Birgit Hofer

From Fault Localization of Programs written in 3rd Level Languages to Spreadsheets

Doctoral Thesis

Graz University of Technology

Institute for Softwaretechnology

Supervisor: Prof. Dr. Franz Wotawa

Graz, July 2013

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Graz,	______________	______________
	Date	Signature

Eidesstattliche Erklärung[1]

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Graz, am	______________	______________
	Datum	Unterschrift

[1]Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008; Genehmigung des Senates am 1.12.2008

Abstract

Software debugging is a process that is rarely automated. Most programmers debug their programs manually or semi-automatically by using tools for setting break points. Therefore, software debugging is a very time-consuming and consequently expensive task. There exist automated debugging approaches, but they are rarely used in practice. The reasons for not using such approaches are their low maturity and lack of debugging quality. There is a strong need for improving existing debugging techniques so that they are accepted by programmers.

In this thesis, we address the first step of the debugging process, i.e. fault localization. We introduce two approaches to improve fault localization of programs written in imperative or object-oriented languages (i.e. 3rd generation languages): Sendys and Conbas. Conbas is an approach that aims to reduce the size of slices by means of constraint solving. Conbas can be used as a filtering step in other debugging approaches. We empirically show that Conbas reduces the number of potential faulty statements by 28 % for the single fault and 50 % for the double fault case compared to dynamic slicing. Due to its computational complexity, Conbas is intended for debugging small programs only. In contrast, Sendys is able to handle large programs. Sendys is an approach that combines spectrum-based fault localization (Sfl) with slicing-hitting-set computation (Shsc), a lightweight model-based software debugging (Mbsd) approach. This combination eliminates some of the disadvantages that exist when using the previously mentioned approaches separately. In an empirical evaluation, we show that Sendys reduces the number of statements a programmer needs to check manually. In particular, we gain reductions of about 50 % for Shsc and 25 % for Sfl.

This thesis not only focuses on the fault localization of programs written in 3rd generation languages but also on debugging support for end-user programs, in particular spreadsheets. For this purpose, debugging approaches for 3rd generation languages, e.g. Mbsd, Sfl and Sendys, are adapted to the spreadsheet domain. The empirical evaluation shows that Mbsd, Sfl and Sendys are promising techniques for debugging spreadsheets. These techniques produce short lists of possibly faulty cells which leads to shorter debugging times.

Kurzfassung

Die Fehlerbehebung in Software ist ein Prozess, der kaum automatisiert ist. Die Mehrheit der Programmierer beseitigt Fehler manuell oder halbautomatisiert mit Hilfe von 'Break points'-Werkzeugen. Deswegen ist Debugging eine zeitintensive und folglich auch kostenintensive Aufgabe. Es gibt viele automatisierte Debugging-Techniken. Diese Techniken werden in der Praxis kaum verwendet, da sie meist unausgereift sind und die Debugging-Qualität nicht ausreichend ist. Deswegen besteht Bedarf die existierenden Debugging-Techniken zu verbessern.

Diese Doktorarbeit beschäftigt sich mit dem ersten Schritt im Debugging-Prozess, der Fehlerlokalisierung. Wir stellen zwei Ansätze vor, um die Fehlerlokalisierung von Programmen zu verbessern, die in imperativen oder objektorientierten Sprachen (d.h. Sprachen der dritten Generation) geschrieben wurden: Sendys und Conbas. Conbas reduziert Slices mit Hilfe von Constraint Solving und kann als Vorverarbeitungsschritt verwendet werden, um den Suchraum zu minimieren. Eine empirische Studie zeigt, dass Conbas die Anzahl der zu betrachtenden Programmzeilen im Vergleich zu dynamischen Slices um 28 % für Einfachfehler und 50 % für Zweifachfehler reduziert. Aufgrund seiner rechnerischen Komplexität kann Conbas nur auf kleine Programme angewendet werden. Im Gegensatz dazu kann Sendys auch auf größere Programme angewendet werden. Sendys verbindet die Spektrum-basierte Fehlerlokalisierung (Sfl) mit der Slicing-Hitting-Set Methode (Shsc), einer einfachen model-basierten Software Debugging Technik (Mbsd). Diese Verbindung eliminiert einige der Nachteile von Sfl und Shsc. Eine empirische Studie zeigt, dass Sendys die Anzahl der zu betrachtenden Quellcode-Zeilen um 50 % verglichen mit Shsc und um 25 % verglichen mit Sfl reduziert.

Diese Doktorarbeit beschäftigt sich nicht nur mit der Fehlerlokalisierung in Programmen in Sprachen der dritten Generation, sondern auch mit sogenannten End-Nutzer Programmen, im Besonderen mit Tabellenkalkulationsprogrammen. Hierfür werden die Debugging-Ansätze Mbsd, Sfl and Sendys an die Tabellenkalkulationsprogram-Domäne angepasst. Eine empirische Studie zeigt, dass Mbsd, Sfl und Sendys vielversprechende Ansätze für das Debugging von Tabellenkalkulationsprogrammen sind. Die Ansätze liefern eine kurze Aufstellung von möglicherweise fehlerhaften Zellen. Dadurch wird die Zeit, die für das Debugging benötigt wird, reduziert.

Acknowledgements

I am particularly thankful for the advice and support from my supervisor Prof. Franz Wotawa. He showed me how to conduct research and let me develop and follow my own ideas. In addition, I want to thank all of my co-authors, in particular Prof. Rui Abreu, Elisabeth Getzner, and Simon Außerlechner.

During my research and teaching at Graz University of Technology, I had wonderful teaching assistants who helped me to manage classes. Their efforts allowed me to conduct my research alongside teaching. Therefore, I want to thank Elisabeth Getzner, Patrick Koch, Philip Kohler, Phillip Taferner, Roxane Roitz, Georg Hinteregger and Sandra Fruhmann. In addition, I thank all colleagues at the Institute for Software Technology, especially Petra Pichler.

Last but not least, I want to thank my parents, Edmund and Martha Hofer, and my sister Anita Teschl with family for their support. I am very thankful for the support and understanding of my fiancé Matthias Straka while I was writing my thesis. In addition, he helped me in proof-reading the thesis.

Contents

List of Figures

List of Tables

List of Algorithms

Listings

List of Acronyms

CONBAS	Constraint-Based Slicing
CONBUG	Constraint-Based Debugging
CSP	Constraint Satisfaction Problem
IDE	Integrated Development Environment
LOC	Lines Of Code
MBD	Model-Based Debugging
MBSD	Model-Based Software Debugging
MUSSCO	Mutation Supported Spreadsheet Correction
NCSS	Non Commenting Source Statements
SENDYS	Spectrum-Enhanced Dynamic Slicing
SFL	Spectrum-Based Fault Localization
SHSC	Slicing-Hitting-Set-Computation
SMT	Satisfiability Modulo Theories
SSA	Static Single Assignment
SUT	System Under Test

List of Symbols

AB(C)	Boolean variable representing the health state of component C
B	Boundary value (e.g. for the maximal size of leading diagnoses)
CELLS	Set of cells
CON	Set of constraints
CV(t)	Conflicting variables for test case t
DDG	Data dependence graph
Δ	Diagnosis
Δ^S	Set of minimal diagnoses
ET	Execution trace
ETG	Execution trace graph
$\ell(c)$	Formula of the cell c
LD	Leading diagnoses
M	Number of test cases ($\lvert T \rvert$)
N	Number of statements in a program ($\lvert \Pi \rvert$)
$\nu(c)$	Value of the cell c
$\mathcal{L}$	Programming/spreadsheet language
ω	A concrete program state or variable environment
Π	Program/spreadsheet
S	Relevant slice
Σ	Set of all program states
SD	System description
t	Test case
T	Test suite
$[\![\cdot]\!]$	Interpretation function

Part I.

Introduction

1. Introduction

Every year, the requirements on software increase. The trends in software engineering are towards increasing functionality, increasing flexibility and decreasing the time-to-market. However, these trends come with a price: Because of the high market competitiveness and the pressure to be the first on market, the testing and debugging stay behind. Many faults remain undetected in the pre-release phase because of superficial testing. Undetected faults can result in many facets: from simple unintended behavior over drastic financial loses (e.g. the explosion of the Ariane 5 [Dow97] causing 370 million dollars damage) to life-risking consequences (e.g. overdoses in radiation[1] killing eight people).

In practice, there exist many automated software testing tools that allow to expose a huge amount of errors in software. Unfortunately, programmer are often not able to correct all the bugs reported by the testers. This is a consequence of the fact that the next steps after error detection, i.e., fault localization and correction, are rarely automated. Therefore, not all of the discovered errors can be corrected in an acceptable amount of time. Often, programmers can only correct the most critical bugs. As a consequence, debugging has been identified as a bottleneck for improving reliability [AZG06].

Debugging has a long history. The first bug was reported by Grace Hopper in 1947 and was a moth that was taped into a logbook [Bil89]. This logbook entry lead to the name debugging. Traditionally, debugging was done manually by adding print statements to the source code. Print statements allow the programmer to check if certain statements are reached during the execution and to determine the value of a variable at a certain point in the program. Nowadays, debugging tools improve the manual debugging approach: Breakpoints and step-by-step execution allow the observation of program states. However, this step-by-step execution is very time consuming because a programmer has to manually narrow down the search space by introducing break points and comparing intermediate variable values of the program with the expectations. Moreover, setting the right break points or obtaining the right information from a program in order to speed up the debugging process, is a hard task. Even worse, in many cases a programmer

[1] `http://www.wired.com/software/coolapps/news/2005/11/69355?currentPage=2`

does not fully understand a program and can hardly find such optimal decisions. Consequently, there is a strong need for tools guiding the programmer through the fault localization process. Such tools should point to source code locations where the fault is most likely to be found.

On the academic side, automated debugging techniques and prototypes are available which help to narrow down possible fault locations. However, these approaches are hardly used by programmers. The main reason for the refusal to use these techniques is that they are not mature. In addition, debugging tools do not smoothly integrate in the available IDEs. Furthermore, these debugging tools fail to identify unique root causes and still leave a lot of work for the programmer. Moreover, the techniques applied in debugging tools can also be computationally demanding, which prevents them from being used for large-scaled programs and in an interactive manner.

The goal of this thesis is to improve the state of the art in fault localization by minimizing the number of user interactions when searching for the fault location. We are confident that only a combination of debugging techniques can be used for real world problems. Therefore, this thesis aims at combining existing debugging techniques in order to concentrate their advantages and eliminate their disadvantages. In doing so, we pursue the goal of fault localization by two different strategies: (1) We rank statements according to a value stating the likelihood of being faulty. When assuming that a user will start investigating the highest ranked statements first, the ranking indicates the required user interactions. (2) We provide the programmer a set of possible explanations. Only statements contained in this set can explain an observed misbehavior. Statements that are missing in this set can be excluded from the manual fault localization.

In addition to improving fault localization, this thesis aims to enlarge the field of application for fault localization. A possible domain for this enlargement are spreadsheets. Spreadsheets are by far the most successful example of end-user programming. End-user programmers vastly outnumber professional ones. Panko and Port [PP12] state that spreadsheets are used in more than 95 % of the U.S. companies for a variety of purposes, e.g. for financial reporting, forecasting, planning, data management and chart creation. Furthermore, companies often use dozens or even hundreds of spreadsheet applications which often tend to be large and complex comprising hundreds or even thousands of formulas.

Considering that important decisions are often based on spreadsheets, it is desirable that spreadsheets are free from errors. Unfortunately, numerous studies have shown that existing spreadsheets contain redundancy and errors at an alarmingly high rate [CKR01]. Panko published the results of field audits for spreadsheets on a website[2]: On average, 88 % of the spreadsheets investigated between 1995 and 2007 contained at least one error. However, when basing business decisions on spreadsheets, such errors can lead to a significant financial loss or to other business risks as reported on the web site of the European Spreadsheet Risk Interest Group[3].

Most spreadsheets are created for single-use only. Consequently, maintainability and scalability issues are not considered in these spreadsheets. However, many of these single-use spreadsheets are still used after several years and spread to other employees. Furthermore, spreadsheets lack support for abstraction, testing, encapsulation, and structured programming.

Therefore, debugging spreadsheets can be a time-intensive and frustrating task. Surprisingly, only few techniques have been borrowed from the software engineering domain. However, many techniques and concepts from the software engineering discipline can help to avoid faults, to increase maintainability and to lower the debugging time in spreadsheets. One of the objectives of this thesis is to adapt fault localization techniques from the software engineering domain (further on referred as fault localization techniques for 3rd generation languages) to the spreadsheet domain. Our goal is to provide the spreadsheet developer with tools that support him or her in identifying the formulas that are responsible for an observed misbehavior.

1.1. Concepts and definitions

This thesis makes use of the following nomenclature:

Fault	A fault (or defect) is the root cause for an observed misbehavior. In this thesis, a fault is a particular location in the program or spreadsheet, i.e a statement or a cell.
Failure	A failure is the inability of the system to perform as required.
Error	An error is the discrepancy between computed and specified (or assumed) values. Another term for error is 'observed misbehavior'.
Error detection	Errors can be detected during testing or when using software. The detection of errors is a prerequisite for triggering the debugging process.

[2] `http://panko.shidler.hawaii.edu/ssr/`
[3] `http://www.eusprig.org/horror-stories.htm`

Debugging Debugging is the process of fault localization and/or fault correction.

Fault localization Fault localization is the process of pinpointing the fault that leads to failures and errors.

1.2. Contributions

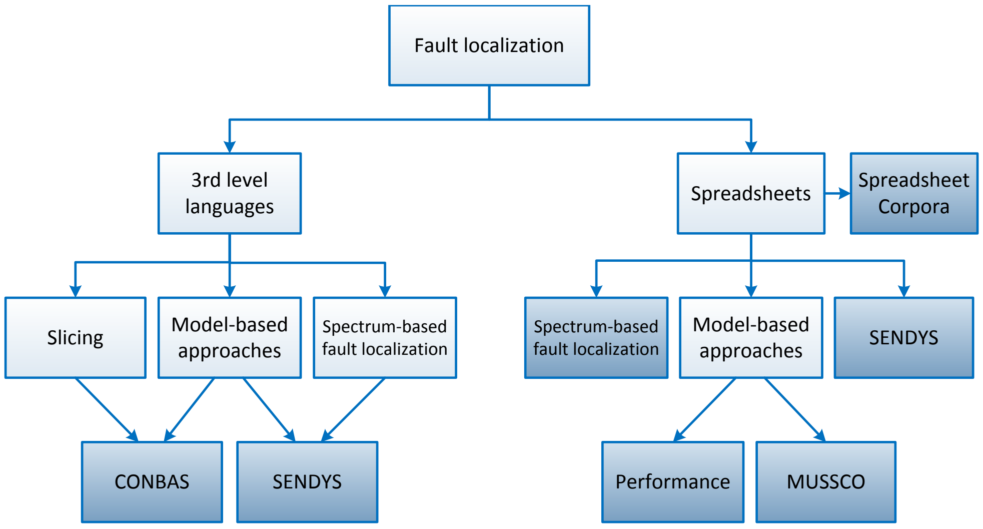


Figure 1.1.: Overview of the thesis contribution

This thesis improves the state of the art in fault localization in both, 3rd generation languages and the spreadsheet domain. Figure 1.1 illustrates how the techniques and other contributions of this thesis fit into the current field of fault localization. The main contributions for fault localization for 3rd generation languages are as follows:

- As a central contribution, we present Sendys (short for Spectrum-Enhanced Dynamic Slicing). Sendys is a lightweight model-based debugging approach enhanced with spectra information. This technique is intended to be used for debugging large-scaled programs.
- Another important contribution is Conbas (short for Constraint-Based Slicing). Conbas is a technique that reduces the size of slices by means of constraint solving. In contrast to Sendys, Conbas is indented to be used for small programs only.

In the field of fault localization for spreadsheets, this thesis makes the following contributions:

- We show how traditional debugging techniques can be used for debugging spreadsheets.
- In the case of model-based software debugging, we show that we can improve runtime performance when using a state-of-the-art SMT solver instead of a constraint solver.
- Furthermore, this thesis describes how to correct a faulty spreadsheet with mutations and distinguishing test cases. The technique for spreadsheet fault localization by generating repair suggestions is called MuSSCO (short for Mutation-Supported Spreadsheet Correction).
- In addition, we created spreadsheet corpora containing both correct and faulty versions of spreadsheets. These corpora can used for benchmarking.

1.3. Outline

Part II and Part III deal with the fault localization in 3rd generation languages and spreadsheets, respectively. Each of these parts discusses related work (Sections 2 and 6) and preliminaries (Sections 3 and 7). Spectrum-Enhanced Dynamic Slicing (Sendys) and Constraint-Based Slicing (Conbas) for the fault localization for 3rd generation languages are discussed in the Sections 4 and 5. Section 8 discusses the spreadsheet corpora that are used to evaluate the developed spreadsheet debugging techniques. Section 9 adapts approaches for fault localization in 3rd generation languages to spreadsheets. Section 10 introduces ConBug, a model-based debugging approach for spreadsheets. Section 11 explains how this model-based debugging approach can be improved w.r.t. runtime and variable domains. Section 12 introduces the MuSSCO approach which not only localizes faults but also computes repair suggestions. In Part IV, deficiencies, open challenges and future work for both domains, i.e. 3rd generation languages and spreadsheets, are discussed.

Part II.

Debugging of programs written in 3rd generation languages

2. Software debugging techniques

Software debugging techniques can be divided into fault localization and fault correction techniques. Fault localization techniques focus on narrowing down possible fault locations. They comprise spectrum-based fault localization, delta debugging, program slicing, and model-based software debugging. Fault correction techniques focus on finding solutions to eliminate an observed misbehavior. They comprise for instance genetic programming techniques. In the following, we discuss these automatic software debugging techniques in detail.

There exist alternative classification schemes for debugging techniques. For example, Seviora [Sev87] classifies debugging approaches into static and dynamic strategies. Static debugging strategies use the source code as knowledge basis. The most prominent static debugging approach is static slicing. Dynamic approaches use execution traces as knowledge basis. Examples for dynamic debugging approaches are dynamic slicing and spectrum-based fault localization. As there exist techniques that use both, the source code and the execution trace, we refrain from using this classification method. For example, there exist model-based software debugging techniques that work on the source code or on the execution trace.

Ducassé [Duc93] distinguishes tutoring systems and diagnosis systems. Tutoring systems are used to debug programs written by novices. The programs debugged with tutoring systems are usually small, the specification is given and possible errors are often known in advance. In contrast, diagnosis systems help to locate bugs in programs written by experts. These programs are generally large and a specification is seldom available. The debugging techniques presented below fit into the category diagnosis systems.

Furthermore, Ducassé [Duc93] defines three categories for automated debugging techniques: (1) verification with respect to specification, (2) checking with respect to language knowledge, and (3) filtering with respect to symptoms. The verification strategy compares a program with its specification. Verification techniques do not locate errors, but only indicate whether there exist any errors. One of the most prominent verification techniques is symbol execution. Checking techniques systematically parse programs and search for language dependent errors and code patterns which might lead to errors ("code smells"). For example, FindBugs [Aye+07] analyzes Java Code by means of about 300 fault patterns. The checking strategy detects only stereotyped errors. The filtering strategy identifies parts of the code that cannot be responsible for an observed misbehavior. The aim of filtering techniques is not to point at suspicious code, but to reduce the amount of code that must be investigated. Well-known filtering techniques are program slicing and model-based software debugging. Delta debugging can be seen as a special case of a filtering techniques since it reduces the failure inducing input and therefore implicitly the execution trace. The verification strategy and the checking strategy rather focus on error detection than on fault localization. Therefore, these strategies are not included in the following discussion of related work. Instead, we focus on filtering techniques and ranking techniques. One of the most prominent ranking technique is spectrum-based fault localization. Unfortunately, ranking techniques are were not taken into consideration by Ducassé.

2.1. Slicing

Program slicing narrows down the search space for fault locations by considering only the statements which directly or indirectly influence the values of a given set of variables at a certain program point. The basic idea of slicing is to reason backwards: Start from the failure and use the control and data flow of the program in the backward direction in order to reach the faulty statement. There exists several variants and enhancements of slicing.

In 1982, Weiser [Wei82] introduced the concept of static slicing. He defined a slice as a subset of program statements that behaves like the original program for a given set of variables at a given location in the program. Weiser's slicing method relies on static program analysis and does not take into account the input that leads to the failure. A slice is minimal if no statement of the slice can be reduced so that the reduced slice behaves like the original program for the given set of variables. Computing minimal static slices is equivalent to the halting problem [Wei84] and therefore undecidable. However, there exist good approximations to compute slices, e.g. the use of system dependency graphs [HRB88].

Static slices tend to be rather large. Subsequently, a programmer has to look at numerous statements in order to reach the root cause of the detected misbehavior. Therefore, Korel *et al.* [KL88] introduced the concept of dynamic slicing which relies on a concrete program execution. Dynamic slices behave like the original program only for a given test case. Since dynamic slices are more restrictive, they yield smaller slices than their static counterpart. Occasionally, dynamic slices do not include statements which are responsible for a fault if the fault causes the non-execution of some parts of a program. This disadvantage is eliminated by the usage of relevant slicing [GBF99].

A mentionable alternative to relevant slicing is the method published by Zhang *et al.* [Zha+07]. This method introduces the concept of implicit dependencies. Implicit dependencies are obtained by predicate switching. They are the analog to potential data dependencies in relevant slicing. The obtained slices are smaller since the use of implicit dependencies avoids a large number of potential dependencies.

Another technique by Zhang *et al.* [ZGG06] reduces the size of dynamic slices via confidence values. These confidence values represent the likelihood that the corresponding statement computes the correct value. Statements with a high confidence value are excluded from the dynamic slice.

Other mentionable work by Zhang *et al.* includes [ZGZ04; ZGG07; ZTG06]. In [ZGZ04], they discuss how to reduce the time and space required for storing dynamic slices. In [ZTG06], they deal with the problem of handling dynamic slices of long running programs. In [ZGG07], they evaluate the effectiveness of dynamic slicing for locating real bugs. They found out that most of the faults could be captured by considering only data dependencies.

Sridharan *et al.* [SFB07] identified data structures as the main reason for overly large slices. They argue that data structures provided by standard libraries are well-tested and thus they are unlikely to be responsible for an observed misbehavior. Their approach, called "Thin Slicing", removes such statements from slices.

Lyle and Weiser [LW87] introduced the concept of program dicing. A dice is the set difference of the static slice of a variable with a faulty output and the static slice of a variable with a correct output. The resulting dice contains the faulty statement if the following three conditions are satisfied: (1) The test set is reliable. (2) The program contains only one fault. (3) Faults cannot be covered, i.e. if a variable x has an incorrect value and variable y uses the value of x, then the value of y must also be incorrect. If one of these conditions is violated, the slice of the variable with the correct output value might contain the faulty statement. As a consequence, the dice might not contain the faulty statement. Chen and Cheung [CC93] improve the work of Lyle and Weiser by introducing dynamic program dicing. Their approach can be applied even when there exists only one output variable because of the use of dynamic slices. In addition, Chen and Cheung make their approach more robust by using the intersection of several slices of correct variables.

DeMillo *et al.* [DPS96] introduced "Critical Slicing". This slicing technique borrows from mutation-based testing the idea of removing statements [MOK06]. The statements contained in the execution trace of a failing test case are systematically removed from the program. If the reduced program still produces the same values for the variables in the slicing criterion, the removed statements can be ignored. Otherwise the removed statements are critical and therefore must be part of the slice. For a given slicing criterion, we can derive the following relationships: The critical slice is a subset of the static slice. However, the critical slice might contain statements that are not part of the dynamic slice and vice versa.

Many other slicing techniques have been published. For a deeper analysis on slicing techniques the reader is referred to Tip [Tip95] and Kamkar [Kam95] for slicing techniques in general and to Korel and Rilling [KR98] for dynamic slicing techniques. Slicing is often used in conjunction with other debugging techniques. Some of these combining techniques are discussed in Section 2.6.

