guidelines for the future work.

7.1 Conclusions

In this thesis, we have discussed an approach that have intended to answer the following academic question: "Is it possible to reduce the amount of program points that need to be screened in a program without losing diagnosis accuracy?". We have applied this approach to the Siemens Set benchmark. We have instrumented all the program points of the benchmark program's using Zoltar toolset and calculated the diagnosis potential of each one as well as its distance to the programs' fault. The results obtained confirm that there are indeed very few variables that have some kind of diagnosis potential. Moreover these variables tend to aggregate near the fault, also confirming that their location actually has some impact in the variables' diagnosis capabilities. In spite of this, the actual values of cost of diagnosis found are much higher than expected. Additionally, the number of violations per program point is also very low, which indicates a deficient test coverage, further decreasing the variable's diagnosis quality. These behaviors are justifiable by the apparent variables' incapability of maintaining its diagnosis capabilities throughout their program's versions. Moreover, all the candidates to collar variables found have been ruled out after a deeper analysis. Although there had been also found strong evidences of false negatives in their screeners error detection. Thus indicating that even if we have not been able to find any collar variables in our study, that does not mean that they do not exist.

In this thesis we make the following **contributions**:

- Empirical findings showing that a program point's diagnosis potential is affected by its proximity to the fault, indicating that there is *locality*
- Empirical evidence showing that most of a program's program points completely irrelevant for automatic error detection purposes.

- Empirical evidence exposing that it is not possible to find collar variables using only range and bit-mask fault screeners. As its false negative rate is too high

7.2 Future Work

Future work includes change the fault screeners type to bloom filters, which as a much lower false negative rate, and redo the experiments. If that also fails we will have to take a step back and leave the use of screeners to focus in the raw values of each variable in order to completely validate or invalidate this thesis crucial question.

References

[Abr09] Rui Abreu. *Spectrum-based Fault Localization in Embedded Software*. PhD thesis, Delft University of Technology, November 2009.

[Acr80] A.T. Acree. *On Mutations*. PhD thesis, School of Information and Computer Science, Georgia Institute of Technology, 1980.

[ACRL07] Piramanayagam Arumuga Nainar, Ting Chen, Jake Rosin, and Ben Liblit. Statistical debugging using compound boolean predicates. In Sebastian Elbaum, editor, *International Symposium on Software Testing and Analysis*, London, United Kingdom, July 9–12 2007. To appear.

[AZvG] Rui Abreu, Alberto González Peter Zoeteweij, and Arjan J.C. van Gemund. On the performance of fault screeners in software development and deployment. In Cesar Gonzalez-Perez and Stefan Jablonski, editors, *ENASE'08, Procedings of the 3rd International Conference on Evaluation of Novel Approaches to Software Engineering*.

[AZvG08] Rui Abreu, Alberto González Peter Zoeteweij, and Arjan J.C. van Gemund. Automatic software fault localization using generic program invariants. In *SAC'08, Procedings of the 23rd Annual ACM Symposium on Applied Computing - Software Engineering Track*, pages 712–717, Fortaleza, Ceará, Brazil, March 16 – 20 2008. ACM Press.

[Blo70] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[BPH08] George K. Baah, Andy Podgurski, and Mary Jean Harrold. 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, 2008. ACM.

[Bud80] T.A. Budd. *Mutation analysis of programs test data*. PhD thesis, Yale University, 1980.

[CB94] James M. Crawford and Andrew B. Baker. Experimental results on the application of satisfiability algorithms to scheduling problems. In *In Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 1092–1097, 1994.

[CKF$^+$02] Mike Y. Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, O Fox, and Eric Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *In Proc. 2002 Intl. Conf. on Dependable Systems and Networks*, pages 595–604, 2002.

[DB86] Thomas G. Dietterich and James S. Bennett. The test incorporation theory of problem solving (preliminary report). In *Oregon State Univ*, pages 145–161, 1986.

[DLZ05] Valentin Dallmeier, Christian Lindig, and Andreas Zeller. Lightweight defect localization for java. In *Proceedings of 19th European Conference on Object-Oriented Programming, ECOOP 2005*, number 3586 in Lecture Notes in Computer Science, pages 528–550. Springer, July 2005.