2.2. Model-based software debugging

Model-based software debugging (MBSD) derives from Model-based diagnosis (MBD), a successful hardware debugging method. MBD techniques have been applied to debug electronic circuits [Rei87], hardware designs specified in VHDL [FSW99], functional programs [SW99], complex configuration knowledge bases [Fel+04], ontologies [Shc+12], and workflow processes [FMS10].

Usually, model-based techniques rely on a model, i.e. a formal description or specification of the program. However, such models are expensive to build and often contain many errors. Musuvathi *et al.* [ME03] mentioned that models for debugging can be as error-prone as the concrete implementation. In MBD, the buggy physical system itself acts as model. MBSD borrows this idea of "reasoning from first principles" and derives a model from the source code or execution trace of the faulty program. The derived model is enhanced with variables representing the "health" state for each component. In contrast to the original program, this model allows for consistency checking.

Console *et al.* [CFD93] were among the first to show how the debugging process of declarative programs written in Prolog can be improved using MBD techniques. Shapiro [Sha83] provided an algorithmic basis for debugging Prolog programs. In his thesis, Shapiro presented a divide-and-query algorithm for interactive fault localization. This algorithm aims to avoid unnecessary interactions of the user with the system when debugging. In contrast to Console *et al.*, Shapiro did not establish his work on the theory of MBD. However, Bond and Pagurek [BP94] showed that Shapiro's technique is a special case of MBD.

Mbsd techniques basically differ in the computational complexity and accuracy owing to the used model. Mayer and Stumptner [MS07; MS08] give an overview of different models used in Mbsd. They distinguish dependency-based, value-based and abstraction-based models. In dependency-based models, the flow of correct and incorrect values through the program is modeled. Wotawa [Wot02] discusses the relationship dependency-based models and program slicing. Dependency-based models have a small computational complexity. Therefore, they can be applied even to large programs. However, these types of models often result in large result sets. In value-based models, the concrete values are known and propagated through the model. These models are more precise, but also computationally more expensive. Therefore, value-based models can only be used to debug small programs. Abstraction-based models are used when precise values cannot be determined.

Mateis *et al.* [Mat+00; MSW00] were among the first who applied Mbd to an object-oriented language. In their work, Mateis *et al.* describe the conversion of Java programs into simple dependency models. Wotawa *et al.* [WN08; WNM12] present a model-based software debugging approach which relies on a value-based model. They show how to formulate a debugging problem as a constraint satisfaction problem. In addition, Wotawa *et al.* [Wot+09] discuss the computational costs of Mbsd. They show how to transform a debugging problem into a constraint satisfaction problem (Csp). A Csp can be represented as a hypertree. The width of a hypertree is a good indicator for the complexity of the Csp. In addition, they prove that there exists no constant upper-bound for the hypertree width of arbitrary programs. The upper-bound of the hypertree width for a particular program is given by the number of statements in the program. In case of loops, there exists an upper bound on the hypertree width for the particular program. Problems with a hypertree width > 5 are said to be hard problems. The authors showed that the hypertree width is often greater than 5 even for small programs. Therefore, they conclude that debugging is a very hard problem.

Nica *et al.* [NWW09] suggested to use assertions and invariants in Mbsd. They found out that even weak invariants can be useful when debugging. Their empirical evaluation shows that the use of assertion information eliminates additional 30 % of the fault candidates. Other mentionable work of Nica *et al.* includes their work on distinguishing test cases [NNW12]. Their approach computes mutations of the program as possible repair suggestions. As the number of possible repairs can be huge, they make use of distinguishing test cases in order to reduce the number of solutions presented to the user.

2.3. Spectrum-based fault localization

Many debugging techniques focus on the usage of failing test cases only. In contrast, Spectrum-based fault localization (Sfl) is a technique that uses the execution specific information of both passing and failing test cases. This execution specific information is called program spectra. There are several types of program spectra, e.g. hit spectra and count spectra. Harrold *et al.* [Har+98] give an overview of different types of program spectra. The program spectra are stored in so-called observation matrices. Besides the spectra information, these observation matrices also contain the information which test cases were passing and which were failing onces. The collection of the test result of all test cases is called error vector. A high similarity of a statement to the error vector indicates a high probability that the statement is responsible for the error [Abr+09b]. There exist several similarity coefficients to numerically express the degree of similarity, e.g. Jaccard [Zoe+07], Tarantula [JH05], CrossTab [Won+08], and MKBC [Xu+11].

The "Nearest Neighbor" approach [RR03] developed by Renieris and Reiss selects a passing test case that resembles the failing test case most with respect to the Hamming distance. The failing test case minus this passing test case builds the set of suspicious statements. From this set, the program dependency graph is traversed until the fault is found. This approach performs better than simple union and intersection approaches. However, it does not perform as good as Tarantula and Ochiai. The Liblit05 [Lib+05] and the Sober [Liu+06] approaches use predicate switching for fault localization. Similar to the "Nearest Neighbor" approach, these approaches make use of a breadth-first search in the program dependency graph.

Wong *et al.* [WDC10] introduce three coverage-based heuristics for fault localization. The most interesting Heuristic is Heuristic III where the test cases are grouped. Tests in the same group contribute with the same weight to the suspiciousness of a statement. The weights of the different groups are chosen in a way that the total contribution of the passing test cases is smaller than the total contribution of the failing test cases. The authors present the tool χ-Debug which comes with a user-friendly GUI that highlights the suspiciousness of statements in color. In an empirical evaluation, the authors show that Heuristic III is well-suited for fault localization and outperforms other debugging techniques like Tarantula.

Jones *et al.* [JHB07] address the problem of using Sfl in case of several faults. They create fault focusing clusters and specialized test suites that allow for localizing single faults with Sfl. Therefore, they propose two clustering methods: (1) clustering based on profiles and fault localization results, and (2) clustering based on fault localization results. They propose to use as stopping criterion for the clustering a set-similarity coefficient that is based on the suspicious values of the statements. DiGuiseppe and Jones [DJ12] introduce “Concept-Based Failure Clustering”. This technique uses latent-semantic analyses to cluster test cases.

2.4. Delta Debugging

Zeller and Hildebrandt [ZH02] introduced delta debugging. Delta debugging is a technique that can be used to systematically minimize a failure-inducing input. The basic idea of delta debugging is: the smaller the failure-inducing input, the less program code is covered. Therefore, delta debugging can be used as pre-processing step before starting the real fault localization process. Zeller and Hildebrandt propose a modified version of the delta debugging algorithm that allows for isolating the failure-inducing input. Furthermore, there exist adapted versions of the delta debugging [Zel02; CZ05] enabling to directly use it for fault localization.

2.5. Genetic approaches

Genetic debugging is a technique that locates faults by searching for possible repairs. It is based on genetic programming and generates mutants for a given faulty program. A mutation can be for example to insert additional code, to change operators or to delete code. The generated mutants are evaluated with the help of the available test cases. The best mutants are selected and further mutated until all test cases pass. Genetic Programming often uses fault localization techniques as a pre-processing step. Arcuri [Arc08], Weimer *et al.* [Wei+09; Wei+10] and Debroy and Wong [DW10] have published the most prominent approaches in this field of debugging.

2.6. Hybrid techniques

There exist several methods combining some of the previously mentioned approaches. For example, Deputo and Barinel combine Mbsd with Sfl. Deputo [Abr+09a] uses Sfl to obtain an initial fault ranking and Mbsd afterwards to filter out those statements of the ranking that cannot explain the observed misbehavior. If no explanation is found after the Mbsd step, a best-first search is applied: The program statements are traversed along the data- and control dependencies starting with the statements with the highest fault probability. In contrast to Mbsd, Deputo implements an incremental debugging strategy: the algorithm stops as soon as a fault has been identified. As mentioned in [Abr+09a], Deputo is 2.5 times faster than Mbsd. Barinel [AG10] uses the program spectra obtained by Sfl to model the program behavior. Afterwards, Barinel employs a Bayesian approach to deduce multiple-fault candidates.

Gupta *et al.* [Gup+05] present a technique that combines delta debugging with forward and backward slices. Delta debugging is used to find the minimal failure-inducing input. Forward slices are computed for the failure-inducing input. Backward slices are computed for the erroneous output variables. The intersection of the forward and backward slices results in the failure-inducing chop. Burger and Zeller [BZ11] propose an approach named Jinsi that combines event slicing, delta debugging and dynamic slicing.

Wen *et al.* [Wen+11; Wen12] introduce an approach that is based on program slicing and statistical debugging, namely PSS-SFL (program slicing spectrum-based software fault localization). This approach extracts the dependencies of program components, eliminates unrelated statements, builds a program slicing spectrum model and ranks the statements with a new suspiciousness metric.

3. Preliminaries

The content of this chapter is based on the work published in [HW12a], [HW12c] and [HWA12]. In Section 3.1, basic definitions are given and the concept of dynamic slicing is explained. Model-based software debugging (Mbsd) and its variations are discussed in Section 3.2. Spectrum-based Fault localization (Sfl) is discussed in Section 3.3.

3.1. Basic definitions

In this thesis, the language $\mathcal{L}$ is used for explaining the basic functionality of the approaches Sendys and Conbas (Chapters 4 and 5). The introduced language $\mathcal{L}$ is approximated to Java, but restricted to non-object-oriented functionality. The reason for the simplification is to focus on the underlying ideas instead of solving technical details. However, this simplification does not restrict generality. The empirical evaluations in the Chapters 4 and 5 are performed on programs written in Java.

In this section, we briefly introduce this language $\mathcal{L}$ with respect to syntax and semantic. In addition, we define (failing) test cases and the debugging problem. The approaches described in Chapter 4 and 5 rely on dynamic slices. Therefore, we define execution traces and dynamic slices formally in Section 3.1.3.

3.1.1. The Language $\mathcal{L}$

The syntax definition of $\mathcal{L}$ is given in Backus-Naur form (BNF) in Figure 3.1. Terminals of the language $\mathcal{L}$ are underscored. The start symbol of the grammar is P. A program P comprises a sequence of statements $stmt$. There are three different types of statements: (1) the assignment statement, (2) the if-then-else statement, and (3) the while-statement. In the following, if-then-else statements and while-statements are referred to as conditional statements (or conditionals). The right side of an assignment has to be a variable (*id*). The name of the variable can be any word except a keyword. An expression is

$$
\begin{array}{lcl}
P & ::= & S \\
S & ::= & stmt\ S \\
 & | & \epsilon \\
stmt & ::= & id\ \underline{=}\ E\ \underline{;} \\
 & | & \underline{\textbf{if}}\ E\ \underline{\{}\ stmt\ \underline{\}}\ else \\
 & | & \underline{\textbf{while}}\ E\ \underline{\{}\ stmt\ \underline{\}} \\
else & ::= & \underline{\textbf{else}}\ \underline{\{} stmt\ \underline{\}} \\
 & | & \epsilon \\
E & ::= & id \\
 & | & num \\
 & | & \underline{\textbf{true}} \\
 & | & \underline{\textbf{false}} \\
 & | & E\ op\ E \\
 & | & \underline{(}\ E\ \underline{)} \\
op & ::= & \underline{+} \mid \underline{-} \mid \underline{*} \mid \underline{/} \mid \underline{\leq} \mid \underline{\geq} \mid \underline{==} \mid \underline{<>} \mid \underline{\&} \mid \underline{|}
\end{array}
$$

Figure 3.1.: The syntax of the language $\mathcal{L}$

either an Integer (*num*), a truth value (*true*, *false*), a variable, or two expressions concatenated with an operator. An Integer number is a sequence of any digit from 0 to 9. It can be preceded by '-' for negative numbers. Data types are not introduced in $\mathcal{L}$, but it is assumed that the use of Boolean and Integer values follow the usual expected type rules. Comments start with <u>//</u> and are terminated when the line ends.

Example 3.1 *Listing 3.1 shows an example program written in $\mathcal{L}$.* □

```
while (a>0) {
    x = a * c;
    y = b * 3;
    z = x + y;
    a = a - 1;
}
if (b<0) {
    z = z + 20;
}
```

Listing 3.1: A program written in the language $\mathcal{L}$

After defining the syntax of $\mathcal{L}$, its semantics have to be defined. The interpretation function $[\![.]\!]$ maps programs and states to new states or the undefined value $\perp$:

$$\text{Interpretation function } [\![.]\!] : \mathcal{L} \times \Sigma \mapsto \Sigma \cup \{\perp\}$$

In this definition, Σ represents the set of all states. A concrete state $\omega \in \Sigma$ specifies values for the variables used in the program. A state is called variable environment. Hence, ω itself is a function ω : VARS $\rightarrow$ DOM where VARS denote the set of variables and DOM its domain comprising all possible values. $\omega \in \Sigma$ is represented as a set $\{\ldots, x_i|v_i, \ldots\}$ where $x_i \in$ VARS is a variable, and $v_i \in$ DOM is its value. DOM $= \mathbb{Z}^M \cup \mathbb{B}$ where $\mathbb{Z}^M = \{x \in \mathbb{Z}|min \leq x \leq max\}$ is an Integer number between a pre-defined minimum and maximum value, and $\mathbb{B} = \{\mathbf{T}, \mathbf{F}\}$ is the Boolean domain. We assume that all variable values necessary to execute the program are known and defined in $\omega \in \Sigma$.

The definition of the semantics of $\mathcal{L}$ is given in Figure 3.2. For the semantics of conditions and expressions, we assume that *num* (*id*) represents the lexical value of the token *num* (*id*) used in the definition of the grammar. An Integer *num* is evaluated to its corresponding value $num* \in \mathbb{Z}^M$ and the truth values are evaluated to their corresponding values in $\mathbb{B}$. A variable *id* is evaluated to its value specified by the current variable environment ω. Expressions with operators are evaluated accordingly to the semantics of the used operator.

The semantics of the statements in $\mathcal{L}$ is defined in a similar manner: A sequence of statements, i.e., the program itself or a sub-block of a conditional or while statement, $S_1 \ldots S_n$, is evaluated by executing the statements S_1 to S_n in the given order. Each statement might change the current state of the program. An assignment statement changes the state for a given variable. All other variables remain unchanged. An if-then-else statement allows for selecting a certain path (via block B_1 or B_2) based on the execution of the condition. A while-statement executes its block B until the condition evaluates to false. Therefore, the formal definition of the semantics is very similar to the semantics definition of an if-then-else statement without else-branch. If a program does not terminate or in case of a division by zero, the semantics function returns the undefined value $\perp$.

Example 3.2 *When executing the program Π from Listing 3.1 on the input state $\omega_I = \{(a,1),(b,1),(c,2),(z,0)\}$ the semantics function $[\![.]\!]$ returns the output state $\omega_O = \{(a,0),(b,1),(c,2),(x,2),(y,3),(z,5)\}$.* $\square$

3.1.2. The Debugging Problem

For stating the debugging problem, we have to define test cases and test suites. In the context this thesis, a test case comprises information about the values of input variables and some information regarding the expected output. In principle, it is possible to define expected values for variables at arbitrary

Semantics of expressions:

$$
\begin{aligned}
&[\![num]\!]\omega = num* \\
&[\![\underline{\textbf{true}}]\!]\omega = \mathbf{T} \\
&[\![\underline{\textbf{false}}]\!]\omega = \mathbf{F} \\
&[\![id]\!]\omega = \omega(id) \\
&[\![\underline{(}E\underline{)}]\!]\omega = [\![E]\!]\omega
\end{aligned}
$$

$$
[\![E_1 \text{ op } E_2]\!]\omega = \begin{cases}
[\![E_1]\!]\omega & + & [\![E_2]\!]\omega & \text{if op = ‘}\underline{+}\text{”} \\
[\![E_1]\!]\omega & - & [\![E_2]\!]\omega & \text{if op = ’}\underline{-}\text{’} \\
[\![E_1]\!]\omega & * & [\![E_2]\!]\omega & \text{if op = ’}\underline{*}\text{’} \\
[\![E_1]\!]\omega & / & [\![E_2]\!]\omega & \text{if op = ’}\underline{/}\text{’} \\
[\![E_1]\!]\omega & < & [\![E_2]\!]\omega & \text{if op = ’}\underline{<}\text{’} \\
[\![E_1]\!]\omega & > & [\![E_2]\!]\omega & \text{if op = ’}\underline{>}\text{’} \\
[\![E_1]\!]\omega & = & [\![E_2]\!]\omega & \text{if op = ’}\underline{==}\text{’} \\
[\![E_1]\!]\omega & \neq & [\![E_2]\!]\omega & \text{if op = ’}\underline{<>}\text{’} \\
[\![E_1]\!]\omega & \wedge & [\![E_2]\!]\omega & \text{if op = ’}\underline{\&}\text{’} \\
[\![E_1]\!]\omega & \vee & [\![E_2]\!]\omega & \text{if op = ’}\underline{|}\text{’}
\end{cases}
$$

Semantics of statements:

$$
\begin{aligned}
&[\![\ ]\!]\omega = \omega \\
&[\![S_1 \ldots S_n]\!]\omega = [\![\ldots S_n]\!]\ ([\![S_1]\!]\omega)
\end{aligned}
$$

$$
[\![\text{id } \underline{=} \ E\underline{;}]\!]\omega = \omega' \text{ with } \omega'(x) = \begin{cases} [\![E]\!]\omega & \text{if } x = id \\ \omega(x) & \text{otherwise} \end{cases}
$$

$$
[\![\underline{\textbf{if}}\ E\ \underline{\{}B_1\underline{\}}\ ]\!]\omega = \begin{cases} [\![B_1]\!]\omega & \text{if } [\![E]\!]\omega = \mathbf{T} \\ \omega & \text{otherwise} \end{cases}
$$

$$
[\![\underline{\textbf{if}}\ E\ \underline{\{}B_1\underline{\}}\ \ \underline{\textbf{else}}\ \ \underline{\{}B_2\underline{\}}]\!]\omega = \begin{cases} [\![B_1]\!]\omega & \text{if } [\![E]\!]\omega = \mathbf{T} \\ [\![B_2]\!]\omega & \text{otherwise} \end{cases}
$$

$$
[\![\underline{\textbf{while}}\ E\ \underline{\{}B\underline{\}}]\!]\omega = \begin{cases} [\![B]\!][\![\underline{\textbf{while}}\ E\ \underline{\{}B\underline{\}}]\!]\omega & \text{if } [\![E]\!]\omega = \mathbf{T} \\ \omega & \text{otherwise} \end{cases}
$$

Figure 3.2.: The semantics of the language $\mathcal{L}$

positions in the code. For reasons of simplicity, we do not make use of such an extended definition and we do not discuss testing in general. Instead, we refer the interested reader to the standard books on testing, e.g., [Bei90].

Definition 1 (Test case) *A test case t is a tuple (I, O) where $I \in \Sigma$ is the input and $O \in \Sigma$ is the expected output.*

A given program $\Pi \in \mathcal{L}$ passes a test case $t = (I, O)$ iff $[\![\Pi]\!] I \supseteq O$. Otherwise, the program fails test case t. Because of the use of the $\supseteq$ operator, partial test cases are allowed which do not specify values for all output variables.

Example 3.3 *A test case t for the example from Listing 3.1 is for instance $I = \{(a, 1), (b, 1), (c, 2), (z, 0)\}$ and $O = \{(z, 6)\}$. This is obviously a failing test case since the evaluation of the example on the input I is not a superset of the expected output ($[\![\Pi]\!] I = \{(a, 0), (b, 1), (c, 2), (x, 2), (y, 3), (z, 5)\} \not\supseteq O$).* □

Definition 2 (Test suite) *A test suite T for a program $\Pi \in \mathcal{L}$ is a set of test cases.*

We divide a test suite into two disjoint sets comprising only passing (PASS) respectively failing (FAIL) test cases, i.e., $T = \text{PASS} \cup \text{FAIL}$ and $\text{PASS} \cap \text{FAIL} = \emptyset$. Formally, we define these two subsets as follows:

$$\text{PASS} = \{(I, O) | (I, O) \in T, [\![\Pi]\!] I \supseteq O\} \tag{3.1}$$

$$\text{FAIL} = T \setminus \text{PASS} \tag{3.2}$$

For a negative test case $t = (I, O) \in \text{FAIL}$, there must be some variables $x_1, \ldots, x_k$ where for all $i \in \{1, \ldots, k\}$ the expected value $v_i \in O$ and the computed value $w_i \in [\![\Pi]\!] I$ are different ($v_i \neq w_i$). We call such variables $x_1, \ldots, x_k$ conflicting variables $CV(t)$. If the test case t is a positive test case, the set $CV(t)$ is defined to be empty.

Definition 3 *The set of conflicting variables CV for a test case t with (I, O) consists of all variables x where the computed value $w_i \in [\![\Pi]\!] I$ differs from the expected value $v_i \in O$.*

Example 3.4 *For our running example from Listing 3.1 and test case t, the set of conflicting variables is $CV(t) = \{z\}$.* □

Using these definitions, we define the debugging problem as follows:

Definition 4 (Debugging problem) *Given a program $\Pi \in \mathcal{L}$ and a test suite T containing at least one failing test case. The problem of identifying the root cause for a failing test case $t \in T$ in Π is called the debugging problem.*

This definitions describes the debugging problem as a fault localization problem. Another possibility is to describe the debugging problem as a fault correction problem. A solution for the debugging problem as a fault localization problem is a set of statements in a program Π that are responsible for the conflicting variables $\mathrm{CV}(t)$. The identified statements in a solution have to be changed in order to turn all failing test cases into passing test cases for the corrected program.

3.1.3. Program Slicing

When obtaining a result that contradicts with the expectations, we are interested in finding the fault, i.e., locating the statements that are responsible for the fault. Mark Weiser [Wei82] introduced the idea to support this process by using the dependence information represented in the program. Weiser's approach identifies those parts of the program that contribute to faulty computations, i.e., the slice. In this thesis, we use extensions of Weiser's static slicing approach and consider the dynamic case where only statements are considered that are executed in a particular test run. In order to define dynamic slices [KL88] and further on relevant slices [GBF99], we first introduce execution traces.

Definition 5 (Execution trace) *An execution trace of a program $\Pi \in \mathcal{L}$ and an input state $\omega \in \Sigma$ is a sequence $\langle s_1, \ldots, s_k \rangle$ where $s_i \in \Pi$ is a statement that has been executed when running Π on test input ω, i.e., calling $[\![\Pi]\!]\omega$.*

Example 3.5 *For our example from Listing 3.1, the execution trace of the input ω_I is illustrated in Listing 3.2 and comprises the statements 1-7. Statement 8 is not executed.* □

For formally defining dependence relations, we introduce the functions DEF and REF: DEF returns a set of variables defined in a statement. REF returns a set of variables referenced (or used) in the statement. DEF returns the empty set for conditional statements and a set representing the variable on the left side of an assignment statement. With these functions, we define data dependencies as follows:

```
(1  while (a>0))    // true
(2    x = a * c;)   // x = 2
(3    y = b * 3;)   // y = 3
(4    z = x + y;)   // z = 5
(5    a = a - 1;)   // a = 0
(1  while (a>0))    // false
(7  if (b<0))       // false
```

Listing 3.2: The execution trace of example from Listing. 3.1

Definition 6 (Data dependency) *Given an execution trace $\langle s_1, \ldots, s_k \rangle$ for a program Π and an input state $\omega \in \Sigma$. An element of the execution trace s_j is data dependent on another element s_i where $i < j$, i.e., $s_i \rightarrow_D s_j$, iff there exists a variable x that is element of $\textsc{Def}(s_i)$ and $\textsc{Ref}(s_j)$, and there exists no element s_k, $i < k < j$, in the execution trace where $x \in \textsc{Def}(s_k)$.*

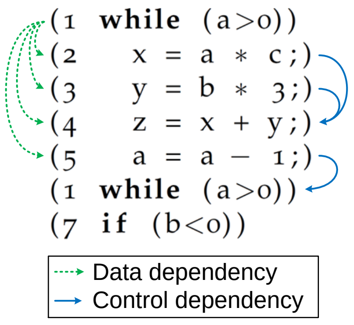
Beside data dependencies, we have to deal with control dependencies representing the control flow of a given execution trace. In $\mathcal{L}$, only if-then-else and while statements influence the control flow. Therefore, we only have to consider these two types of statements.

Definition 7 (Control dependency) *Given an execution trace $\langle s_1, \ldots, s_k \rangle$ for a program Π and an input state $\omega \in \Sigma$. An element of the execution trace s_j is control dependent on a conditional statement s_i with $i < j$, i.e., $s_i \rightarrow_C s_j$, iff the execution of s_i causes the execution of s_j.*

If the condition of the while statement executes to true, then all statements of the outermost sub-block of the while-statement are control dependent on this while statement. If the condition evaluates to false, no statement is control dependent because the first statement after the while-statement is always executed regardless of the evaluation of the condition. Please note that infinite loops are out of the scope of this thesis and therefore they are not considered.

If the condition of an if-then-else statement evaluates to true, the statements of the then-block are control dependent on the conditional statement. If it evaluates to false, the statements of the else-block are control dependent on the conditional statement. In case of nested while-statements or if-then-else statements, the control dependencies are not automatically assigned for the blocks of the inner while-statements or if-then-else statements.

Example 3.6 *Figure 3.3 shows the execution trace for our running example where the data and control dependencies have been added.* □



```
(1 while (a>0))
(2   x = a * c;)
(3   y = b * 3;)
(4   z = x + y;)
(5   a = a - 1;)
(1 while (a>0))
(7 if (b<0))
```

Figure 3.3.: The execution trace of the example from Listing 3.2 enhanced with data and control dependencies

In addition to data and control dependencies, relevant slicing uses potential data dependencies [GBF99]. A potential data dependency occurs whenever the evaluation of a conditional statement causes the non-execution of statements which potentially change the value of a variable. Ignoring such potential data dependencies might lead to slices where the faulty statements are missing.

Definition 8 (Potential relevant variables) *Given a conditional (if-then-else or while) statement n. The potential relevant variables are a function PR that maps the conditional statement and a Boolean value to the set of all defined variables in the block of n that is not executed because the corresponding condition of n evaluates to true or false.*

If there are other while-statements or if-then-else statements in a sub-block, the defined variables of all their sub-blocks must be considered as well. Table 3.1 summarizes the definition of potential relevant variables.

Table 3.1.: Potential relevant variables

Statement n	Condition E	Potential relevant variables PR
`while` E {	true	$PR(n, \text{true}) = \{\}$
B	false	$PR(n, \text{false}) = \{m \mid m \text{ defined in } B\}$
}		
`if` E {		
B_1	true	$PR(n, \text{true}) = \{m \mid m \text{ defined in } B_2\}$
} `else` {	false	$PR(n, \text{false}) = \{m \mid m \text{ defined in } B_1\}$
B_2		
}		

Example 3.7 *There are the following potential relevant variables for the execution trace from Listing 3.2:* $PR(while(a > 0), true) = \{\}$, $PR(while(a > 0), false) = \{a, x, y, z\}$, $PR(if(b < 0), true) = \{\}$, *and* $PR(if(b < 0), false) = \{z\}$. □

Based on this definition of the potential relevant variables, we define potential data dependencies straightforward.

Definition 9 (Potential data dependency) *Given an execution trace* $\langle s_1, \dots, s_k \rangle$ *for a program* Π *and an input state* $\omega \in \Sigma$. *An element of the execution trace* s_j *is potentially data dependent on a conditional statement* s_i *with* $i < j$, *which evaluates to true (false) , i.e.,* $s_i \rightarrow_P s_j$, *iff there is a variable* $x \in PR(s_i, true)$ $(x \in PR(s_i, false))$ *that is referenced in* s_j *and not re-defined between* i *and* j.

After defining the dependence relations of a program that is executed on a given input state, we are able to formalize relevant slices.

Definition 10 (Relevant slice) *A relevant slice* S *of a program* $\Pi \in \mathcal{L}$ *for a slicing criterion* (ω, x, n), *where* $\omega \in \Sigma$ *is an input state,* x *is a variable and* n *is a line number in the execution trace, comprises those parts of* Π *that contribute to the computation of the value for* x *at the given line number* n.

Definition 11 (Execution trace graph) *An execution trace graph (ETG) is a directed acyclic graph that represents an execution trace extended by data, control and potential data dependencies.*

A statement contributes to the computation of a variable value if there is a dependence relation. Hence, computing slices can be done by following the dependencies in the ETG. Algorithm 3.1 computes the relevant slice for a given execution trace ET of a program Π and a given variable x at the execution trace position n. First, the execution trace graph is computed from ET and Π. The program Π is required for determining the potential data dependencies. Afterwards, the statement where the variable x is defined for the last time is marked (Step 2). All control statements c are marked where the following two rules apply: (1) c succeeds the previously marked statement in the ETG and (2) c has the variable x in its PR set (Step 3). From the marked nodes, traverse the graph in reverse direction and mark all reached nodes (Step 4). The relevant slice comprises all statements that are marked.

Example 3.8 *Algorithm 3.1 is applied to our running example for the variable* z *at Line 7 as follows: First, the ETG (see Figure 3.3) is computed. In Step 2, the statement in Line 4 is marked. The conditional statements that occur after Line 4*

Algorithm 3.1 RelevantSlice(ET, Π, x, n)

```
Require: Execution trace ET = $\langle s_1, \ldots, s_m \rangle$
Require: Program $\Pi$
Require: Variable of interest $x$
Require: Certain line number of the execution trace $n$
Ensure: Relevant slice $S$.
 1: Compute the execution trace graph ETG using the dependence relations
    $\rightarrow_C$, $\rightarrow_D$, and $\rightarrow_P$.
 2: Mark the node $s_k$ in the ETG, where $x \in \mathrm{DEF}(s_k)$ and there is no other
    statement $s_i$, $k < i \leq n$, $x \in \mathrm{DEF}(s_i)$ in the ETG.
 3: Mark all test nodes between $s_k$ and $s_n$, which evaluate to the Boolean
    value $B$ and where $x \in \mathrm{PR}(s_k, B)$.
 4: Traverse the ETG from the marked nodes in the reverse direction of the
    arcs until no new nodes can be marked. Let $S$ be the set of all marked
    nodes.
 5: return Set $S$
```

and that have the variable z in their PR set are marked in Step 3. These conditional statements are the second while-statement and the if-statement. From these three marked statements, we traverse the graph in reverse direction and mark all reachable statements in Step 4. In this example, all statement are marked and are therefore part of the slice for variable z. □

3.2. Model-based software debugging

Model-based software debugging (MBSD) is a software debugging technique that is adapted from a hardware debugging technique, namely model-based debugging (MBD). In MBD, diagnoses are derived from observations and a model that captures the correct behavior of components. In 1987, Reiter [Rei87] introduced the theoretical foundations of MBD. The basic idea of MBD is to formalize the behavior of each component C in the form $\neg \mathrm{AB}(C) \rightarrow \mathrm{BEHAV}(C)$. The predicate AB stands for abnormal and is used to state the incorrectness of a component. Hence, when C is correct, $\neg \mathrm{AB}(C)$ has to be true and the behavior of C has to be valid. In software debugging, we make use of the same underlying idea. Instead of components, we deal with statements. The behavior of a statement is given by a formal representation of the statement's source code. We use constraints for this representation. The total of all constraints represents the system description SD.

For representing bug candidates in the context of software debugging, Reiter's definition of a diagnosis [Rei87] is adapted: A diagnosis is formalized as a set of correctness assumptions that does not lead to a contradiction with respect to the given observations (test case).

Definition 12 (Diagnosis) *Given a formal representation SD (i.e. a system description) of a program $\Pi \in \mathcal{L}$, where the behavior of each statement s_i is represented as $\neg AB(s_i) \rightarrow BEHAV(s_i)$, and a failing test case (I, O). A diagnosis Δ (or bug candidate) is a subset of the set of statements of Π such that $SD \cup \{I, O\} \cup \{\neg AB(s) | s \in \Pi \setminus \Delta\} \cup \{AB(s) | s \in \Delta\}$ is satisfiable.*

A diagnosis is minimal if there does not exist a proper subset of this diagnosis that is a diagnosis. For further information about the definition of diagnoses, we refer the interested reader to [Rei87; KMR92; CDT91]. Please note that the program Π can be represented through its source code or through the execution trace of a concrete test case.

In the following subsections, we explain the concept of MBSD with value-based models (Section 3.2.1) and with dependency-based models (Section 3.2.2). Subsequently, we explain the slicing-hitting-set approach, which is an MBSD technique that uses a dependency-based model.

3.2.1. Value-based models

The use of a constraint representation transforms the task of checking for consistency into a constraint satisfaction problem (CSP).

Definition 13 (Constraint Satisfaction Problem (CSP)) *A constraint satisfaction problem is a tuple (V, D, CON) where V is a set of variables defined over a set of domains D connected to each other by a set of arithmetic and Boolean relations, called constraints CON. A solution for a CSP represents a valid instantiation of the variables V with values from D such that none of the constraints in CON is violated.*

We refer to Dechter [Dec03] for more information on constraints and the constraint satisfaction problem.

For retrieving the constraint representation of a program Π, we rely on the work of Nica *et al.* [NNW12] and Wotawa *et al.* [WNM12]. The transformation of a program into constraints comprises two major steps: (1) the conversion of the execution trace into its static single assignment (SSA) form [BM94] and (2) the conversion of the SSA representation into constraints. In contrast to Nica *et al.* and Wotawa *et al.*, loop unrolling is not necessary in our approach since we rely on a concrete execution trace.

The SSA form is an intermediate representation of a program with the property that no two left-side variables share the same name. The SSA form can be easily obtained from an execution trace by adding an index to each variable: Every time a variable is re-defined, the value of the index gets incremented. Every time a variable is referenced, the most recent index is used. Algorithm 3.2 formalizes the conversion of an execution trace ET into its SSA form. This algorithm makes use of the function INDEX. This function returns the current index of a variable. The expression 'y_INDEX(y)' (see Step 5) is a String concatenation of the variable y, '_' and the Integer returned by INDEX(y).

Algorithm 3.2 SSA(ET)

```
Require: An execution trace ET $= \langle s_1, \dots, s_m \rangle$.
Ensure: The execution trace ET$'$ in SSA form and a function INDEX that maps
    each variable to its maximum index value used in the SSA form.
 1: Let INDEX be a function mapping variables to Integers. The initial Integer
    value for each variable is 0.
 2: Let ET$'$ be the empty sequence.
 3: for $j = 1$ to $m$ do
 4:   if $s_j$ is an assignment statement of the form $x = E$ then
 5:     Let $E'$ be $E$ where all variables $y \in E$ are replaced with $y$_INDEX($y$).
 6:     Let INDEX($x$) be INDEX($x$) + 1.
 7:     Add $x$_INDEX($x$) $= E'$ to the end of the sequence ET$'$.
 8:   else
 9:     Let $s'$ be the statement $s_j$ where all variables $y \in E$ are replaced with
        $y$_INDEX($y$).
10:     Add $s'$ to the end of the sequence ET$'$.
11:   end if
12: end for
13: Return (ET$'$, INDEX).
```

Example 3.9 *The application of Algorithm 3.2 on the execution trace from Listing 3.2 results in the execution trace shown in Listing 3.3.* □

In the second step, the SSA form of the execution trace is converted into constraints. In this thesis, we make use of mathematical equations instead of using a specific constraint solver language. In order to distinguish equations from statements, we use $==$ to represent the equivalence relation. Algorithm 3.3 formalizes the conversion into constraints. In this algorithm, we make use of the function INDEX that maps each element of ET to a unique identifier representing its corresponding statement. Such a unique identifier might be the line number where the statement starts. We represent each statement n of the execution trace using the logical formula 'AB(n) $\vee$ Constraint',

```
while (a_0 >0)
  x_1 = a_0 * c_0;
  y_1 = b_0 * 3;
  z_1 = x_1 + y_1;
  a_1 = a_0 - 1;
while (a_1 >0)
if (b_0 <0)
```

Listing 3.3: The SSA converted execution trace from Listing 3.2

which is logically equivalent to '$\neg$AB$(n) \rightarrow$ Constraint' (Lines 2 to 9). In the Lines 10 to 15, the given test case is converted into constraints.

Algorithm 3.3 CONSTRAINTS(ET,t with (I,O), INDEXSSA)

```
Require: An execution trace ET = $\langle s_1, \ldots, s_m \rangle$ in SSA form
Require: A test case $t$ with $(I,O)$
Require: INDEXSSA returning the final SSA index value for each variable.
Ensure: The constraint representation of ET and the test case $t$.
 1: Let CON be the empty set.
 2: for $j = 1$ to $m$ do
 3:   if $s_j$ is an assignment statement of the form $x = E$ then
 4:     CON = CON $\cup \{AB(\text{INDEX}(s_j)) \vee (x == E)\}$
 5:   else
 6:     Let $c$ be the condition of $s_j$ where & is replaced by $\wedge$ and | by $\vee$.
 7:     CON = CON $\cup \{AB(\text{INDEX}(s_j)) \vee (c)\}$
 8:   end if
 9: end for
10: for all $(x, v) \in I$ do
11:   CON = CON $\cup \{x\_0 == v\}$
12: end for
13: for all $(x, v) \in O)$ do
14:   CON = CON $\cup \{\text{INDEXSSA}(x) == v\}$
15: end for
16: return $CO$.
```

Example 3.10 *Applying Algorithm 3.3 on the* SSA *form of the execution trace from Listing 3.3 extracts the following constraints:*

$$\left\{\begin{array}{c} a\_0 == 1, \\ b\_0 == 1, \\ c\_0 == 2, \\ z\_0 == 0, \\ z\_1 == 6, \\ \mathit{AB(1)} \vee (a_0 > 0), \\ \mathit{AB(2)} \vee (x_1 == a_0 * c_0), \\ \mathit{AB(3)} \vee (y_1 == b_0 * 3), \\ \mathit{AB(4)} \vee (z_1 == x_1 + y_1), \\ \mathit{AB(5)} \vee (a_1 == a_0 - 1), \\ \mathit{AB(1)} \vee (a_1 > 0), \\ \mathit{AB(7)} \vee (b_0 < 0) \end{array}\right\}$$

Please note that these constraints are written in mathematical notation instead of a specific constraint language. □

Since arrays are not part of the language $\mathcal{L}$, we do not explain their conversion into constraints. Instead, we refer the interested reader to Wotawa *et al.* [WNM12] for details about the handling of arrays.

3.2.2. Dependency-based models

In value based models, the value of certain variables can be derived from other variables. However, in dependency-based models, the relation between variables is abstract. Dependency-based models only capture the information about which variables depend on which other variables. However, it is not possible to derive concrete values.

Algorithm 3.4 illustrates the conversion of a program into constraints. In Lines 1 and 2, the program is transformed into is program dependency graph and further on into its strongly connected super-graph. In the Lines 4 to 13, the nodes of the strongly connected super-graph are converted into constraints. Simple nodes, i.e. nodes with exactly one sub-node, are converted as indicated in Line 7: The conjunction of the negated abnormal variable with the ok-states of all input edges of that node implies the ok-state of the output edge. For more complex nodes, an additional constraint is added that maps the connection of the single sub-nodes (Line 9 to 11).

From Definition 12 and the dependency model obtained with Algorithm 3.4, the diagnoses can be computed. However, Wotawa [Wot02] has proven that slices can be used instead of a dependency model. The advantage of slices is a reduced computation time. In order to slices instead of dependency models, we use the concept of conflicts. Reiter [Rei87] defined conflicts as follows:

Algorithm 3.4 DependencyModel(Π) [Wot02]

```
Require: Program $\Pi$
Ensure: System description SD
 1: Compute the program dependency graph $G$ for $\Pi$
 2: Compute the strongly connected super-graph $S_G$ for $G$
 3: $\mathrm{SD} = \{\}$
 4: for all vertices $n \in S_G$ do
 5:    Let $y$ be the output of $n$
 6:    if $n$ has exactly one sub-vertex then
 7:       $\mathrm{SD} = \mathrm{SD} \cup \{\neg \mathrm{AB}(n) \wedge \bigwedge_{x \in input(n)} ok(x) \rightarrow ok(y)\}$
 8:    else
 9:       Let $m_1, \ldots, m_k$ be the sub-vertices of $n$
10:       $\mathrm{SD} = \mathrm{SD} \cup \{\neg \mathrm{AB}(m_1) \wedge \cdots \wedge \neg \mathrm{AB}(m_k) \rightarrow \mathrm{ab}(n)\}$
11:       $\mathrm{SD} = \mathrm{SD} \cup \{\neg \mathrm{ab}(n) \wedge \bigwedge_{x \in input(n)} ok(x) \rightarrow ok(y\})$
12:    end if
13: end for
14: return SD.
```

Definition 14 (Conflict) *A set C is a conflict for a system description SD and an observation $\{I, O\}$ iff $SD \cup \{I, O\} \cup \{\neq AB(C) | C \in CO\}$ is contradictory.*

Reiter has shown that minimal diagnoses are identical to the hitting sets of a set of conflict sets. A hitting set is defined for a set of sets CO as follows:

Definition 15 (Hitting set) *A set h is a hitting set if and only if for all $x \in CO$ there exists a non-empty intersection between x and h, i.e., $\forall x \in CO : x \cap h \neq \{\}$. A hitting set h is said to be minimal if there exists no real subset of h that is itself a hitting set.*

There exists many different algorithms for computing minimal hitting sets. The most prominent algorithm is the corrected Reiter algorithm [GSW89]. In Algorithm 3.5, we show a minimal hitting sets algorithm that computes minimal hitting sets up to a certain cardinality. The function Sort sorts a given set of sets CO with increasing cardinality of the sets. The left-most set in CO is the set with the lowest cardinality.

3.2.3. Fault Localization Based on Dynamic Slicing and Hitting-Set Computation

Mbsd approaches like the one described in Section 3.2.1 can be very time consuming when debugging large programs. For this reason, Wotawa [Wot02]

Algorithm 3.5 MinHittingSets(CO, n) [Wot10]

Require: Set of conflict sets CO and a maximal size of hitting sets n
Ensure: Minimal hitting sets

```
1:  Sort(CO)
2:  MHS = {}
3:  Generate node $n$ with $h(n) = \{\}$
4:  $L = \{\} \cup n$
5:  $L' = \{\}$
6:  $i = 0$
7:  while $i < n$ and $L \neq \{\}$ do
8:    for all $n \in L$ do
9:      Get left-most set $C \in$ CO where $C \cap h(n) = \{\}$.
10:     if $\nexists C$ with $C \cap h(n) = \{\}$ then
11:       MHS = MHS $\cup \{n\}$
12:     else
13:       for all $x \in C$ do
14:         if $\nexists$ node $m \in L'$ with $h(m) = h(c) \cup x$ then
15:           Generate a new node $n'$ with $h(n') = h(c) \cup x$
16:           if $\nexists$ node $m$ in MHS where $h(m) \subset h(n')$ then
17:             $L' = L' \cup \{h(n')\}$
18:           end if
19:         end if
20:       end for
21:     end if
22:   end for
23:   $i = i + 1$
24:   $L = L'$
25: end while
26: return MHS
```

introduced a lightweight MBSD technique, namely the Slicing-Hitting-Set Computation (SHSC) approach. The basic idea used in this approach is to combine program slices of faulty variables such that they result in minimal diagnoses. Algorithm 3.6 shows the basic approach: For all conflicting variables x in a test case t, a slice is computed (see Line 5). In principle, every type of slice can be computed, but because of its precision and size a relevant slice [GBF99] is favored over static [Wei82] and dynamic slices [KL88]. Each computed slice is a conflict set. All slices together constitute the set of conflict sets CO. Please note that CO is a set of sets. In Line 9, the minimal hitting sets are computed using the function MINHITTINGSETS. This function is called with the predefined number n which indicates the maximal size of hitting sets. Considering the fact that most bugs are single or double fault, $n = 2$ should be a good value.

Algorithm 3.6 ALLDIAGNOSES(Π, T)

```
Require: Program $\Pi$ and test suite $T$
Ensure: Set of minimal diagnoses $\Delta^S$
 1: Conflict set CO $= \{\}$
 2: for all test cases $t \in T$ do
 3:    if $t$ is a failing test case for program $\Pi$ then
 4:       for all conflicting variables $x$ is CV$(t)$ at position $n$ do
 5:          CO $=$ CO $\cup \{\text{RELEVANTSLICE}(\Pi, x, n, t)\}$
 6:       end for
 7:    end if
 8: end for
 9: $\Delta^S$ = MINHITTINGSETS(CO,n)
10: return $\Delta^S$
```

The approach works for one or more failing test cases. In case of a single failing test case with n statements in the slice, the approach delivers n single-fault diagnoses. Single-fault diagnoses are a valuable support for programmers. However, in case of several faults, single-fault diagnoses miss to detect the real faults. In this case, multiple-fault diagnoses can be useful. Nevertheless, multiple-fault diagnoses are confusing since a programmer might check the same statement several times for correctness. An extension of the approach [Wot10] solves this problem by mapping diagnoses back to a summary slice. Algorithm 3.7 illustrates the computation of such a summary slice: First, the initial fault probabilities $p_F(s)$ are computed for all statements $s \in \Pi$ (Step 1). It is assumed that each statement is equal likely to be faulty. Afterwards, the set of leading diagnoses LD is computed from the set of minimal diagnoses Δ^S (Step 2). A leading diagnosis is a superset of at least one minimal diagnosis. The number of leading diagnoses grows exponential with the program size. A boundary value $B \geq 0$ is used to reduce the number of generated supersets. The computed leading diagnoses LD comprise all supersets of the minimal diagnoses Δ^S which have at most B elements more than the contained minimal diagnosis. The fault probabilities $p(\Delta_i)$ for all diagnoses $\Delta_i \in$ LD are computed (Step 3). These fault probabilities are used to compute the fault probabilities $p_{pred}(s)$ of the statements (Step 4). Finally, the statement probabilities are normalized (Step 5) and the statements are sorted using their fault probabilities (Step 6).

Algorithm 3.7 HS-Slice(Π, Δ^S)

```
Require: Program $\Pi$ and minimal diagnoses $\Delta^S$
Ensure: HS-slice $S$
1: Compute initial fault probability of all statements:
       $\forall s \in \Pi : p_F(s) = \frac{1}{|\Pi|}$
2: Compute set of Leading Diagnoses LD for all minimal diagnoses $\Delta^S$:
       $\mathrm{LD}(\Delta^S) = \{x | x \subseteq \Pi \wedge \exists \Delta \in \Delta^S : (x \supseteq \Delta \wedge (|x| - |\Delta|) \leq B)\}$
3: Compute the fault probability for all diagnoses:
       $\forall \Delta_i \in \mathrm{LD} : p(\Delta_i) = \prod_{s \in \Delta_i} p_F(s) \times \prod_{s' \in \Pi \setminus \Delta_i} (1 - p_F(s'))$
4: Derive the probability that a statement $s$ is faulty:
       $\forall s \in \Pi : p_{pred}(s) = \sum_{\Delta \in LD(\Delta^S) \wedge s \in \Delta} p(\Delta)$
5: Normalize the fault probabilities:
       $\forall s \in \Pi : p'_F(s) = \frac{p_{pred}(s)}{\sum_{s' \in \Pi} p_{pred}(s')}$
6: return statements $s$ in descending order of $p'_F(s)$
```

The summary slice enhanced by the fault probabilities is called *HS-slice*. It consists of a set of pairs of statements and normalized fault probabilities as given in Equation 3.3.

$$S = \{(s, p'_F(s)) | \exists \Delta \in \Delta^S : s \in \Delta\} \tag{3.3}$$

Shsc is a dependency-based model [MS08]. Dependency-based models come with the following disadvantage: A program execution with a long chain of control- and data dependencies most likely results in a huge number of possible explanations. However, more accurate models are limited to small programs only because of their computational complexity. SHSC is faster than debugging with value-based models, but it is less precise.