[ECGN01] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, 27:213–224, 2001.

[EPG$^+$07] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3):35–45, 2007.

[HFGO94] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. 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, 1994. IEEE Computer Society Press.

[HL02] Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection, 2002.

[HRS$^+$00] Mary Jean Harrold, Gregg Rothermel, Kent Sayre, Rui Wu, and Liu Yi. An empirical investigation of the relationship between spectra differences and regression faults. *Software Testing, Verification and Reliability*, 10:2000, 2000.

[JAG09] Tom Janssen, Rui Abreu, and Arjan J. C. van Gemund. 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, 2009. IEEE Computer Society.

[JH05] James A. Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 273–282, New York, NY, USA, 2005. ACM.

[JHS02] James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *In Proceedings of the 24th International Conference on Software Engineering*, pages 467–477, 2002.

[KJ97] Ron Kohavi and George H. John. Wrappers for feature subset selection. *ARTIFICIAL INTELLIGENCE*, 97(1):273–324, 1997.

[LA04] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[LFY$^+$06] Chao Liu, Long Fei, Xifeng Yan, Jiawei Han, Senior Member, and Samuel P. Midkiff. Statistical debugging: A hypothesis testing-based approach. *IEEE Transaction on Software Engineering*, 32:831–848, 2006.

[Lib08] Ben Liblit. 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, 2008. ACM.

[Liu06] Chao Liu. Failure proximity: A fault localization-based approach. In *In Proceedings of the International Symposium on the Foundations of Software Engineering*, pages 286–295, 2006.

[LMP08] Davide Lorenzoli, Leonardo Mariani, and Mauro Pezzè. Automatic generation of software behavioral models. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 501–510, New York, NY, USA, 2008. ACM.

[LNZ$^+$05] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. Scalable statistical bug isolation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 15–26, New York, NY, USA, 2005. ACM.

[MH03] Tim Menzies and Ying Hu. Computing practices data mining for very busy people, 2003.

[MJ96] Christoph C. Michael and Ryan C. Jones. On the uniformity of error propagation in software. In *In Proceedings of the 12th Annual Confererence on Computer Assurance (COMPASS '97*, pages 68–76, 1996.

[MOR07] Tim Menzies, David Owen, and Julian Richardson. The strangest thing about software. *Computer*, 40(1):54–60, 2007.

[MR95] Rajeev Motwani and Prabhakar Raghavan. *Randomized algorithms*. Cambridge University Press, New York, NY, USA, 1995.

[MR06] Tim Menzies and Julian Richardson. Making sense of requirements, sooner. *Computer*, 39(10):112–114, 2006.

[MW95] Aditya P. Mathur and W. Eric Wong. Reducing the cost of mutation testing: An empirical study. *The Journal of Systems and Software*, 31:185–196, 1995.

[ODC06] David Owen, Dejan Desovski, and Bojan Cukic. Effectively combining software verification strategies: Understanding different assumptions. *Software Reliability Engineering, International Symposium on*, 0:321–330, 2006.

[PBB$^+$02] David Patterson, Aaron Brown, Pete Broadwell, George Candea, Mike Chen, James Cutler, Patricia Enriquez, Armando Fox, Emre Kiciman, Matthew Merzbacher, David Oppenheimer, Naveen Sastry, William Tetzlaff, Jonathan Traupman, and Noah Treuhaft. Recovery oriented computing (roc): Motivation, definition, techniques,. Technical report, Berkeley, CA, USA, 2002.

[Pel04] Radek Pelánek. Typical structural properties of state spaces. In *In Proc. of SPIN Workshop, volume 2989 of LNCS*, pages 5–22. Springer, 2004.

[PKI05] Karthik Pattabiraman, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Application-based metrics for strategic placement of detectors. In *PRDC '05: Proceedings of the 11th Pacific Rim International Symposium on Dependable Computing*, pages 75–82, Washington, DC, USA, 2005. IEEE Computer Society.