3.3. Spectrum-based fault localization

Spectrum-based fault localization (SFL) is a statistical fault localization technique. SFL is based on an observation matrix from which similarity coefficients are computed for each block. A block can be a component, a method or a compound statement. Blocks with the highest coefficients most likely contain a fault.

The observation matrix consists of two parts: the program spectra and the error vector. A program spectrum is an abstraction of an execution trace. It maps only one specific view of the dynamic behavior of a program [AZG06]. This could be e.g. the number of times the block was executed (block count spectrum) or simply if the block was visited at all (block hit spectrum). A detailed overview of the different types of program spectra can be found in [Har+98]. Block hit spectra are used in the approaches discussed in this thesis. They only indicate which parts of a program have been executed during a run [JAG09]. In the case of block hit spectra, the entries of the observation matrix are Boolean values (covered / not covered). The error vector indicates whether the respective test case passes or fails.

Equation 3.4 shows an observation matrix with M code blocks and N test cases. For example, the entry x_{12} indicates whether Statement 2 was executed in Test Case 1. The entry e_1 indicates whether Test Case 1 passed (e_1 = false) or failed (e_1 = true). The error vector can be interpreted as a hypothetical code block that is responsible for all observed errors. Spectrum-based fault localization aims to find the block whose column vector resembles the error vector most.

$$ObservationMatrix = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1M} \\ x_{21} & x_{22} & \cdots & x_{2M} \\ \vdots & \vdots & \ddots & \vdots \\ x_{N1} & x_{N2} & \cdots & x_{NM} \end{bmatrix} \begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_N \end{bmatrix} \tag{3.4}$$

Algorithm 3.8 shows how to construct an observation matrix for a program Π and a test suite T. The function GETBOOLEANMATRIX (Line 1) creates a Boolean matrix of the size $|T| \times |\Pi + 1|$. Assume that the matrix entries can be accessed by indicating a test case and a certain statement. Initially, all values of that matrix are set to false. The function Run (Line 3) executes the given test case t on program Π and returns the execution trace ET and the test result. Whenever a statement s is executed in a test case t, the corresponding matrix entry is set to true (Line 6). The result of the test case is stored in the observation matrix at position $\Pi + 1$ (Line 9).

Algorithm 3.8 ObservationMatrix(Π, T)

```
Require: Program $\Pi$ and test suite $T$
Ensure: Observation matrix $O$
 1: O = GetBooleanMatrix($|T|$,$|\Pi + 1|$)
 2: for all $t \in T$ do
 3:     [ET,result] = Run($\Pi$,$t$)
 4:     for all $s \in \Pi$ do
 5:         if $s \in$ ET then
 6:             $O[t][s] =$ true
 7:         end if
 8:     end for
 9:     $O[t][\Pi + 1] =$ result
10: end for
11: return $O$
```

The information of the observation matrix can be further compressed as shown in Equations 3.5-3.7. $a_{11}(i)$ represents the number of failed runs in which block i is involved, $a_{10}(i)$ is the number of passed runs in which block i is involved, and $a_{01}(i)$ is the number of failed runs in which block i is not involved.

$$a_{11}(i) = |\{j|x_{ji} = 1 \land e_j = 1\}| \tag{3.5}$$
$$a_{10}(i) = |\{j|x_{ji} = 1 \land e_j = 0\}| \tag{3.6}$$
$$a_{01}(i) = |\{j|x_{ji} = 0 \land e_j = 1\}| \tag{3.7}$$

Spectrum-based fault localization is based on the assumption that a high similarity of a block to the error vector indicates a high probability that a block is responsible for the error [Abr+09b]. In principle, any type of similarity coefficient can be used. We have chosen the Ochiai coefficient, which originates in the molecular biology domain. Several experiments (e.g. [AZG06; Abr+09b]) have shown that it outperforms other coefficients like Tarantula and Jaccard. The Ochiai coefficient is computed as described in Equation 3.8.

$$s_{\text{Ochiai}}(i) = \frac{a_{11}(i)}{\sqrt{(a_{11}(i) + a_{01}(i)) * (a_{11}(i) + a_{10}(i))}} \tag{3.8}$$

It is assumed that a high similarity to the error vector indicates a high probability that the corresponding component of the software causes the detected error. Therefore, the program components are ranked with respect to the calculated similarity coefficients. This information is valuable to guide the developer to the root cause of observed failures.

Spectrum-based fault localization is a black-box diagnosis technique: It can be applied without any additional modeling effort. Furthermore, it can be easily used out-of-the-box when a test suite for the program is available. Spectrum-based fault localization requires only little time for computing diagnoses:

- $O(N)$ for executing N test cases
- $O(N)$ to compute the similarity coefficient per component and thus $O(M \cdot N)$ for computing the similarity coefficients for all M components
- $O(M \cdot \log M)$ to rank the statements in the diagnostic report

Therefore, the overall time complexity is $O(N + M \cdot N + M \cdot \log M)$. The space complexity is low with a complexity of $O(M \cdot N)$ for storing the coverage matrix. It can be reduced to $O(3 \cdot M)$ when updating a_{11}, a_{10}, and a_{01} (see Equations 3.5-3.7) during the execution of the test cases instead of storing the information in an observation matrix.

The major drawback of Sfl is that it only relies on the execution information. Statements with the same execution pattern cannot be distinguished. Thus, the lowest possible granularity for spectrum-based fault localization is the block level. In addition, Sfl cannot reason over multiple faults. Another disadvantage of Sfl is that its fault localization capabilities highly depend on the quality of the test suite. Furthermore, the number of test cases influences the quality of the fault localization [Abr09]. Mayer *et al.* [May+08] suggest to use six failing test cases and 20 passing test cases for debugging.

4. Spectrum-Enhanced Dynamic Slicing

This chapter is based on the work published in [HW11], [HW12c], [HWA12] and [HW12d].

4.1. Introduction

Spectrum-ENhanced DYnamic Slicing (SENDYS) describes an approach that is based on the idea that no single debugging technique is able to cope with all debugging problems. For this reason, SENDYS combines two state-of-the-art approaches: (1) spectrum-based fault localization (SFL, see Section 3.3) [JAG09] and (2) slicing-hitting-set-computation (SHSC, see Section 3.2.3) [Wot10]. We feel confident that only a combination of different fault localization techniques can finally lead to a completely automated debugger that can be used in practice. This firm conviction is motivated by the observation that there are dozens of debugging approaches reported in scientific literature but there is only limited impact in practice. Often, the reason lies in the complexity of the approaches, which finally leads to a bad scalability and thus prevents them from being used in an interactive fashion. The idea of combining different approaches is not new. For example, Mayer *et al.* [May+08] state that no single technique is able to deal with all types of faults. They encourage to combine different fault localization techniques to build more accurate and robust debugging tools. They argue that the combination of semantic and trace-based debugging approaches is particularly appealing, because these approaches work with complementary information. The combination of MBSD with SFL described in this chapter is along that line of thought.

In the SENDYS approach, the SFL coefficients are used as a-priori fault probabilities of single statements in the SHSC approach. This ensures that statements used by many passing test cases are lower ranked than statements, which are only used in failing test cases. The advantage of combining SFL and SHSC lies in the use of the available information for ranking fault candidates. SENDYS makes use of the available dependence information like data and control dependencies of programs. SFL is not able to distinguish statements occurring in the same basic building block. However, SENDYS allows for fault localization at statement level. Moreover, SENDYS analyzes the execution information from both passing and failing test cases. This eliminates the weakness of SHSC: SHSC often ranks initialization statements high because initial statements are contained in most slices. Thus, SENDYS helps to improve the accuracy of fault localization compared to SHSC.

The remainder of this chapter is organized as follows: In Section 4.2, the differences of SENDYS and similar fault localization approaches are discussed. Section 4.3 explains the underlying algorithm, discusses the advantages and disadvantages of SENDYS and analyzes the performance of SENDYS in terms of runtime. In Section 4.4, the usage of SFL, SHSC, and SENDYS are explained by means of an example. The fault localization capabilities of SENDYS are compared to those of the two basic approaches and to other fault localization approaches in Section 4.5. Section 4.6 concludes the approach.

4.2. Related Research

The idea of combining SFL with other approaches is not new. For example, Xu *et al.* [Xu+11] improve spectrum-based fault localization by adding a noise reduction term to the suspiciousness coefficient computation and by using chains of key basic blocks (KBC - Key Block Chain) as program features. SENDYS differs from their approach by adding dynamic dependency information through slices to the data available by SFL.

BARINEL and DEPUTO share the same basic idea with SENDYS: they combine SFL with MBSD. Therefore, we compare SENDYS with BARINEL and DEPUTO in Section 4.5. DEPUTO [Abr+09a] uses SFL to rank statements and MBSD to eliminate the top ranked candidates that cannot explain an observed misbehavior. In contrast, SENDYS directly uses the spectra information in the reasoning step in the model. In addition, SENDYS uses a dependency-based model, which is a lightweight model. Therefore, SENDYS has a lower computational complexity than the more sophisticated model of DEPUTO.

BARINEL [AG10] is a Bayesian framework that computes fault probabilities per statement using maximum likelihood estimation. In contrast to SENDYS, BARINEL relies only on the information which statements were covered and does not make use of dependency information. Thus, it does not filter statements which are executed in faulty runs but do not contribute to the value of the faulty variable(s).

4.3. The SENDYS Algorithm

The basic approaches have already been explained in Sections 3.2.3 (Shsc) and 3.3 (Sfl). The combination of these approaches is illustrated in Figure 4.1.

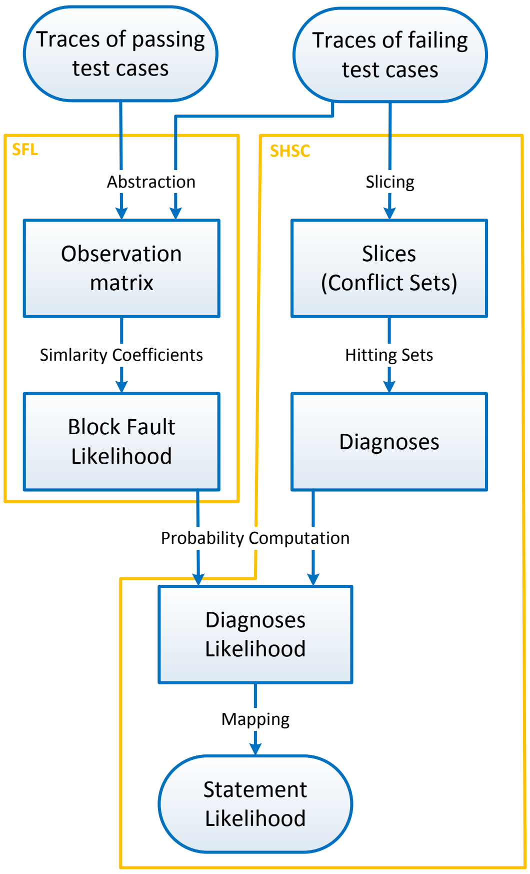


Figure 4.1.: The Sendys approach

The detailed process is given in Algorithm 4.1: First, the observation matrix O (Algorithm 3.8) and the similarity coefficients $sc_o(s)$ for all statements $s \in \Pi$ are computed and normalized ($sc_{norm}(s)$) (Step 1-3). In Step 4, the minimal diagnoses Δ^S are computed using the function AllDiagnoses (Algorithm 3.6). In Step 5, the function HS-Slice (Algorithm 3.7) is used with the normalized similarity coefficients $sc_{norm}(s)$ and the minimal diagnoses Δ^S as input. The resulting summary slice S' is returned.

Similar to related approaches, Sendys highly depends on the quality of the test suite and cannot detect missing statements. However, Sendys eliminates the following disadvantages that are present in the original approaches: On

Algorithm 4.1 SENDYS(Π, T)

```
Require: Program $\Pi$ and test suite $T$
Ensure: HS-Slice $S'$
1: Compute observation matrix $O$ for program $\Pi$ and test suite $T$:
       $O = \textsc{observationMatrix}(\Pi, T)$
2: Compute similarity coefficients $sc_o(s)$ for all statements $s \in \Pi$:
       $\forall s \in \Pi : sc_o(s) = \textsc{ochiai}(s, O)$
3: Compute the normalized values of the similarity coefficients $sc_{norm}(s)$ for
   all statements $s \in \Pi$:
       $\forall s \in \Pi : sc_{norm}(s) = \frac{sc_o(s)}{\sum_{j=1}^{|\Pi|} sc_o(j)}$
4: Compute the minimal diagnoses $\Delta^S$:
       $\Delta^S = \textsc{AllDiagnoses}(\Pi, T)$
5: Compute the summary slice $S'$ with Algorithm 3.7 (HS-SLICE). Start with
   Step 2. Use $sc_{norm}(s)$ instead of $p_F(s)$.
6: return $S'$
```

the one hand it does not rank initialization statements high (compared to Shsc) and on the other hand it is finer grained than Sfl.

You might think that the combined approach is only valuable in case of single faults, which is not true. As mentioned in [Wot10], a probability-based slice is a comprehensive but compact representation. It provides a better overview than a list of diagnoses. There might be statements that are part of several slices. Such statements are investigated by the programmer several times when processing the diagnoses one after another. Therefore, the summary slice *HS-slice* provides an overview of all statements and their fault probabilities. The programmer can process the statements in descending order of their fault probabilities. This guarantees that each statement is investigated only once.

Our debugging method requires a marginal run-time overhead compared to the single approaches. Both approaches, i.e., Sfl and Shsc, require a program Π to be executed, which can be performed in $O(\Pi)$. Given M test cases and N statements, Sfl requires $O(\Pi \cdot M + M \cdot N + N \cdot \log N)$ time as discussed in Section 3.3. For Shsc, the relevant slices are required. The computation of slices depends on the size of the execution trace. In the worst case, the time complexity of computing all relevant slices is $O(\Pi \cdot M)$. In addition to the computation of the slices, it is necessary to compute hitting-sets, which is in the worst case exponential in the size of Π. However, when considering only single and double faults, we retain polynomial complexity. As a consequence the complexity of Algorithm 3.6 (AllDiagnoses) is $O(\Pi \cdot M + N^2)$. Since there exist at maximum N^2 diagnoses when considering only single and double faults, the fault probabilities of the diagnoses and statements in Algorithm 3.7 (HS-Slice) can be computed in $O(N^3)$ time. Thus, Algorithm 3.7 has $O(N^3)$ time complexity for single and double fault diagnoses. Therefore, the overall run-time of Shsc, including all parts of the approach and under the given assumptions, has a time complexity of $O(\Pi \cdot M + N^3)$. When combining these time complexities, the Sendys approach is still bounded by a complexity of $O(\Pi \cdot M + M \cdot N + N^3)$.

4.4. Example of Application

We illustrate the profitableness of Sendys by means of an example. This example deals with transactions on a bank account and is a slight modified version of [Wot10]. We demonstrate the fault localization capabilities for the original fault (Section 4.4.1) and a version containing a fault in an initialization statement (Section 4.4.2).

4.4.1. The bank account example

```
 1 public class BankAccount {
 2   public long balance;
 3   public long limit;
 4   public BankAccount(long bal,long limit){
 5     this.balance = bal;
 6     this.limit = limit;
 7   }
 8   public void withdraw(long amount){
 9     if((balance − amount) >= limit){
10       balance = balance − amount;
11     }
12   }
13   public void deposit(long amount){
14     balance = balance + amount;
15   }
16   public void transferTo(BankAccount acc){
17     long money = this.balance;
18     if(money!=0){ // FAULT
19       this.withdraw(money);
20       acc.deposit(money);
21     }
22   }
23 }
```

Listing 4.1: The Bank Account Example - Line 18 contains a fault. The correct statement would be `if(money>0){`.

Listing 4.1 shows the source code of this example. The balance of the account must never fall below the specified limit. The balance can only be transferred to another account if it is larger than zero. In this example program, we introduce a fault in Line 18. We extend the original example with a constructor in order to be able to highlight the advantage of our approach. The original bank account example contains only one test case (`T1`). Since our approach requires both passing and failing test cases, we extend the test suite. Listing 4.2 shows the extended test suite. Test case `T1` is a failing test case, the other test cases are passing test cases. In the following subsections, we demonstrate how to apply SHSC, SFL, and SENDYS on this example.

Slicing-Hitting-Set-Computation

When using SHSC for debugging, we only consider the failing test cases. The failing test case `T1` has two variables with wrong values: `a1.balance` and `a2.balance`. Therefore, the slices for `a1.balance` ($S_1 = \{5, 6, 9, 10, 17, 18, 19\}$)

```java
public void testTransfer1 { //T1
  BankAccount a1 = new BankAccount(-100,-1000);
    BankAccount a2 = new BankAccount(0,0);
    a2.deposit(200);
    a1.transferTo(a2);
    Assert.assertEquals(-100, a1.balance);
    Assert.assertEquals(200, a2.balance);
}
public void testWithdraw(){ //T2
  BankAccount a1 = new BankAccount(0,-1000);
    a1.withdraw(100);
    Assert.assertEquals(-100, a1.balance);
}
public void testDeposit() { //T3
    BankAccount a2 = new BankAccount(100,0);
    a2.deposit(200);
    Assert.assertEquals(300, a2.balance);
}
public void testTransfer2() { //T4
    BankAccount a1 = new BankAccount(0,-1000);
    a1.withdraw(100);
    BankAccount a2 = new BankAccount(0,0);
    a2.deposit(200);
    a2.transferTo(a1);
    Assert.assertEquals(100, a1.balance);
    Assert.assertEquals(0, a2.balance);
}
public void testTransfer3() { //T5
    BankAccount a1 = new BankAccount(0,-1000);
    BankAccount a2 = new BankAccount(0,0);
    a2.deposit(200);
    a1.transferTo(a2);
    Assert.assertEquals(0, a1.balance);
    Assert.assertEquals(200, a2.balance);
}
```

Listing 4.2: The Bank Account Example - An extended test suite.

and `a2.balance` ($S_2 = \{5, 14, 17, 18, 20\}$) are computed. From these two slices, 11 minimal diagnoses can be computed by means of an hitting set algorithm: 3 single fault explanations ({5}, {17} and {18}) and 8 double fault explanations ({6,14}, {6,20}, {9,14}, {10,14}, {14,19}, {9,20}, {10,20}, and {19,20}). We set the boundary value $B = 0$ so that the set of Leading Diagnoses LD is identical to the computed minimal diagnoses Δ^S. This simplifies the computation and eases the traceability of this example. The results can change somewhat if B is set to a higher value, but it is assumed that the changes are not substantial. We set the initial fault probabilities to $p_F(s) = 1/9$ because there are nine different statements contained in the slices. The fault probabilities for the single fault diagnoses are $p(\Delta_i) = \frac{1}{9} \times (1 - \frac{1}{9})^8 \approx 0.043$. Those of the double fault probabilities are $p(\Delta_j) = (\frac{1}{9})^2 \times (1 - \frac{1}{9})^7 \approx 0.005$. From the fault probabilities of the diagnoses, we map back to statements. Statement 5 is only contained in one diagnosis. Therefore, the fault probability of Statement 5 $p_{pred}(5) = p(\{5\}) \approx 0.043$. Statement 6 is contained in two diagnoses ({6,14} and {6,20}). The fault probability of statement 6 $p_{pred}(6) = p(\{6, 14\}) + p(\{6, 20\}) \approx 0.011$. Table 4.1 indicates the fault probabilities for all statements and the results. The last column of the table shows that the faulty statement is ranked at position 1, but there are 2 other statements with the same ranking. Thus, 3 of 9 statements (33 %) must be investigated.

Table 4.1.: The Bank Account Example - Ranking of the statements based on Shsc. The faulty statement is marked with •.

Line s	$p_{pred}(s)$	$p'_F(s)$	**Ranking**
5	0.043	0.200	1
6	0.011	0.050	6
9	0.011	0.050	6
10	0.011	0.050	6
14	0.022	0.100	4
17	0.043	0.200	1
• 18	0.043	0.200	1
19	0.011	0.050	6
20	0.022	0.100	4

Shsc has two major limitations: First, it is not able to localize faults which are caused by missing code. Second, it always ranks constructor statements high since they are part of every slice.

Spectrum-based fault localization

SFL considers both, passing and failing test cases. Table 4.2 shows the observation matrix obtained when executing the test cases from Listing 4.2 on the bank account example. The rightmost columns show the computed coefficients and the resulting ranking when using the Ochiai coefficient. The faulty statement is ranked at third position together with 3 other statements. In this case, 6 of 9 statements (67 %) must be investigated.

Table 4.2.: The Bank Account Example - The observation matrix, the Ochiai coefficients and the subsequent ranking of the statements. The faulty statement is marked with •.

Line	T1	T2	T3	T4	T5	Coefficient	Ranking
5	•	•	•	•	•	0.447	9
6	•	•	•	•	•	0.447	9
9	•	•		•		0.577	3
10	•	•		•		0.577	3
14	•		•	•	•	0.500	7
17	•			•	•	0.577	3
• 18	•			•	•	0.577	3
19	•			•		0.707	1
20	•			•		0.707	1
Error	•						

The major drawback of SFL is its granularity. The finest granularity can only be a compound statement. This is due to the fact that SFL cannot distinguish statements with identical execution patterns [JAG09]. This drawback is eliminated in SENDYS because of the usage of slices.

Spectrum-enhanced dynamic slicing

SENDYS uses the normalized Ochiai coefficients as initial fault probabilities. Table 4.3 shows the fault probabilities and the resultant ranking. In this case, only 2 of 9 statements (22 %) must be investigated.

4.4.2. The modified Bank Account example

The combined approach performs better than SHSC when the fault is not part of the initialization of the program. One might think that it performs worse when the fault is located in the initialization, since it decreases the probabilities of these parts. However, this is not the case. In order to illustrate

Table 4.3.: The Bank Account Example - Ranking of the statements based on SENDYS. The faulty statement is marked with ●.

Line s	$p_{pred}(s)$	$p'_F(s)$	Ranking
5	0.032	0.163	3
6	0.008	0.043	9
9	0.011	0.054	7
10	0.011	0.054	7
14	0.018	0.092	5
17	0.040	0.205	1
● 18	0.040	0.205	1
19	0.012	0.063	6
20	0.024	0.122	4

a fault in the initialization, we modify our example program from Figure 4.1 in the following way:

```
 5.    this.balance = balance;
18.    if(money>0){
```

Spectrum-based fault localization

Table 4.4 shows the observation matrix, the computed Ochiai coefficients and the subsequent ranking for SFL. The faulty statement is ranked at position 2 together with the other initialization statement. The union of all faulty execution traces comprises 5 statements. In the worst case, 3 of these 5 statements (60 %) must be investigated.

Table 4.4.: The modified Bank Account Example - The observation matrix, the Ochiai coefficients and the statements ranking. The faulty statement is marked with ●.

Line	T1	T2	T3	T4	T5	Coefficient	Ranking
● 5	●	●	●	●	●	0.632	2
6	●	●	●	●	●	0.632	2
14	●		●	●	●	0.707	1
17	●			●	●	0.408	4
18	●			●	●	0.408	4
Error	●		●				

SHSC and SENDYS

For computing the ranking of the statements with the SHSC and SENDYS, the dynamic slices for the test cases `T1` ($S_{a1.balance} = \{5, 17, 18\}$) and `T3` ($S_{a2.balance} = \{5, 14\}$) are computed. These slices result in 3 minimal diagnoses ({5}, {14,17} and {14,18}). Tables 4.5 and 4.6 summarize the results of SHSC and SENDYS. The faulty statement is ranked at position 1 for both approaches. This example demonstrates that the combined approach is able to detect faults in initialization statements with the same accuracy as the original SHSC approach.

Table 4.5.: The modified Bank Account Example - Ranking of the statements based on SHSC. The faulty statement is marked with •.

Line s	$p'_F(s)$	**Ranking**
• 5	0,429	1
6	0,000	5
14	0,286	2
17	0,143	3
18	0,143	3

Table 4.6.: The modified Bank Account Example - Ranking of the statements based on SENDYS. The faulty statement is marked with •.

Line s	$p'_F(s)$	**Ranking**
• 5	0.476	1
6	0.000	5
14	0.262	2
17	0.131	3
18	0.131	3

4.5. Empirical evaluation

The empirical evaluation of SENDYS consists of three major parts: (1) the comparison of SENDYS with SFL and SHSC (Section 4.5.1), (2) the influence of different similarity coefficients on the fault localization quality (Section 4.5.2), and (3) the comparison of SENDYS with other state-of-the-art debugging approaches (Section 4.5.3). Before presenting the results, we explain the prototype implementation of SENDYS and introduce the tested programs by quantitatively and qualitatively describing them.

Our implementation of SENDYS works with Java programs and JUnit test cases. It utilizes the JavaSlicer[1] for obtaining execution traces and dynamic slices. The spectra information is obtained from the execution traces. Besides the fact that the used slicer is only a dynamic slicer and not a relevant slicer, the slicer has some weaknesses [Ham08], which influence the obtained results: (1) Data dependencies can vanish when a method is called by reflection or when native code is executed. (2) Due to the restriction to dynamic slices, it is possible that the real fault is not part of a slice. This is why we have to exclude faults leading to incomplete slices from the case study. However, we are still able to proof our concept.

We investigated the 8 programs listed in Table 4.7. BankAccount, Mid, and StaticExample are toy examples. The BankAccount is the demo example from Section 4.4. Mid and StaticExample are demo programs which compute the medial of three numbers or work with static members and methods. The TrafficLight example is borrowed from the JADE (Java Diagnosis Engine) project[2] and simulates the different phases of a traffic light. Since this program does not have any JUnit test cases, we created our own. ATMS is an Assumption-based Truth Maintenance System. The source code of the previously mentioned programs is publicly available[3]. ReflectionVisitor is a Java-implementation of the Visitor-Pattern. JTOPAS is a text parser and is taken from the Software Infrastructure Repository [DER05]. TCAS is a Java Implementation of the traffic collision avoidance [Hut+94] system from the Siemens Set.

It was not possible to compute the possible fault locations for all available faulty program versions because of the following three reasons. (1) No slices can be computed for test cases which produce endless loops. (2) The used slicer is not able to compute correct slices for all faulty program versions due to the previously discussed limitations. (3) JTOPAS includes predefined faults which are not detected by the available test cases. We excluded program versions from our case study for which any of these cases applies. Finally, 42 program versions remained for the empirical evaluation. Table 4.7 gives an overview of the number of faulty program versions used in the evaluation (see column 'Faults'). The first number in the brackets indicates the number of program variants which were excluded from the case study because there were no failing test cases. The second number indicates the number of variants which were excluded because of endless loops or limitations in the slicer. The third number in the brackets indicates the number of program variants that were used in the evaluation.

[1]http://www.st.cs.uni-saarland.de/javaslicer/
[2]http://www.dbai.tuwien.ac.at/proj/Jade/
[3]http://dl.dropbox.com/u/38372651/Debugging/EP.zip

Table 4.7.: Description of the investigated programs including the the Non Commenting Source Statements (NCSS), the number of test cases (TC) and the number of investigated faults, which is tripartite: faults with no failing test cases, excluded faults and investigated faults.

Program	NCSS	TC	Faults
BankAccount	17	5	2(- / - / 2)
Mid	17	8	1(- / - / 1)
StaticExample	16	8	1(- / - / 1)
TrafficLight	33	7	2(- / - / 2)
ATMS	1573	14	3(- / 1 / 2)
ReflectionVisitor	338	14	5(- / - / 5)
JTOPAS - Version 1	1368	127	8(4 / 3 / 1)
JTOPAS - Version 2	1485	115	12(11/ - / 1)
JTOPAS - Version 2	3931	183	14(7 / 4 / 3)
TCAS	77	1545	39(- /15/24)

In the following, we use two different metrics: SCORE$_{EXEC}$ and SCORE$_{LOC}$. Equations 4.1 and 4.2 illustrate these metrics. Both metrics use the number of statements that must be investigated until the first statement with a bug is reached (*invest*). The SCORE$_{EXEC}$ metric is the ratio of *invest* and the number of executed statements (*exec*). In contrast, the SCORE$_{LOC}$ metric indicates the ratio of *invest* and the total number of statements in the program (*total*).

$$\text{SCORE}_{\text{EXEC}} = \frac{invest}{exec} \cdot 100\% \tag{4.1}$$

$$\text{SCORE}_{\text{LOC}} = \frac{invest}{total} \cdot 100\% \tag{4.2}$$

Please note that we always indicate the worst case scenario: If the faulty statement is ranked at position 14 together with two other statements, the number of statements that are investigated (*invest*) is 16. In case of a best case scenario it would be 14.

4.5.1. SENDYS vs. basic approaches

When comparing the fault localization capabilities of SENDYS with those of the basic approaches, i.e. SHSC and SFL, it turns out that SENDYS leads to huge savings in the number of statements that must be investigated. Table 4.8 opposes the fault localization capabilities of SENDYS to those of SFL and SHSC. The FaultID is the unique identification number of the fault. In case of JTOPAS, this number consists of the version number and the fault number. It

Table 4.8.: Comparison of the number of statements that must be investigated when using SHSC, SFL (with Ochiai as coefficient) and SENDYS. In addition, the table shows how many statements were executed in total (column 'Stmt') in the failing test cases.

Program	FaultID	Stmt	SHSC	SFL	SENDYS
Bank Account	1	9	3	6	2
	2	5	1	3	1
Mid	1	6	5	1	1
StaticExample	1	5	2	2	1
TrafficLight	1	25	14	2	1
	2	17	9	17	9
ATMS	1	318	226	91	46
	2	307	188	55	28
ReflectionVisitor	1	35	17	20	17
	2	47	39	7	11
	3	119	43	24	15
	4	65	46	34	41
	5	48	42	5	5
JTOPAS	1_2	16	3	9	3
	2_3	41	7	3	1
	3_6	630	258	3	2
	3_7	665	253	36	9
	3_9	686	389	77	31
TCAS	1	25	21	3	2
	2	26	18	14	14
	3	29	18	24	17
	4	25	21	3	3
	5	29	16	22	15
	6	24	20	5	5
	9	26	22	18	13
	12	29	16	24	16
	15	29	16	22	14
	20	25	21	17	10
	21	25	21	17	15
	22	26	22	18	17
	23	26	22	18	18
	24	26	22	18	17
	25	26	22	2	2
	26	29	17	25	15
	28	30	15	14	11
	29	27	19	16	8
	30	26	18	14	8
	31	24	20	5	5
	32	25	21	5	5
	34	29	29	24	29
	35	30	15	14	9
	37	29	19	2	1

is obvious that the combined approach performs at least as good as the best of the two basic approaches. In 25 cases, the combined approach improves the fault localization precision. In 14 cases, SENDYS performs as good as the better of the original approaches. Only in three cases it performs worse than SFL. Figure 4.2 pairwise compares the fault localization capabilities of SENDYS, SFL and SHSC. Therefore, the data from Table 4.8 was normalized to the number of executed statements (SCORE$_{\text{EXEC}}$ metric). From Figure 4.2a, we cannot determine whether SFL or SHSC performs better. However, the Figures 4.2b and 4.2c clearly show that SENDYS performs better than SFL and SHSC.

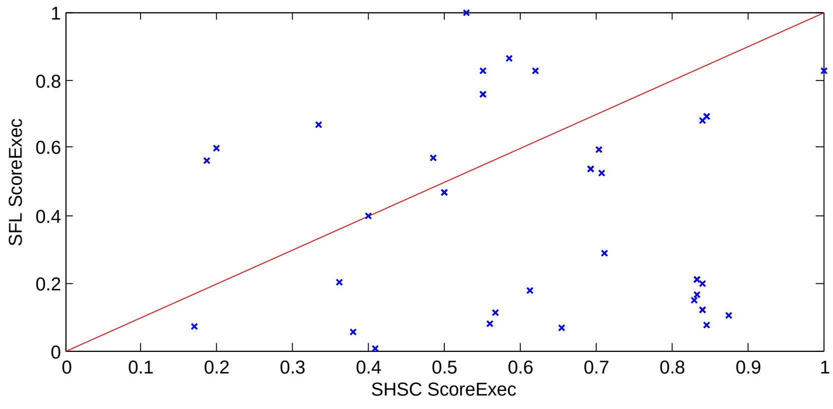
Table 4.9 summarizes the results from Table 4.8. It compares the overall effectiveness of SENDYS to those of the basic approaches with respect to the median, the mean value and the standard deviation of number of statements that must be investigated in order to find the bug. Since the investigated programs considerably differ in size, we have normalized the raw values before computing the median, mean value and standard deviation: For each fault, instead of using the absolute number, we used the ratio of the statements that must be investigated and the total number of statements that were executed in the failing test cases in that program version (SCORE$_{\text{EXEC}}$ metric). From this table, it is obvious that SENDYS improves the fault localization capabilities of SHSC by 50 % and those of SFL by 25 %.

Table 4.9.: Median, mean value and standard deviation of the SCORE$_{\text{EXEC}}$ metrics for SHSC, SFL (using Ochiai as coefficient) and SENDYS in %. Lower scores are preferable.

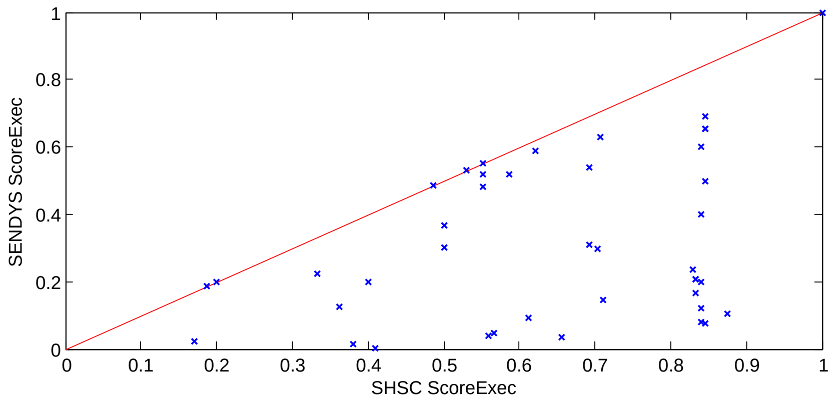
	SHSC	**SFL**	**SENDYS**
median	67.37	49.49	22.81
mean	63.95	43.65	31.76
stdev	21.23	28.99	23.97

Figure 4.3 graphically compares the fault localization capabilities of SENDYS, SFL and SHSC in terms of the amount of code that must be investigated in order to find the faulty statement. The x-axis represents the percentage of code that is investigated (SCORE$_{\text{EXEC}}$ metric). The y-axis represents the percentage of faults that are localized within that amount of code. This figure reads as follows: If you investigate the top 40 % ranked statements of the 42 investigated faulty program versions, SENDYS contains the faulty statement for 65 % of the program versions. SFL only contains the faulty statement for 45 % of the program versions, and SHSC for 18 % of the program versions. It can be seen that SENDYS allows to detect faults earlier than with the basic approaches.

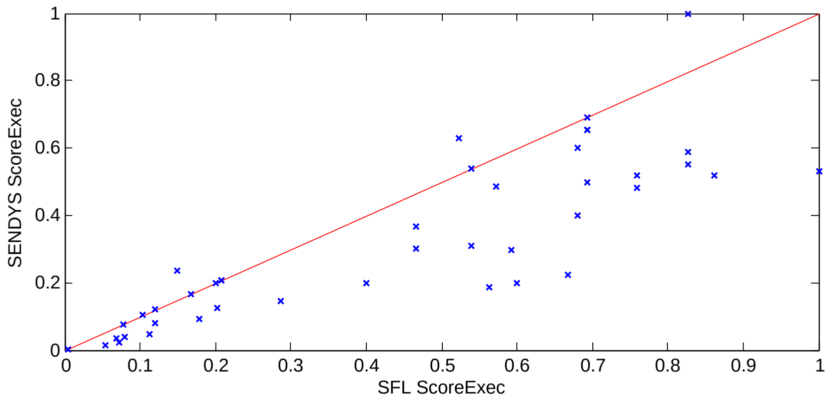
The computational overhead of SENDYS is marginal compared to the basic approaches. In this evaluation, the execution of the test cases absorbs the major



(a) SHSC versus SFL



(b) SHSC versus SENDYS



(c) SFL versus SENDYS

Figure 4.2.: Pairwise comparison of the fault localization capabilities of SENDYS, SHSC and SFL (with Ochiai as coefficient) in terms of the $\text{SCORE}_{\text{EXEC}}$ metric. Data points close to the red line indicate that the approaches perform equal. Data points below that line indicate that the approach labeled on the y-axis performs better.

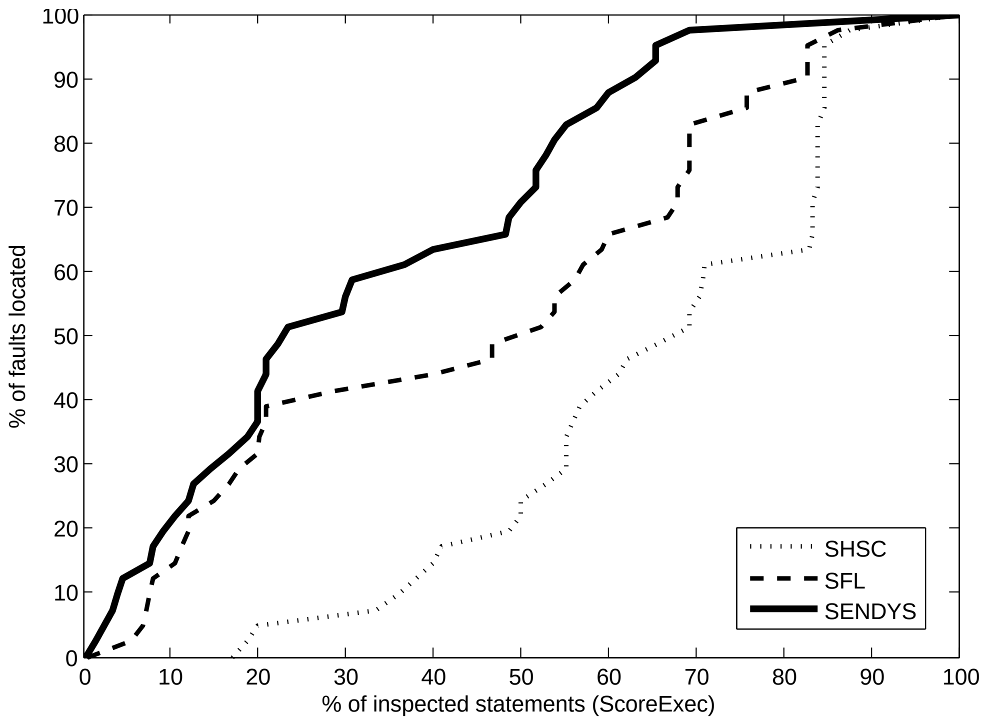


Figure 4.3.: Comparison of Sendys with Shsc and Sfl (with Ochiai as coefficient) in terms of the amount of code that must be investigated.

part of the total computation time. For the larger programs (Atms, JTopas, ReflectionVisitor and Tcas), the spectra creation and coefficients computation requires approximately 10 % of the time required for the execution. The computation time of the slices and hitting sets ranges from 10 % to 25 % of the execution time. The computations of Sendys (Slice and spectra computation, hitting sets, fault probabilities) account for 20 % to 35 % of the total execution time.

4.5.2. Comparison of different similarity coefficients

So far, our experiments have been performed using the Ochiai coefficient. We have chosen the Ochiai coefficient since studies [AZG06; Abr+09b] have shown that Ochai delivers better results than other similarity coefficients. However, Sendys is able to work with other coefficients as well. Table 4.10 shows the amount of statements that must be investigated in percentage of the executed statements ($\text{Score}_{\text{exec}}$) when using different similarity coefficients. This table affirms that Ochiai performs better than Tarantula [JH05] and Jaccard [Zoe+07]. In addition, it shows that using Sendys improves the fault localization capabilities. Figure 4.4 illustrates the fault localization capabilities of Sendys with different similarity coefficients. The figure compares Sendys based on the different similarity coefficients with the basic coefficients in terms of the amount of code that must be investigated in order to find the faulty statement.

Table 4.10.: Average percentage of statements that must be investigated in order to find the faulty one on basis of the total number of executed statements ($\text{Score}_{\text{exec}}$) for different similarity coefficients.

	Ochiai	Tarantula	Jaccard
Standalone	43.5	52.3	46.0
Part of Sendys	31.7	33.7	34.8

4.5.3. Comparison with other approaches

Similar to our approach, Barinel and Deputo are approaches that combine Sfl with Mbsd. Therefore, we compare the fault localization capabilities of Sendys with those of Deputo and Barinel. The following evaluation is performed on Tcas from the Siemens Set [DER05]. The prototypes of Deputo and Barinel work with programs written in C. The prototype of Sendys is implemented for Java programs. Therefore, the original C version of Tcas is used for evaluating Deputo and Barinel and a Java implementation for evaluating Sendys. The C version comprises 105 Non Commenting Source

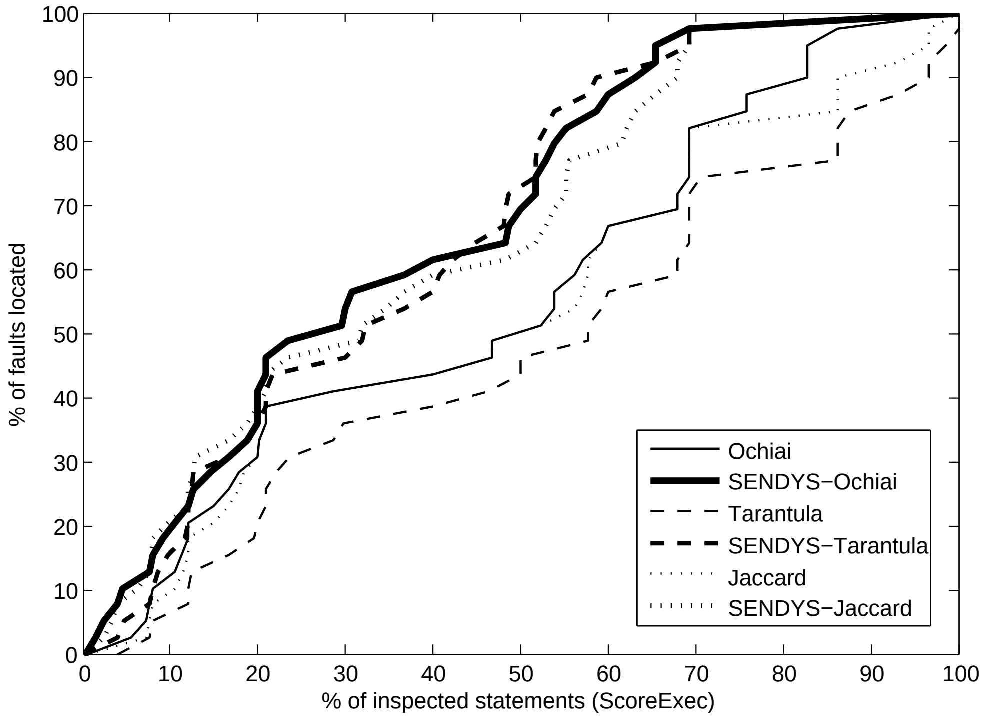


Figure 4.4.: Comparison of Sendys with different similarity coefficients in terms of the amount of code that must be investigated.

Statements (NCSS) and 1608 test cases. The Java version comprises 77 NCSS and 1545 test cases. Both program variants have the same faulty program versions. A fault consists of one to three faulty code lines. Because of the different numbers of NCSS, we make use of the SCORE$_{LOC}$ metric to compare the three fault localization approaches.

Table 4.11 shows the SCORE$_{LOC}$ for SFL, MBSD, BARINEL, DEPUTO, SHSC, and SENDYS for some of the faulty program versions of TCAS. This table only shows the results for those faulty TCAS variants where all approaches where able to produce results. The table shows that on average, the combined approaches perform better than their basic approaches. DEPUTO has the best fault localization capabilities with an average SCORE$_{LOC}$ of 5.55 %. The basic approaches SFL (SCORE$_{LOC}$: 17.27 %) and SHSC (SCORE$_{LOC}$: 25.58 %) perform worst. SHSC is a low level variant of MBSD. MBSD is in general known to be computational complex. However, SHSC comes with low computational costs, but is less precise than other MBSD techniques. When using MBSD, on average only 11.30 % of the source code must be investigated until the fault location is found. However, DEPUTO and MBSD are only suited for small programs, because they do not scale to large software systems. Due to their (comparable) small time/space complexity, BARINEL and SENDYS are alternatives for debugging larger systems. BARINEL has an average SCORE$_{LOC}$ of 17.10 %. SENDYS has an average SCORE$_{LOC}$ of 14.74 %.

Figure 4.5 gives a graphical overview of the fault localization capabilities of the discussed approaches. It plots the percentage of located faults in terms of the percentage of inspected code (i.e., effort to find the root cause). From this figure, it can be observed that the combined approaches largely outperform their basic approaches.

4.6. Conclusion

SENDYS (short for Spectrum-ENhanced DYnamic Slicing) combines SFL with slicing-hitting-set-computation (SHSC). The approach solves some disadvantages of SFL and SHSC that occur when applying them individually. We discussed the SENDYS approach in detail and compared its outcome with the individual approaches in an empirical study. This empirical study indicates that the combined approach outperforms SFL and SHSC. SENDYS provides an improved ranking of fault candidates. Thus, SENDYS is a valuable aid for programmers when debugging. In particular, SENDYS improves the fault localization capabilities of SHSC by 50 % and those of SFL by 25 %. A handicap of the basic approaches as well as of SENDYS and many other debugging techniques is the lack of the ability to advise the programmer that the fault

Table 4.11.: SCORE$_{LOC}$ in [%] for SFL, MBSD, BARINEL, DEPUTO, SHSC, and SENDYS for the faulty program versions of TCAS. Ochiai was used as similarity coefficient.