[RCMM07] Paul Racunas, Kypros Constantinides, Srilatha Manne, and Shubhendu S. Mukherjee. Perturbation-based fault screening. In *HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 169–180, Washington, DC, USA, 2007. IEEE Computer Society.

[RR03] Manos Renieris and Steven P. Reiss. Fault localization with nearest neighbor queries, 2003.

[SÖ9] Hasan Sözer. *Architecting fault-tolerant software systems*. PhD thesis, Enschede, January 2009. IPA Dissertation 2009-05.

[SEES08] Friedrich Steimann, Thomas Eichstädt-Engelen, and Martin Schaaf. Towards raising the failure of unit tests to the level of compiler-reported errors. In *TOOLS (46)*, pages 60–79, 2008.

[WG03] Ryan Williams and Carla P. Gomes. Backdoors to typical case complexity. pages 1173–1178, 2003.

[WWQZ08] Eric Wong, Tingting Wei, Yu Qi, and Lei Zhao. 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, 2008. IEEE Computer Society.

[YPW08] Cemal Yilmaz, Amit Paradkar, and Clay Williams. Time will tell: fault localization using time spectra. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 81–90, New York, NY, USA, 2008. ACM.

[ZLN06] Alice X. Zheng, Ben Liblit, and Mayur Naik. Statistical debugging: simultaneous identification of multiple bugs. In *In ICML '06: Proceedings of the 23rd international conference on Machine learning*, pages 1105–1112. ACM Press, 2006.

Appendix A

Set up instructions

Here we explain how to install and use the tools employed in this thesis research.

A.1 Installing LLVM

The following shows the method we have used to download and install LLVM. First, we have downloaded the llvm version 2.6 sources and the llvm-gcc frontend sources. The latter is required to compile C programs using llvm. Both have been downloaded here. Next, we have unpacked both using

```
# tar -xzf llvm-2.6.tar.gz
# tar -xzf llvm-gcc-4.2-2.6.source.tar.gz
```

We have made an object directory and an install directory for llvm:

```
# mkdir llvm-obj
# mkdir llvm-install
Installed llvm:
# cd llvm-obj
# ../llvm-2.6/configure
# make
```

At this point, an LLVM version without gcc-frontend have been built. To make the gcc-frontend, however, the following installation of llvm is needed. We have made an object directory and an install directory for llvm-gcc as well:

```
# cd ..
# mkdir llvm-gcc-obj
# mkdir llvm-gcc-install
```

and configured it to use the llvm we have just installed.

```
# cd llvm-gcc-obj
# ../llvm-gcc4.2-2.6.source/configure \
> --prefix=‘pwd‘/../llvm-gcc-install --program-prefix=llvm- \
> --enable-llvm=‘pwd‘/../llvm-obj --enable-languages=c,c++
# make
# make install
```

After this, llvm has been reconfigured to include llvm-gcc.

```
# cd ../llvm-obj
# ../llvm-2.6/configure --prefix=‘pwd‘/../llvm-install \
> --withllvmgccdir=‘pwd‘/../llvm-gcc-install
# make
# make install
```

The above procedure created two installation directories, llvm-install and llvm-gcc-install. The bin directories of both have been added to the PATH environment, enabling a system wide call to the llvm tools.

A.2 Installing Zoltar

The following shows the method we have used to download and install the Zoltar tool set. First we have downloaded the source package here. Then we have unpacked it with:

```
# tar -xzf zoltar.tar.gz
```

Inside the unpacked directory we have ran

```
# ./configure --with-llvmsrc=<LLVMSRCDIR> \
> --with-llvmobj=<LLVMOBJDIR> --prefix=<INSTALLDIR>
```

where <LLVMSRCDIR> and <LLVMOBJDIR> are the directories containing the llvm source files and llvm object files respectively (/path/to/llvm-2.6 and /path/to/llvm-obj in the previous section). The tool binaries and libraries have been located in <INSTALLDIR>. After that we have built and installed the tool set using:

```
# make
# make install
```

At this point the Zoltar tools have been installed in the bin directory of <INSTALLDIR> and the libraries have been installed in the lib directory. Zoltar’s bin directory was added to the PATH environment variable and lib directory was included the library search path.