Fault	SFL	MBSD	BARINEL	DEPUTO	SHSC	SENDYS
1	0.95	19.05	0.95	0.95	27.27	2.60
2	4.76	4.76	4.76	3.81	23.38	18.18
3	4.76	12.38	4.76	3.81	23.38	22.08
4	39.05	19.05	39.05	10.48	27.27	3.90
5	39.05	9.52	39.05	6.67	20.78	19.48
6	13.33	18.10	13.33	13.33	25.97	6.49
9	39.05	10.48	39.05	10.48	28.57	16.88
12	12.38	8.57	13.33	4.76	20.78	20.78
15	18.10	9.52	18.10	5.71	20.78	18.18
20	36.19	19.05	33.33	6.67	27.27	12.99
21	12.38	19.05	9.52	9.52	27.27	19.48
22	14.29	8.57	14.29	8.57	28.57	22.08
23	39.05	10.48	39.05	10.48	28.57	23.38
24	12.38	18.10	13.33	12.38	28.57	22.08
25	2.86	9.52	2.86	2.86	28.57	2.60
26	2.86	10.48	2.86	2.86	22.08	19.48
28	39.05	1.90	39.05	1.90	19.48	14.29
29	12.38	2.86	9.52	2.86	24.68	10.39
31	0.95	17.14	0.95	0.95	25.97	6.49
32	0.95	15.24	0.95	0.95	27.27	6.49
34	0.95	9.52	0.95	0.95	37.66	37.66
35	15.24	1.90	15.24	1.90	19.48	11.69
37	36.19	4.76	39.05	4.76	24.68	1.30
Avg.	**17.27**	**11.30**	**17.10**	**5.55**	**25.58**	**14.74**

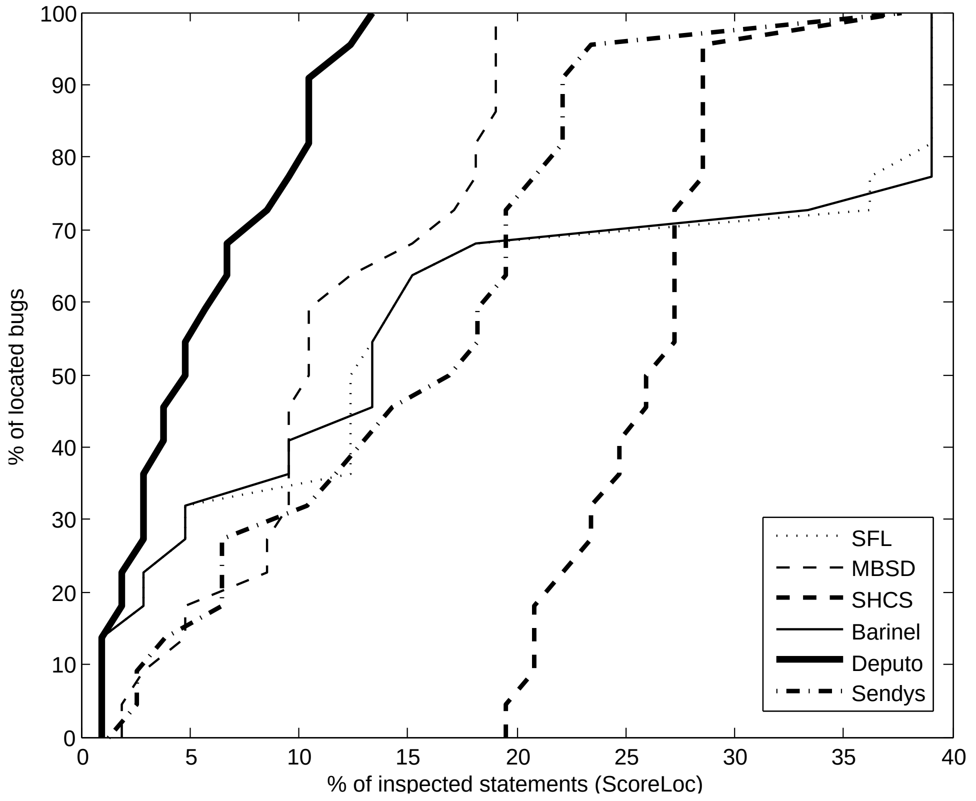


Figure 4.5.: Comparison of Deputo, Barinel, and Sendys with their basic approaches in terms of the amount of code that must be investigated.

might be caused by missing code. Similar to the original approaches, the introduced approach highly depends on the quality of the test suite.

In addition, we compared different coefficients used for ranking statements. The empirical results show that Ochiai is performing best compared to Tarantula and Jaccard. Furthermore, we showed that using Sendys improves the fault localization quality independently of the used similarity coefficient. In the third part of the empirical evaluation, we compared Sendys with Deputo and Barinel. Deputo outperforms Sendys, but is only applicable to small programs. Therefore, Sendys is a good alternative to Deputo since Sendys scales to large programs and performs slightly better than Barinel. However, a general recommendation for preferring Sendys over Barinel is not given, since the evaluation basis is too small.

5. Constraint-Based Slicing

This chapter is based on the work published in [HW12b] and [HW12a].

5.1. Introduction

CONBAS (short for CONstraint BAsed Slicing) is a method that increases the precision of automated fault localization by improving existing program slicing techniques. Dynamic slicing and its variations significantly reduce the size of slices. However, slices could still be too large to be of practical use. Large slices are obviously not a good help when debugging. Therefore, we focus on providing a methodology that allows for reducing the size of dynamic slices. Reducing the size of dynamic slices improves the debugging capabilities of dynamic slicing.

In the following, we make use of the running example illustrated in Figure 5.1, the TASTE example, in order to demonstrate our approach. The TASTE example is borrowed from the Unravel project[1]. The method `getTastes` takes four Integer values representing colors as input and returns an Integer array containing tastes. There is a fault in Line 3: Instead of `red=red*2;` the expression `red=red*5;` would be correct. In order to reveal the bug, we have used the test case defined by Gupta *et al.* [Gup+05]:

```
Input:           getTastes(1,5,8,2)
Expected Output: result = {49,40,40,7}
```

Figure 5.1 shows the resultant execution trace when applying the test case on the TASTE example.

The inserted fault causes the computation of wrong values for the variables `bitter`, `sweet` and `sour`. We trace back the data and control dependencies in order to compute the possible root causes. The resultant slices are:

$$\begin{aligned} S_{\text{bitter}} &= \{T,3,5,6,7,8,9,12,13,14\} \\ S_{\text{sweet}} &= \{T,3,4,14\} \\ S_{\text{sour}} &= \{T,3,5,6,7,8,9,14\} \end{aligned}$$

[1] http://hissa.nist.gov/unravel/

```
 1 public int[] getTastes(int red, int blue,
       int green, int yellow){
 2   int sweet, sour, salty, bitter;
 3   red = red * 2; //Error: red = red * 5
 4   sweet = red * green;
 5   sour = 0;
 6   int i = 0;
 7   while(i < red) {
 8     sour = sour + green;
 9     i = i + 1;
10   }
11   salty = blue + yellow;
12   yellow = sour + 1;
13   bitter = yellow + green;
14   return {bitter, sweet, sour, salty};
15 }
```

Listing 5.1: The TASTE example. There is a bug in Line 3.

N	Statements
T^1	`getTastes(1,5,8,2)`
3^2	`red = red * 2`
4^3	`sweet = red * green`
5^4	`sour = 0`
6^5	`i = 0`
7^6	`while (i < red)`
8^7	`sour = sour + green`
9^8	`i = i + 1`
7^9	`while (i < red)`
8^{10}	`sour = sour + green`
9^{11}	`i = i + 1`
7^{12}	`while (i < red)`
11^{13}	`salty = blue + yellow`
12^{14}	`yellow = sour + 1`
13^{15}	`bitter = yellow + green`
14^{16}	`return {bitter,sweet,sour,salty}`

Figure 5.1.: An execution trace for the TASTE example. The column N indicates which of the executed statements are the test case statement (marked with T) and which are part of the tested code (marked with the corresponding source code line number). The exponent consecutively numbers the statements in order of their execution.

The intersection of all slices only comprises the statements from Line 3, Line 14, and the test case statement T. However, computing the intersection may not be a good idea in case of multiple faults. Some of the faulty statements might be absent in some slices. Since the intersection only contains statements that are part of all slices, the faulty statements might be missing in the intersection. In particular, this is the case if there are at least two faults which influence the output of different variables.

The union of all slices comprises all statements of the execution trace except the statement in Line 11. This reduction is too small to be a valuable help for a programmer when debugging. In short, whereas the intersection of all slices is too restrictive, the union of all slices allows too many statements. Hence, improving the slicing result is highly needed.

How is it possible to reduce the size of a slice without loosing its fault localization capabilities? The answer to this question is motivated by the following three observations: (1) Using data and control dependencies reduces the number of potential root causes. Statements that have no influence on faulty variables can be ignored. (2) During debugging, programmers make assumptions about the correctness or in-correctness of statements. From these assumptions, they try to predict a certain behavior, which should not be in contradiction with the expectations. (3) Backward reasoning, i.e., deriving values for intermediate variables from other variables, is essential to further reduce the number of fault candidates. For this purpose, statements are not interpreted as functions that change the state of the program but as equations. This kind of interpretation allows us to derive the input value from output values. Further reductions of fault candidates can be obtained when considering alternative execution paths. However, considering alternative execution paths is computationally more demanding than considering the execution trace of a failing test case only. Because of this computational complexity, alternative execution paths are not considered in CONBAS.

CONBAS contributes to the field of debugging by improving dynamic slicing with respect to the computed number of bug candidates. CONBAS is not designed as standalone debugging method, but rather as part or pre-processing step used in other debugging techniques. The remainder of this chapter is organized as follows: We compare CONBAS to related work in Section 5.2. We formalize the approach in Chapter 5.3 and explain the usage of the algorithm by means of a running example in Section 5.4. In the empirical evaluation (Section 5.5), we show that CONBAS is able to reduce the size of slices for programs with single and double faults without losing the fault localization capabilities. CONBAS achieves a 28 % reduction of slice sizes compared to dynamic slicing. In Section 5.6, we discuss the benefits and limitations of the approach as well as future work and conclude.

5.2. Related Research

In principle, CONBAS is based on [Wot11]. In constrast to [Wot11], CONBAS does not ignore control flow statements. This allows CONBAS to reason about diagnoses affecting the control flow of a program. In [Wot11], bugs that change only the control flow but do not directly modify the values of any variables are not in the set of the remaining diagnoses. This conceptional flaw is eliminated in CONBAS.

For retrieving the formal representation of a program Π, we basically rely on the work of Nica *et al.* [NNW12; WNM12]. However, CONBAS and [NNW12; WNM12] differ in two major aspects: (1) Nica *et al.* represent all possible execution paths up to a specified size as constraints. In contrast, CONBAS only uses the current execution path. It is not necessary to explicitly unroll loops. As a consequence, the representation becomes smaller and the modeling easier. On the contrary, we loose information of the program and we are not able to eliminate candidates that impact the execution path. Even though, CONBAS cannot match with results from Nica *et al.*, our approach executes faster. (2) We compute diagnoses via the hitting set algorithm and only use a constraint solver to check whether a solution can be found. In contrast, Nica *et al.* use a constraint solver to obtain the diagnoses directly.

Gupta *et al.* [Gup+05] also developed an approach to reduce the size of slices. Their approach combines delta debugging with forward and backward slices for computing the failure-inducing chop. This technique requires a test oracle. In contrast, CONBAS does not require an oracle because no additional test cases are created.

Zhang *et al.* [ZGG06] reduce the size of dynamic slices via confidence values. Similar to CONBAS, their approach requires only one failing execution trace. However, their approach requires one output variable with a wrong value and several output variables where the computed output is correct. In contrast, CONBAS requires at least one output variable with a wrong value, but no correctly computed output variables.

Jeffrey *et al.* [JGG08] identify potential faulty statements via value replacement. They systematically replace the values used in statements so that the computed output becomes correct. The original value and the new value are stored as an Interesting Value Mapping Pair (IVMP). They state that IVMPs typically occur at faulty statements or statements that are directly linked via data dependencies to faulty statements. They limit the search space for the value replacement to values used in other test cases. We do not limit the search space to values used in other test cases. Instead, our constraint solver determines if there exist any values for the variables in an abnormal statement so that the correct values for the test cases can be computed. On the one hand, our approach is computationally more expensive, but on the other hand it does not depend on the quality of other test cases. As Jeffrey *et al.* stated, the presence of multiple faults can diminish the effectiveness of the value replacement approach. In contrast, the CONBAS approach is designed for handling multiple faults.

5.3. The CONBAS algorithm

The basic idea of CONBAS is to reduce the size of summary slices. Therefore, the minimal diagnoses are computed and further reduced with the aid of a constraint solver. The execution trace of a failing test case is converted into constraints. The constraint solver checks for satisfiability of the converted execution trace assuming that the statements of a diagnosis are incorrect. If the constraint system is not satisfiable, the diagnosis is rejected. Otherwise, the diagnosis is presented to the programmer. Figure 5.2 gives an overview of the CONBAS approach.

Algorithm 5.1 explains the CONBAS approach in detail. In Line 1, the test case t is executed on a program Π using the function RUN(Π, t). This function returns the resulting execution trace ET and the set of conflicting variables CV. In the Lines 2 to 6, the relevant slices are computed for all conflicting variables CV by means of the function RELEVANTSLICE(ET, Π, x, n) (see Algorithm 3.1), at which $|\text{ET}|$ is established for n. The relevant slices are stored in the conflict set CO, which is a set of sets. In Line 7, the minimal hitting sets (see Definition 15 and Algorithm 3.5) of the set of slices are computed. In the Lines 8 to 14, the relevant slices SC_c are computed for all conditional statements c in ET. The function POSITIONINEXECUTIONTRACE(c, ET) is used to obtain the line number of c in the execution trace.

Some bugs cause a wrong evaluation of a condition and thus lead to the non-execution of statements. In order to handle such bugs, the function EXTENDCONTROLSTATEMENTS(ET, Π) in Line 15 adds a small overhead to each control statement c in the execution trace ET: For each variable v that could be redefined in any branch of c, the statement v=v is added to the execution trace ET. These additional statements are inserted after all statements that are control-dependent on c. The inserted statements will be referenced by the line number of c when calling the function INDEX in Algorithm 3.3. The returned execution trace is assigned to ET_C. With this extension, we are able to reason over faults that cause the non-execution of statements. This extension can be compared with potential data dependencies in relevant slicing.

In Line 16, the function SSA(ET_C) (see Algorithm 3.2) transforms the execution trace ET_C into its single static assignment form and also delivers the largest

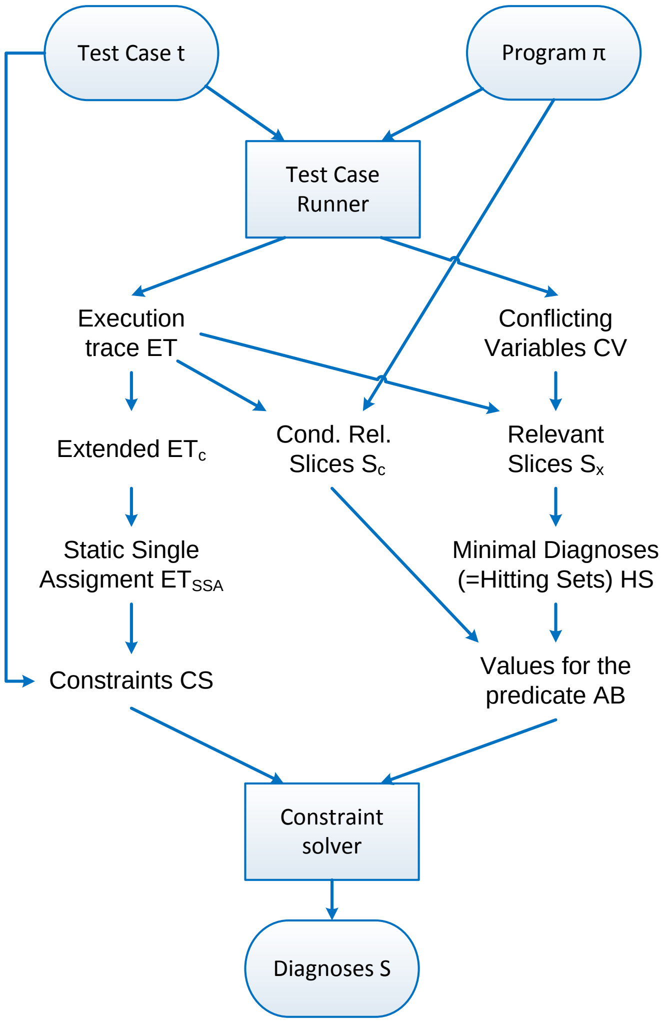


Figure 5.2.: The ConBas approach

Algorithm 5.1 CONBAS(Π, t)

```
Require: Program $\Pi$ and failing test case $t$
Ensure: Dynamic slice $S$
 1: $[\mathrm{ET}, \mathrm{CV}] = \textsc{Run}(\Pi, t)$
 2: Conflict set $\mathrm{CO} = \{\}$
 3: for all $x \in \mathrm{CV}$ do
 4:     $S_x = \textsc{RelevantSlice}(\mathrm{ET}, \Pi, x, |\mathrm{ET}|)$
 5:     $\mathrm{CO} = \mathrm{CO} \cup \{S_x\}$
 6: end for
 7: $\mathrm{HS} = \textsc{MinHittingSets}(\mathrm{CO}, |\Pi|)$
 8: for all $c \in \mathrm{ET}$ do
 9:     $n = \textsc{PositionInExecutionTrace}(c, \mathrm{ET})$
10:     for all variables $x$ in $c$ do
11:         $\mathrm{SC}_x = \textsc{RelevantSlice}(\mathrm{ET}, \Pi, x, n)$
12:     end for
13:     $\mathrm{SC}_c = \cup\, \mathrm{SC}_x$
14: end for
15: $\mathrm{ET}_C = \textsc{ExtendControlStatements}(\mathrm{ET}, \Pi)$
16: $[\mathrm{ET}_{\mathrm{SSA}}, \textsc{IndexSsa}] = \mathrm{SSA}(\mathrm{ET}_C)$
17: $\mathrm{CS} = \textsc{Constraints}(\mathrm{ET}_{\mathrm{SSA}}, t, \textsc{IndexSsa})$
18: $S = \{\}$
19: for all $d \in \mathrm{HS}$ do
20:     $\forall i \in d : \mathrm{AB}(i) = \mathrm{true}$
21:     $\forall i \notin d : \mathrm{AB}(i) = \mathrm{false}$
22:     $\forall c \textit{ where } \mathrm{SC}_c \cap d \neq \{\} : \mathrm{AB}(c) = \mathrm{true}$
23:     if $\textsc{ConstraintSolver}(\mathrm{CS} \cup \mathrm{AB})$ has solution then
24:         $S = S \cup \{d\}$
25:     end if
26: end for
27: return $S$
```

index value for each variable used in the SSA form. In Line 17, the function CONSTRAINTS(ET$_{\text{SSA}}$,t,INDEXSSA) (see Algorithm 3.3) converts each statement into its equivalent constraint representation. The statements added in the function EXTENDCONTROLSTATEMENTS(ET, Π) (v=v, in SSA form: $v_{i+1} = v_i$) are concatenated with the predicate AB(j): AB(j) $\vee$ $v_{i+1} = v_i$ where j is the index of the conditional statement.

The result set S represents the set of possible faulty statements. Initially, the result set is the empty set. For all minimal hitting sets d in the set of minimal hitting sets HS, we check if the constraint solver is able to find a solution. For this purpose, we set all $\mathrm{AB}(i)$ to false except those where the corresponding statements are contained in d. For all conditional statements c where SC_c and d have at least one common element, we set $\mathrm{AB}(c)$ to true. The function CONSTRAINTSOLVER$(\mathrm{CS} \cup \mathrm{AB})$ calls a constraint solver and returns true if the constraint solver is able to find a solution. If a solution is found, we add all elements of d to the result set S (Lines 19 to 26). Finally, we return the set S containing all valid diagnoses (Line 27).

Algorithm CONBAS terminates if the program Π terminates when executing test case t. The computational complexity of CONBAS is determined by the computation of the relevant slices, the hitting sets, and solving of the CSP. Computing relevant slices only adds a small overhead compared to the execution of the program. Hitting set computation and constraint solving are exponential in the worst case (finite case). In order to reduce the computation time, the computation of hitting sets can be simplified: We only compute hitting sets of the size 1 or 2, i.e., we only compute single and double fault diagnoses. Faults with more involved faulty statements are unlikely in practice. Only in cases where the single and double fault diagnoses cannot explain an observed misbehavior, the size of the hitting sets is increased. The complexity of the CSP is determined by its hypertree width [Wot+09]. However, Wotawa *et al.* [Wot+09] have shown that there exists no constant upper-bound for the hypertree width of arbitrary programs.

5.4. Example of application

We demonstrate the functioning of CONBAS by means of the running example from Listing 5.1. The execution trace ET and the slices are shown in Section 5.1. The set of minimal hitting sets HS are computed from the slices $\widehat{S}$:

$$\mathrm{HS} = \{\{3\},\{14\},\{4,5\},\{4,6\},\{4,7\},\{4,8\},\{4,9\}\}$$

The slice for the conditional statement in Line 7 is:

$$\mathrm{SC}_{(i<\mathrm{red})} = \{3,6,9\}$$

The execution trace is extended by the conditional statement conversion. In addition, the method call `getTastes(1,5,8,2)` is split into four assignment statements. Afterwards, the extended execution trace is converted into its static single assignment form. Figure 5.3 shows the extended and converted execution trace.

N	SSA converted statements	ET$_C$
T	$\text{red}_0 = 1$	
T	$\text{blue}_0 = 5$	
T	$\text{green}_0 = 8$	
T	$\text{yellow}_0 = 2$	
3	$\text{red}_1 = \text{red}_0 * 2$	
4	$\text{sweet}_0 = \text{red}_1 * \text{green}_0$	
5	$\text{sour}_0 = 0$	
6	$i_0 = 0$	
7	$\text{while}(i_0 < \text{red}_1)$	
8	$\text{sour}_1 = \text{sour}_0 + \text{green}_0$	
9	$i_1 = i_0 + 1$	
7	$\text{while}(i_1 < \text{red}_1)$	
8	$\text{sour}_2 = \text{sour}_1 + \text{green}_0$	
9	$i_2 = i_1 + 1$	
7	$\text{while}(i_2 < \text{red}_1)$	
7	$\text{sour}_3 = \text{sour}_2$	●
7	$i_3 = i_2$	●
11	$\text{salty}_0 = \text{blue}_0 + \text{yellow}_0$	
12	$\text{yellow}_1 = \text{sour}_3 + 1$	
13	$\text{bitter}_0 = \text{yellow}_1 + \text{green}_0$	
14	$\text{return}\{\text{bitter}_0, \text{sweet}_0, \text{sour}_3, \text{salty}_0\}$	

Figure 5.3.: An extended, SSA converted execution trace for the TASTE example. The column N indicates which of the executed statements are test case statements (marked with T) and which are part of the tested code (marked with the corresponding source code line number). The statements marked with a ● are those statements that are added through the function EXTENDCONTROLSTATEMENTS(ET, Π).

The resultant SSA execution trace is converted into constraints. All statements are concatenated with their AB$_i$ variable through a logical or. In addition, the expected test output is added. After the conversion, we obtain the set of equations illustrated in Figure 5.4.

Finally, the abnormal variables AB$_i$ are systematically set to true. For each minimal hitting set d, a constraint solver is called with the previously computed constraints and the current AB variable configuration. The following

N	**Constraints**
T	$\text{red}_0 == 1$
T	$\text{blue}_0 == 5$
T	$\text{green}_0 == 8$
T	$\text{yellow}_0 == 2$
3	$AB_3 \vee \text{red}_1 == \text{red}_0 * 3$
4	$AB_4 \vee \text{sweet}_0 == \text{red}_1 * \text{green}_0$
5	$AB_5 \vee \text{sour}_0 == 0$
6	$AB_6 \vee i_0 == 0$
8	$AB_8 \vee \text{sour}_1 == \text{sour}_0 + \text{green}_0$
9	$AB_9 \vee i_1 == i_0 + 1$
8	$AB_8 \vee \text{sour}_2 == \text{sour}_1 + \text{green}_0$
9	$AB_9 \vee i_2 == i_1 + 1$
7	$AB_7 \vee \text{sour}_3 == \text{sour}_2$
7	$AB_7 \vee i_3 == i_2$
12	$AB_{12} \vee \text{yellow}_1 == \text{sour}_3 + 1$
13	$AB_{13} \vee \text{bitter}_0 == \text{yellow}_1 + \text{green}_0$
T	$\text{bitter}_0 == 49$
T	$\text{sweet}_0 == 40$
T	$\text{sour}_3 == 40$

Figure 5.4.: The constraint representation of the execution trace for the TASTE example. The column N indicates which of the executed statements are test case statements (marked with T) and which are part of the tested code (marked with the corresponding source code line number).

AB configurations are possible:

$$\{\{3+7\},\{14\},\{4,5\},\{4,6+7\},\{4,7\},\{4,8\},\{4,9+7\}\}$$

All sets d where the constraint solver finds a solution are added to solution set S. Finally, S contains the statements $\{3,4,5,6,7,8,9\}$. The statements in Line 12 and 13 cannot explain the obtained misbehavior.

5.5. Empirical evaluation

The empirical evaluation of Conbas consists of two major parts: (1) We show that Conbas is able to reduce the size of slices without losing the fault localization capabilities of slicing. We show this for single faults as well as for double faults. (2) We investigate the influence of the number of output variables on the reduction result.

We conducted this empirical evaluation using a proof of concept implementation of Conbas. This implementation accepts programs written in the language $\mathcal{L}$ (see Chapter 3). In order to test existing example programs, we have extended the language $\mathcal{L}$ to accept simple Java programs, i.e. Java programs with Integer and Boolean data types only and without method calls and object-orientation. The implementation itself is written in Java and comprises a relevant slicer and an interface to the Minion constraint solver [GJM06]. The evaluation was performed on an Intel Core2 Duo processor (2.67 GHz) with 4 GB RAM and Windows XP as the operating system. Because of the used constraint solver, only programs comprising Boolean and Integer data types (including arrays of Integers) can be directly handled. The restriction to Boolean and Integer domains is only a limitation of the prototype implementation but not a limitation of Conbas.

For this empirical evaluation, we have computed all minimal hitting sets. We did not restrict the size of the hitting sets. However, since we only deal with single and double faults, hitting sets of the size 1 and 2 would be sufficient. This reduction would improve our results concerning the number of final diagnoses and the computation time.

For the first part of the empirical evaluation, we use the ten example programs listed in Table 5.1. Most of the programs implement numerical functions using conditional statements. The programs IfExample, SumPower, TrafficLight, and WhileLoops are borrowed from the JADE project[2]. The program Taste is borrowed from the Unravel project[3]. Table 5.1 depicts the obtained results and contains the following data:

[2] `http://www.dbai.tuwien.ac.at/proj/Jade/`
[3] `http://hissa.nist.gov/unravel/`

Table 5.1.: Conbas results for single faults

Program	V	LOC	Exec. trace	Con.	Int. var.	Total diag.	Valid diag.	Sum. Slice	Red. Slice	Time in ms
AKSWT	1	12	14	29	8	6	5	6	5	656
	2	12	17	39	11	3	3	6	3	203
	3	12	14	29	8	6	5	6	5	500
	4	12	4	3	2	2	1	2	1	141
	5	12	20	45	12	3	3	6	3	219
ProdSum	1	14	14	29	10	6	6	6	6	469
	2	14	14	31	11	7	6	8	7	469
	3	14	11	22	9	7	5	8	6	453
	4	14	14	31	11	3	3	8	3	203
	5	14	11	22	9	3	2	8	2	188
PowerFunc.	1	15	15	25	9	9	8	9	9	625
	2	15	9	9	5	6	5	6	6	359
	3	15	12	16	7	9	7	9	8	578
	4	15	15	23	9	4	4	9	5	250
	5	15	15	23	9	4	4	9	5	266
Multiplicat.	1	16	13	26	12	10	6	10	7	672
	2	16	14	25	9	8	8	8	8	500
	3	16	10	17	9	4	1	8	2	422
	4	16	11	19	7	6	3	8	3	375
	5	16	16	33	13	4	4	10	4	344
Divide	1	15	24	55	17	10	10	10	10	735
	2	15	26	62	22	12	12	12	12	906
	3	15	13	26	8	8	8	8	8	500
	4	15	10	18	6	4	3	8	3	250
	5	15	7	10	4	6	5	6	5	359
IfExamples	1	7	4	4	2	3	3	3	3	172
	2	8	3	3	2	2	1	2	1	110
	3	14	4	2	1	2	1	3	2	125
SumPowers	1	22	43	89	25	12	7	13	8	1141
	2	22	7	8	4	2	1	6	3	125
	3	22	7	8	4	2	1	6	3	109
	4	22	28	51	15	11	11	12	12	907
	5	22	28	58	17	12	12	13	13	890
TrafficLight	1	17	61	80	12	11	1	11	4	1078
	2	17	44	63	12	6	1	6	2	500
WhileLoops	1	14	52	99	14	7	7	7	7	719
	2	14	12	19	6	6	3	6	3	406
	3	14	52	99	14	7	7	7	7	719
Taste	1	15	10	21	11	4	4	7	5	266
	2	15	10	21	11	4	4	7	5	250
	3	15	28	69	23	6	6	9	7	469
	4	15	25	61	21	6	6	9	7	453
	5	15	19	45	17	6	6	9	7	438
Average		**15.1**	**18.1**	**34.1**	**10.4**	**6.0**	**4.9**	**7.7**	**5.5**	**454**

- the name of the program (*Program*),
- the fault version (*V*),
- the number of lines of code (*LOC*),
- the size of the execution trace (*Exec. trace*),
- the number of constraints (*Con.*),
- the number of internal variables in the CSP (*Int. var.*),
- the number of minimal diagnoses that are computed by the hitting set algorithm (*Total diag.*),
- the number of minimal diagnoses that are satisfiable by the constraint solver (*Valid diag.*),
- the number of statements contained in the union of the relevant slices of all faulty variables (*Sum. Slice*),
- the number of statements in the reduced slice (*Red. Slice*), and
- the computation time of CONBAS in milliseconds (*Time*).

Our results show that on average, the size of the slice is reduced by more than 28 % compared to the size of the corresponding summary slice. Figure 5.5 illustrates the relation of size of the program, the summary slice, and the reduced slice for the data presented in Table 5.1.

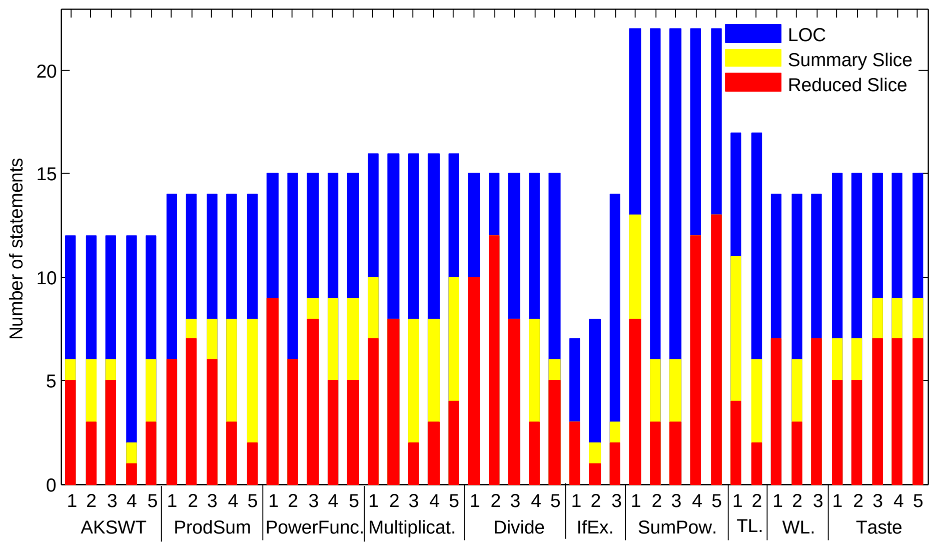


Figure 5.5.: Comparison of the number of statements in total (LOC), in the summary slice and in the reduced slice. The used program variants deal with single faults.

Figure 5.6 illustrates the relation of the number of minimal diagnoses (*Total diag.*) and the number of valid minimal diagnoses (*Valid diag.*) for the data presented in Table 5.1. CONBAS is able to eliminate about 20 % of the diagnoses.

In order to estimate the computation time for larger programs, we have

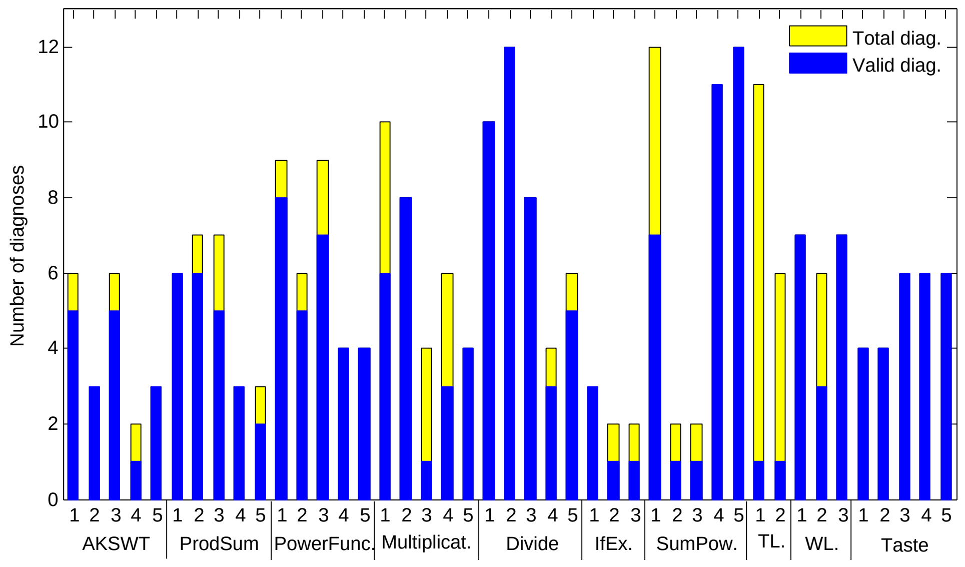


Figure 5.6.: Comparison of the number of minimal diagnoses and the number of valid minimal diagnoses. The used program variants deal with single faults.

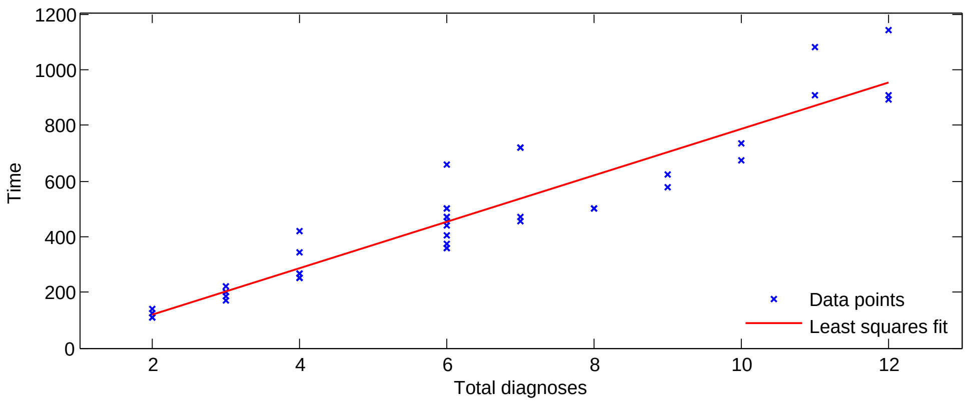


Figure 5.7.: The correlation of the number of diagnoses to be tested (*'Total diag.'*) and the computation time (*'Time'*, in milliseconds)

investigated if there exists a correlation between the time required for Conbas and (1) the *LOC*, (2) the size of the execution trace (*Exec. trace*), (3) the number of constraints (*Con.*), or (4) the number of diagnoses to be tested for satisfiability (*Total diag.*). We found out that the strongest correlation is between the execution time and the number of diagnoses to be tested for satisfiability. Figure 5.7 illustrates this correlation. The data points represent the data from Table 5.1. The red line represents the least squares fit as an approximation of the data.

One advantage of Conbas is that it is able to reduce slices of programs which contain two or more faults. In order to demonstrate this, we have performed a small evaluation on double faults. For this, we combined some faults used in the single fault evaluation. The faults were not combined according a particular schema (i.e. masking of faults or avoiding masking of faults). We only made the following restriction: Faulty program versions were not combined if the faults were in the same program line. The reason for this is, that two faults in the same line can be seen as one single fault. Table 5.2 shows the results obtained when executing Conbas on these new program versions. The table contains the following data:

- the name of the program (*Program*),
- the fault version (*V*),
- the number of lines of code (*LOC*),
- the number of statements contained in the union of the relevant slices of all faulty variables (*Summary Slice*),
- the number of statements in the reduced slice (*Reduced Slice*), and
- the number of faults contained in the reduced slice (*Faults in Red. Slice*).

Sometimes, it can be seen that only one of the two faults is contained in the reduced slice. The reason for this is that one fault can be masked by the other fault. Conbas guarantees that at least one of the faults is contained in the reduced slice. This is not a limitation since a programmer can fix the first bug and then apply Conbas again on the corrected program. Figure 5.8 shows the relation of the size of the program, the summary slice, and the reduced slice for the investigated double faults. On average, the summary slice can be reduced by 50 %.

In the second part of the empirical evaluation, we investigate if more faulty output variables allow for a higher reduction of the summary slice. For this purpose, we use the circuits C17 and C432 of the Iscas 85 [BF85] benchmark. The Iscas 85 circuits describe combinational networks. We have chosen this benchmark, because the different circuits of Iscas 85 have many input and output variables. The circuit C17 has 5 input variables and 2 output variables. The circuit C432 has 36 input variables and 7 output variables. For the evaluation, we have used test cases with different input and output

Table 5.2.: CONBAS results for double faults

Program	V	LOC	Summary Slice	Reduced Slice	Faults in Red. Slice
Taste	6d	15	9	6	2
	7d	15	9	6	2
	8d	15	7	4	2
	9d	15	9	5	2
	10d	15	7	4	2
	11d	15	7	4	2
	12d	15	9	6	2
	13d	15	9	6	2
	14d	15	9	6	2
	15d	15	9	2	2
SumPowers	6d	22	6	1	1
	7d	22	12	7	1
	8d	22	13	7	2
	9d	22	6	1	1
	10d	22	6	1	1
	11d	22	12	7	1
WhileLoops	4d	14	6	3	1
	5d	14	7	7	1
	6d	14	6	3	1
ProdSum	6d	14	8	3	1
	7d	14	8	3	1
	8d	14	8	3	1
	9d	14	8	2	1
	10d	14	8	2	2
	11d	14	8	2	1
Average		**16.3**	**8.2**	**4.0**	

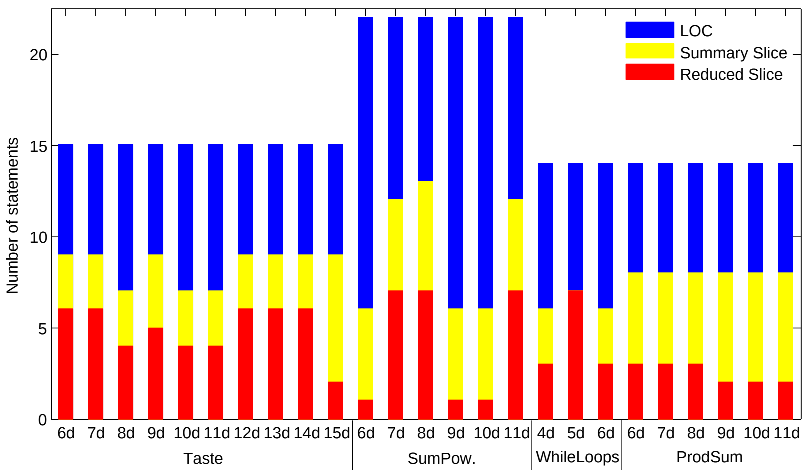


Figure 5.8.: Comparison of the number of statements in total (LOC), in the summary slice and in the reduced slice. The used program variants deal with double faults.

combinations. We used a maximum of three faulty output variables. In total, we created more than 150 program variants. Table 5.3 summarizes the obtained results for the two circuits of the ISCAS 85 benchmark. The column headings are similar to those used in Table 5.1. Therefore an explanation of the column headings can be found there. The results show that CONBAS is able to reduce the size of the summary slice by 66 %.

Table 5.3.: Summary of the CONBAS results for the ISCAS 85 benchmark

Circuit	LOC	Exec. trace	Con.	Int. var.	Total diag.	Valid diag.	Sum. Slice	Red. Slice
C17	31	14	26	25	12.0	5.9	9.3	5.1
C432	832	398	646	662	874.5	373.7	328.6	108.3
Average	**339.1**	**161.7**	**264.5**	**270**	**343.8**	**147.3**	**132.1**	**44.8**

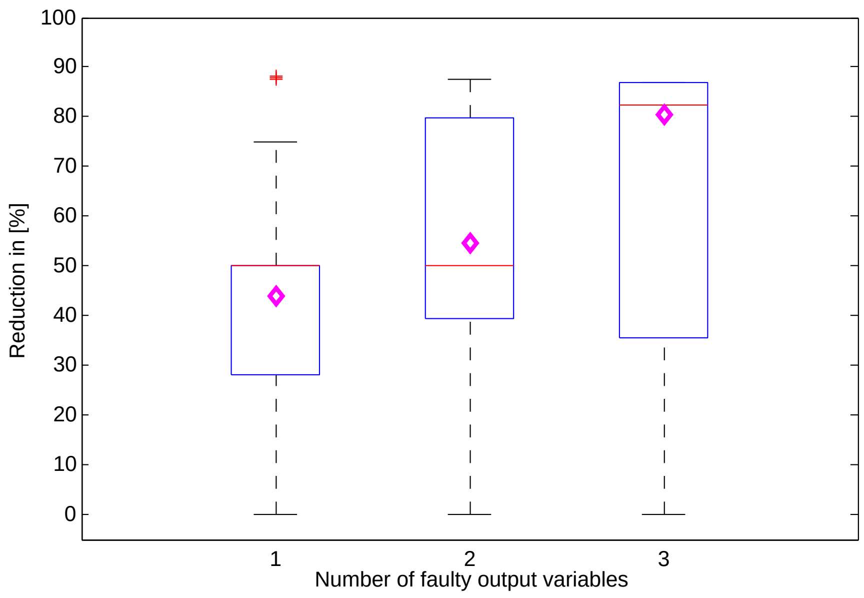


Figure 5.9.: Comparison of the REDUCTION quality for a different number of faulty output variables for the ISCAS 85 benchmark. The used program variants deal with single faults. For each box, the red line represents the median. The edges of the box are the 25th and 75th percentiles. The diamonds indicate the arithmetic mean of the reduction. The whiskers extend to the most extreme data points not considered outliers. Outliers are plotted individually as crosses.

Further, we want to analyze whether increasing the number of output variables leads to a smaller summary slice. In order to answer this question, we make use of the REDUCTION metric, which is defined as

$$\text{Reduction} = (1 - \frac{ReducedSlice}{SummarySlice}) \times 100\,\%. \tag{5.1}$$

We group the tested program variants by the number of faulty output variables and compute the Reduction metric for the program variants. Figure 5.9 shows the box plots for the different numbers of output variables. It can be seen that two and three faulty output variables yield a better reduction of the slice size than only one output variable. The reason for this is that there are fewer possible configurations which meet all of the specified output variables.

5.6. Conclusion

Dynamic program slices are a valuable aid for programmers because they provide an overview of possibly faulty statements when debugging. They are used in many automated debugging techniques as a preprocessing step. However, they are often still too large to be a valuable help. Therefore, we introduce an approach for reducing the size of slices, named Conbas (CONstraint Based Slicing). Conbas can be applied even if there exist multiple faults. In an empirical evaluation, we show that on average, the size of dynamic slices can be reduced by 28 % for single faults and by 50 % for double faults with the aid of constraint solving. In addition, we have applied Conbas on circuits of the Iscas 85 benchmark. These circuits contain many data dependencies but lack control dependencies. For these types of programs, Conbas yields a reduction of 66 % on average compared to the union of all slices.

The objective of Conbas is to improve relevant slicing for debugging. Even though, other approaches outperform Conbas in certain cases, we point out two application areas where Conbas should be the preferred method to use. (1) In case of software maintenance where the root cause for one failing test case has to be identified. In this case, mostly limited knowledge about the program is available. Moreover, the programs themselves are usually large, which makes debugging a hard task. In such a case, low-cost approaches that require a set of test cases might not be applicable and the application of heavy-weighted approaches might be infeasible because of computational requirements. (2) In case of programs with few control statements which need a more detailed analysis of data dependencies and relationships between variables. In such a case, Conbas provides the right means for analysis because of handling data dependencies and constraints between program variables.

Although, Conbas substantially reduces the number of diagnosis candidates, there is still room for improvements. The current implementation is not optimized both in terms of handling different kinds of program language constructs and time required for performing the analysis. Conbas uses a dynamic slicer instead of a relevant slicer. Therefore, root causes of failures might be ignored during the debugging process. Moreover, the integration of the constraint solver must be improved. In the current implementation, the calls to the external constraint solver slow down the computation.

Apart from these technical issues, there are some open research questions. Instead of computing the hitting sets of the slices, the constraint solver can be directly used to compute all solvable diagnoses of a particular size. Such an approach would restrict the number of constraint solver calls and also the time required for computing the hitting sets for the slices. Such an approach would be very similar to the approaches of Nica and colleagues [NNW12; WNM12], but it works on execution traces instead of the whole program representation. Another research challenge is to improve Conbas by using information about the evaluation of conditions. We have to analyze if taking the alternative execution path of a condition (e.g. the else path if the condition evaluates to true) could satisfy the test case. If the change leads to a consistent program behavior, a root cause is identified. Otherwise, the condition can be assumed to be correct and removed from the list of fault candidates.

The empirical evaluation of Conbas, especially in comparison with other approaches, has to be improved. The used programs are rather small. Larger programs that belong to different application domains have to be used for evaluation. An empirical study that compares different approaches such as Sfl with Conbas is necessary in order to structure the general research field of automated debugging.

Even though, Conbas cannot solve all debugging problems, we are convinced that Conbas is a valuable technique for improving the debugging process. Moreover, a combination with other debugging techniques may even increase its fault localization capabilities.

Part III.

Debugging of spreadsheets

6. Spreadsheet Engineering Techniques

Panko and Port [PP12] call End-User Computing "The Dark Matter of Corporate IT". They point out that immense risks come from the usage of spreadsheets, e.g. errors, privacy violations, and trade compliance violations. They encourage to see spreadsheets as the next level of programming languages. In particular, Panko and Port encourage people to treat spreadsheet development as an engineering discipline. Therefore, this chapter not only gives an overview of debugging techniques but also addresses work in the field of spreadsheet engineering, in particular debugging, testing, classification of error causes and spreadsheet engineering in general.

6.1. Debugging

Abraham and Erwig developed GoalDebug [AE07a; AE05; AE08], a spreadsheet debugger based on a constraint-based approach and on a mutation approach. GoalDebug not only focuses on fault localization but also on the generation of a ranked list of repair suggestions given a set of expected output values for individual cells. Whenever the computed output of a cell is incorrect, the user can supply an expected value for a cell. For the given expected value, GoalDebug delivers a list of possible repairs for the faulty spreadsheet. The set of possible changes is determined by pre-defined change inference rules. The subsequent ranking of possible "repairs" is based on the similarity of the changed formula to the original one. Other mentionable work of Abraham and Erwig includes the definition of mutation operators for spreadsheets [AE09]. For example, these mutation operators include absolute value insertion, arithmetic operator replacement and range shrinking mutation operators. The authors suggest to use these mutation operators for testing spreadsheets, for evaluating error-detection tools and for seeding faults into spreadsheets for empirical studies. Another approach developed by Abraham and Erwig is the UCheck system [AE07b]. UCheck is able to detect errors that are caused by unit faults. Therefore, they analyze the header information of spreadsheets and reason about the formulas.

Coblenz [CKM05] also reasoned about errors using the header information. Coblenz introduced the SLATE system, short for "A Spreadsheet Language for Accentuating Type Errors". This spreadsheet language separates the unit from the object of measurement. This technique helps to detect spreadsheet formula errors.

Jannach and Engler [JE10] present a model-based approach that calculates possible error causes in spreadsheets. Their approach uses an extended hitting-set algorithm and user-specified or historical test cases and assertions. Their approach computes the diagnoses by proving the satisfiability of the diagnoses. A disadvantage of their approach is that several test cases are necessary in order to obtain the conflict sets.

Ayalew and Mittermeir [AM03] address the spreadsheet debugging problem by presenting a trace-based fault localization approach. Their approach is data-flow driven and users the concept of slices in the spreadsheet domain. The authors propose an approach that prioritizes cells based on the number of incorrect successor cells and predecessor cells.

Ruthruff *et al.* [Rut+03] propose three techniques for visualizing possibly faulty cells. The first approach can be compared to spectrum-based fault localization with a very basic similarity coefficient. The second approach is similar to program dicing and the third technique is a nearest consumer technique.

6.2. Testing

Spreadsheet testing is closely related to debugging. The WYSIWYT ("What You See Is What You Test") approach [Rot+00] is the most prominent example for supporting the user in spreadsheet testing. This approach is a manual testing approach. Users can indicate incorrect output values by placing a faulty token in a cell. Similarly, they can indicate that the value in a cell is correct by placing a correct token. The "testedness" is determined by the test adequacy criterion DU (definition-use).

There exist approaches dealing with the automated generation of test cases for spreadsheets, e.g. [Fis+02] and AUTOTEST. AUTOTEST [AE06] is a tool that supports the spreadsheet developer through the automated generation of test cases in order to increase the coverage of the tests.

6.3. Classification of error causes

There have been several attempts to classify the causes of spreadsheet errors [RCK08; Pan98; PJ96; Gal+93; HS06; PA10]. Figure 6.1 illustrates the classification of Rajalingham *et al.* [RCK08] They distinguish between system-generated errors that are beyond the control of the user and user-generated errors. User-generated errors are divided into quantitative errors and qualitative errors. Quantitative errors are incorrect values. They arise accidentally or from faults in the reasoning process. Qualitative errors are errors concerning the maintenability and the semantic of the data, e.g. ambiguity in the meaning of data or formatting errors.

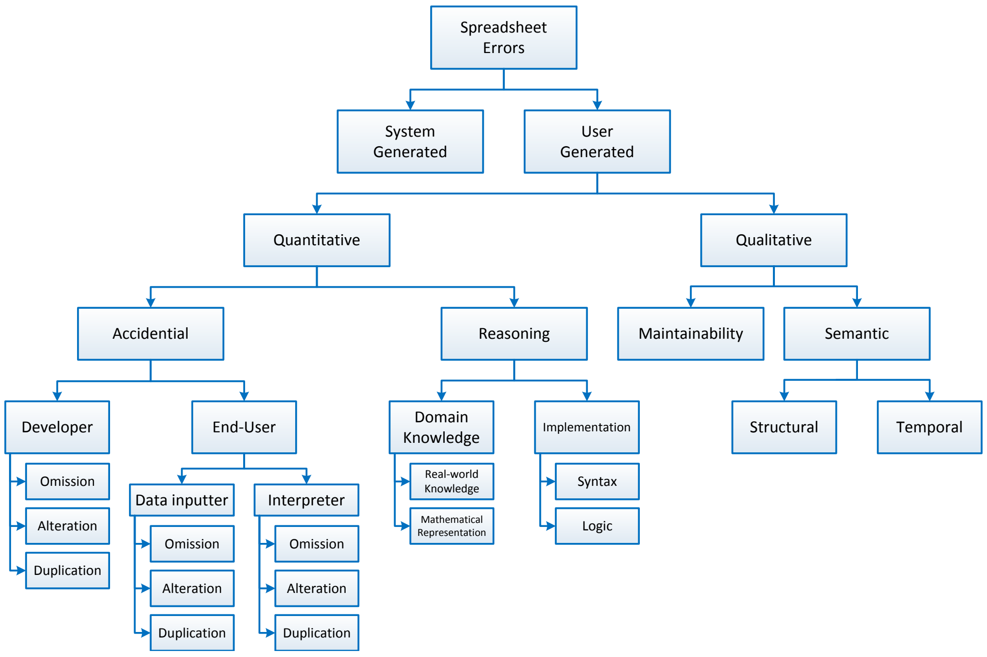


Figure 6.1.: Classification of spreadsheet errors according to Rajalingham *et al.* [RCK08]

6.4. Spreadsheet engineering

Since spreadsheet developers are typically end-users without significant background in computer science, there has been considerable effort to adapt software engineering principles to form a spreadsheet engineering discipline. Burnett *et al.* [Bur+03] suggest to use assertions in the spreadsheet domain. Cunha *et al.* [Cun+12] specialized on model-driven spreadsheet engineering. Mittermeir and Clermont [MC02] focus on identifying high-level structures in spreadsheets. Bregar [Bre08] developed metrics for determining the complexity of spreadsheet models.

Hermans *et al.* [HPD12b; HPD12a] address the issue of code smells in spreadsheets. In particular, they transform code smells defined for object-oriented programs (e.g. coupling and cohesion of classes) to the spreadsheet domain. Spreadsheet code smells are an important tool for improving spreadsheet quality with respect to usability, maintainability and error frequency. Other work of Hermans *et al.* includes the visualization of the dataflow in spreadsheets [HPD11], class diagram extraction [HPD10], data clone detection [Her+13], and metrics measuring the understandability of spreadsheet formulas [HPD12c].

7. Preliminaries

This chapter is based on the work published in [Hof+13].

7.1. Basic Definitions

A spreadsheet is a matrix comprising cells. Each cell is unique and can be addressed using its corresponding column and row number. For simplicity, we assume a function φ that maps the cell names from a set CELLS to their corresponding position (x, y) in the matrix where x represents the column and y the row number. The functions φ_x and φ_y return the column and row number of a cell, respectively.

Aside from a position, each cell $c \in$ CELLS has a value $\nu(c)$ and an expression $\ell(c)$. The value of a cell can be either undefined ϵ, an error $\perp$, or any number, Boolean or String value. The expression of a cell $\ell(c)$ can either be empty or an expression written in the language $\mathcal{L}$. The value of a cell c is determined by its expression. If no expression is explicitly declared for a cell, the function ℓ returns the value ϵ.

Areas are another important basic element of spreadsheets. An area is a set consisting of all cells that are within the area that is spanned by the cells $c_1, c_2 \in$ CELLS. Formally, we define an area as follows:

$$c_1{:}c_2 \equiv_{def} \left\{ c \in \text{CELLS} \;\middle|\; \begin{array}{l} \varphi_x(c_1) \leq \varphi_x(c) \leq \varphi_x(c_2) \;\;\& \\ \varphi_y(c_1) \leq \varphi_y(c) \leq \varphi_y(c_2) \end{array} \right\} \tag{7.1}$$

Obviously, every area is a subset of the set of cells ($c_1{:}c_2 \subseteq$ CELLS). After defining the basic elements of spreadsheets, we introduce the language $\mathcal{L}$ for representing expressions that are used to compute values for cells. For reasons of simplicity, we do not introduce all functions available in today's spreadsheet programs. Instead and without restricting generality, we make use of simple operators on cells and areas. However, extending the used operators with new ones is straightforward. The introduced language takes the values of cells and constants together with operators and conditionals to compute values for other cells. The language is a functional language, i.e., only one value is computed for a specific cell. Moreover, we do not allow recursive functions. First, we define the syntax of $\mathcal{L}$.

Definition 16 (Syntax of $\mathcal{L}$) *We define the syntax of $\mathcal{L}$ recursively as follows:*

- *Constants k representing ϵ, number, Boolean, or String values are elements of $\mathcal{L}$ (i.e., $k \in \mathcal{L}$).*
- *All cell names are elements of $\mathcal{L}$ (i.e., CELLS $\subset \mathcal{L}$).*
- *If e_1, e_2, e_3 are elements of the language ($e_1, e_2, e_3 \in \mathcal{L}$), then the following expressions are also elements of $\mathcal{L}$:*
  - *$\underline{(}e_1\underline{)}$ is an element of $\mathcal{L}$.*
  - *If o is an operator ($o \in \{\underline{+}, \underline{-}, \underline{*}, \underline{/}, \underline{\leq}, \underline{=}, \underline{\geq}\}$), then e_1 o e_2 is an element of $\mathcal{L}$.*
  - *$\underline{\textbf{if}(}e_1\underline{;}\ e_2\underline{;}\ e_3\underline{)}$ is an element of $\mathcal{L}$.*
- *If $c_1\underline{:}c_2$ is an area, then $\underline{\textbf{sum}(}c_1\underline{:}c_2\underline{)}$ is an element of $\mathcal{L}$.*

Second, we define the semantics of $\mathcal{L}$ by introducing an interpretation function $[\![\cdot]\!]$ that maps an expression $e \in \mathcal{L}$ to a value. The value is undefined (ϵ) if no value can be determined or error ($\perp$) if a type error occurs. Otherwise, it is either a number, a Boolean, or a String.

Definition 17 (Semantics of $\mathcal{L}$) *Let e be an expression from $\mathcal{L}$ and v a function mapping cell names to values. We define the semantic of $\mathcal{L}$ recursively as follows:*

- *If e is a constant k, then the constant is returned as result, i.e., $[\![e]\!] = k$.*
- *If e denotes a cell name c, then its value is returned, i.e., $[\![e]\!] = v(c)$.*
- *If e is of the form $\underline{(}e_1\underline{)}$, then $[\![e]\!] = [\![e_1]\!]$.*
- *If e is of the form e_1 o e_2, then its evaluation is defined as follows:*
  - *If $[\![e_1]\!] = \perp$ or $[\![e_2]\!] = \perp$, then $[\![e_1\ o\ e_2]\!] = \perp$.*
  - *else if $[\![e_1]\!] = \epsilon$ or $[\![e_2]\!] = \epsilon$, then $[\![e_1\ o\ e_2]\!] = \epsilon$.*
  - *else if $o \in \{\underline{+}, \underline{-}, \underline{*}, \underline{/}, \underline{\leq}, \underline{=}, \underline{\geq}\}$, then*

$$[\![e_1\ o\ e_2]\!] = \begin{cases} [\![e_1]\!]\ o\ [\![e_2]\!] & \textit{if all sub-expressions evaluate to a number} \\ \perp & \textit{otherwise} \end{cases}$$

- *If e is of the form $\underline{\textbf{if}(}e_1\underline{;}\ e_2\underline{;}\ e_3\underline{)}$, then*

$$[\![e]\!] = \begin{cases} [\![e_2]\!] & \textit{if } [\![e_1]\!] = \textbf{true} \\ [\![e_3]\!] & \textit{if } [\![e_1]\!] = \textbf{false} \\ \epsilon & \textit{if } [\![e_1]\!] = \epsilon \\ \perp & \textit{otherwise} \end{cases}$$

- *If e is of the form $\underline{\textbf{sum}(}c_1\underline{:}c_2\underline{)}$, then*

$$[\![e]\!] = \begin{cases} \sum\limits_{c \in c_1\underline{:}c_2} [\![c]\!] & \textit{if all cells in } c_1\underline{:}c_2 \textit{ have a number or } \epsilon \textit{ (treated as 0) as value} \\ \perp & \textit{otherwise} \end{cases}$$

Frequently, we require information about cells that are used as input in an expression. We call such cells *referenced cells*.

Definition 18 (Referenced cell) *A cell c is said to be referenced by an expression $e \in \mathcal{L}$, if and only if c is used in e.*

Furthermore, we introduce a function $\rho : \mathcal{L} \mapsto 2^{\text{CELLS}}$ that returns the set of referenced cells. Formally, we define ρ as follows:

Definition 19 (The function ρ) *Let $e \in \mathcal{L}$ be an expression. We define the referenced cells function ρ recursively as follows:*

- *If e is a constant, then $\rho(e) = \{\}$.*
- *If e is a cell c, then $\rho(e) = \{c\}$.*
- *If $e = \underline{(}e_1\underline{)}$, then $\rho(e) = \rho(e_1)$.*
- *If $e = e_1 \; o \; e_2$, then $\rho(e) = \rho(e_1) \cup \rho(e_2)$.*
- *If $e = \underline{\textbf{if}(}e_1\underline{;}\, e_2\underline{;}\, e_3\underline{)}$, then $\rho(e) = \rho(e_1) \cup \rho(e_2) \cup \rho(e_3)$.*
- *If $e = \underline{\textbf{sum}(}c_1\underline{:}c_2\underline{)}$, then $\rho(e) = c_1\underline{:}c_2$.*

A spreadsheet is a matrix of cells comprising values and expressions written in a language $\mathcal{L}$. The values of cells are determined by their expressions. Hence, $\forall c \in \text{CELLS} : \nu(c) = [\![\ell(c)]\!]$ must hold. Unfortunately, we face two challenges: (1) In all of the previous definitions, the set of cells need not be of finite size. (2) There might be a loop in the computation of values, e.g. a cell c with $\ell(c) = c\text{+}1$. In this case, we are not able to determine a value for cell c. In order to solve the first challenge, we formally restrict spreadsheets to comprise only a finite number of cells.

Definition 20 (Spreadsheet) *A countable set of cells $\Pi \subseteq CELLS$ is a spreadsheet if all cells in Π have a non empty corresponding expression or are referenced in an expression, i.e., $\forall c \in \Pi : (\ell(c) \neq \epsilon) \vee (\exists c' \in \Pi : c \in \rho(\ell(c')))$.*

In order to solve the second challenge, we have to limit spreadsheets to loop-free spreadsheets. For this purpose, we first introduce the notation of direct data dependence between cells, and furthermore the data dependence graph, which represents all dependencies occurring in a spreadsheet.

Definition 21 (Direct data dependence) *Let c_1, c_2 be cells of a spreadsheet Π. The cell c_2 depends directly on cell c_1 if and only if c_1 is used in c_2's corresponding expression, i.e., $dd(c_1, c_2) \leftrightarrow (c_1 \in \rho(\ell(c_2)))$.*

From the direct data dependence definition, we can derive the definition of data dependence graphs:

Definition 22 (Data dependence graph (DDG)) *Let Π be a spreadsheet. The data dependence graph (DDG) of Π is a tuple (V, A) with:*

- *V as a set of vertices comprising exactly one vertex n_c for each cell $c \in \Pi$, and*
- *A as a set comprising arcs (n_{c_1}, n_{c_2}) if and only if there is a direct dependence between the corresponding cells c_1 and c_2, i.e. $A = \bigcup (n_{c_1}, n_{c_2})$ where $n_{c_1}, n_{c_2} \in V \wedge dd(c_1, c_2)$.*

From this definition, we are able to define general dependence between cells. Two cells of a spreadsheet are dependent if and only if there exists a path between the corresponding vertices in the DDG. In addition, we are able to further restrict spreadsheets to face the second challenge.

Definition 23 (Feasible spreadsheet) *A spreadsheet Π is feasible if and only if its DDG is acyclic.*

From here on, we assume that all spreadsheets of interest are feasible. Hence, we use the terms spreadsheet and feasible spreadsheet synonymously. Standard spreadsheet programs like Excel rely on loop-free computations.

As the focus of this thesis is fault localization, we now focus on important definitions in the context of spreadsheet testing and debugging. In ordinary sequential programs, a test case comprises input values and expected output values. If we want to rely on similar definitions, we have to clarify the terms input, output and test case. Defining the input and output of feasible spreadsheets is straightforward by means of the DDG.

Definition 24 (Input, output) *Given a spreadsheet Π and its DDG (V, A), then the input cells of Π (or short: inputs) comprise all cells that have no incoming edges in the corresponding vertex of Π's DDG. The output cells of Π (or short: outputs) comprise all cells where the corresponding vertex of the DDG has no outgoing vertex.*

$$\begin{aligned} inputs(\Pi) &= \{c | \nexists (n_{c'}, n_c) \in A\} \\ outputs(\Pi) &= \{c | \nexists (n_c, n_{c'}) \in A\} \end{aligned} \tag{7.2}$$

All cells of a spreadsheet that serve neither as input nor as output are called intermediate cells. With this definition of input and output cells we are able to define a test case for a spreadsheet and its evaluation.

Definition 25 (Test case) *Given a spreadsheet Π, then a tuple (I, O) is a test case for Π if and only if:*

- *I is a set of tuples (c, e) specifying input cells and their values. For each $c \in inputs(\Pi)$ there must be a tuple (c, e) in I where $e \in \mathcal{L}$ is a constant.*
- *O is a set of tuples (c, e) specifying expected values for cells. The expected values must be constants of $\mathcal{L}$.*

Please note that the cells in O need not to be output cells. It is also possible to indicate values for intermediate cells. It is not necessary to define values for all output cells. The test case evaluation works as follows: First, the functions $\ell(c)$ of the input cells are set to the constant values specified in the test case. Subsequently, the spreadsheet is evaluated. Afterwards, the computed values are compared with the expected values stated in the test case. If at least one computed value is not equivalent to the expected value, the spreadsheet fails the test case. Otherwise, the spreadsheet passes the test case.

In traditional programming languages, test cases are separated from the source code. Usually, there are several test cases for one function under test. Each of the test cases calls the function with different parameters and checks the correctness of the returned values. However, test cases are only implicitly encoded into spreadsheets. This means, that test cases are not explicitly separated from the formulas under test. If the user wants to add an additional test case, he or she has to duplicate the spreadsheet. A duplication of a spreadsheet for testing purposes is unpractical since the duplicates have to be updated when the spreadsheet is modified or extended. Therefore, usually only one failing test case exists. Hence, we reduce the debugging problem for spreadsheets to handle only one test case.

Definition 26 (Spreadsheet debugging problem) *Given a spreadsheet Π and a failing test case (I, O), then the debugging problem is to find a root cause for the mismatch between the expected output values and the computed ones.*

We define the spreadsheet debugging problem as a fault localization problem. This definition implies that the following debugging approaches pinpoint certain cells of a spreadsheet as possible root causes of faults. Alternatively, the debugging problem can be defined as a fault correction problem.

8. Spreadsheet Corpora

The content of this chapter is based on the work published in [Hof+13] and work submitted for publication [Auß+13].

8.1. Introduction

Existing corpora like the EUSES spreadsheet corpus [FR05] have two main disadvantages: First, they do not come with faulty versions. Second, they contain many spreadsheets that are not suited for debugging purposes:

- small spreadsheets containing less than 5 formulas,
- spreadsheets without input values, and
- spreadsheets in obsolete file formats (Excel 5.0).

Therefore, we created two new spreadsheet corpora. The first corpus is a subset of the EUSES spreadsheet corpus. This new corpus does neither contain spreadsheets with less then 5 formulas, Excel 5.0 spreadsheets nor spreadsheets without input values. This modified EUSES spreadsheet corpus is discussed in Section 8.2. The second corpus is a collection of spreadsheets containing only Integer values and is discussed in Section 8.3. Both corpora are enhanced with faulty versions of the original spreadsheet.

8.2. The modified EUSES spreadsheet corpus

As already mentioned, the EUSES spreadsheet corpus contains many spreadsheets that are not suited for debugging. In a first filtering step, we skipped around 240 Excel 5.0 spreadsheets that are not compatible with our implementations, since our implementations is build on Apache POI[1], which does not support Excel 5.0. In a second filtering step, we removed all spreadsheets containing less than five formulas (about 2,300 files). We have performed this filtering step because automatic fault localization only makes sense for larger spreadsheets. A small spreadsheet is still manageable for humans and thus it is easy to manually locate the fault. For small spreadsheets, a fault correction

[1] http://poi.apache.org/

approach makes more sense than just a fault localization approach. In the third filtering step, we removed all spreadsheets that do not contain input values.

For each spreadsheet, we automatically created up to five first-order mutants. A mutant of a spreadsheet is created by randomly choosing a formula cell of the spreadsheet and applying a mutation operator on it. According to the classification of spreadsheet mutation operators of Abraham and Erwig [AE09], we used the following mutation operators:

- *Continuous Range Shrinking (CRS):* We randomly choose whether to increment the index of the first column/row or decrement the index of the last column/row in area references.
- *Reference Replacement (RFR):* We randomly choose whether to increment the row or the column index of references. We do not explicitly differentiate between single references and references in non-contiguous ranges. For this, a mutation can change a single reference in a non-contiguous range, but never changes the amount of elements in the range.
- *Arithmetic Operator Replacement (AOR):* We replace '+' with '-' and vice versa and '*' with '/'.
- *Relational Operator Replacement (ROR):* We replace the operators '=', '<', '<=', '>', '>=', and '<>' with one another.
- *Constants Replacement (CRP):*
  - For integer values, we add a random number between 0 and 1000.
  - For real values, we add a random number between 0.0 and 1.0.
  - For Boolean values, we replace 'true' with 'false' and vice versa.
- *Constants for Reference Replacement (CRR):* We replace a reference within a formula through a constant.
- *Formula Replacement with Constant (FRC):* We replace a whole formula with a constant.
- *Formula Function Replacement (FFR):* We replace 'SUM' with 'AVERAGE' and 'COUNT' and vice versa. We replace 'MIN' with 'MAX' and vice versa.

For each mutant, we check whether the following two conditions are satisfied: (1) The mutant must be valid, i.e., it does not contain any circular references. (2) The inserted fault must be revealed, i.e., at least for one output cell, the computed value of the mutant must differ from the value of the original spreadsheet. If one of these conditions is violated, we discard the mutant and generate new mutants until we obtain a mutant that satisfies both conditions.

We automatically created 622 mutants. The number of formulas contained in the spreadsheets ranges from 6 to more than 4,000. On average, the spreadsheets contain 225 formula cells. This indicates that the evaluated approaches are able to handle large spreadsheets. We made this modified version of the EUSES spreadsheet corpus publicly available[2]. This enables that other researchers can compare their approaches with ours.

8.3. Integer spreadsheet corpus

Since the constraint solvers used in our prototypes have only a limited ability to deal with Real numbers, we created a specific spreadsheet corpus that contains spreadsheets with Integer values only. This corpus contains 33 different spreadsheets. Whereas some of the spreadsheets are artificially created, 21 spreadsheets are real-life programs, e.g. a spreadsheet that calculates the lowest price combination on a shopping list or even the winner of Wimbleton 2012. The spreadsheets from the corpus contain both arithmetical and logical operators as well as the functions `SUM` and `IF`. The smallest spreadsheet contains seven formulas and the largest contains 233 formulas. On average, a spreadsheet contains 39 formula cells.

We created mutants for each spreadsheet by randomly selecting formulas and applying mutation operators on these formulas. The mutation creation process was the same as described in Section 8.2. In total, we created 220 mutants. This Integer spreadsheet corpus is also publicly available[3].

[2] `https://dl.dropbox.com/u/38372651/Spreadsheets/EUSES_Spreadsheets.zip`
[3] `https://dl.dropbox.com/u/38372651/Spreadsheets/Integer_Spreadsheets.zip`

9. Adaptation of debugging techniques

The content of this chapter is based on the work published in [Hof+13].

9.1. Introduction

In this chapter, we adapt two program-debugging approaches that have been designed for debugging programs written in 3rd generation languages. In particular, we describe how to modify these fault localization techniques in order to render them applicable to the spreadsheet world. We consider the following techniques in our study: Spectrum-based Fault Localization (Sfl) [AZG07], and Spectrum-enhanced dynamic slicing (Sendys) [HW12c]. We evaluate the efficiency of the approaches using mutants of real spreadsheets taken from the Euses Spreadsheet Corpus [FR05].

The remainder of this chapter is organized as follows: Section 9.2 discussed related work Section 9.3 explains the changes that have to be made in order to use the existing debugging techniques for the debugging of spreadsheets. Two traditional debugging techniques are explained in detail. In Section 9.4, we demonstrate these techniques by means of an example. Section 9.5 deals with the setup and the results of the empirical evaluation. Finally, Section 9.6 concludes this chapter.

9.2. Related Work

Ayalew and Mittermeir [AM03] and Ruthruff *et al.* [Rut+03] published work that is most related to the work presented in this chapter. Ayalew and Mittermeir [AM03] also make use of the concept of slices in their approach. They propose an algorithm for fault localization where the user has to determine for cells if they have fault symptoms. In contrast, the approaches presented in this chapter only require the 'health' status of the output variables.

Ruthruff *et al.* [Rut+03] propose three techniques for fault localization. Their approaches can be compared to spectrum-based fault localization and program dicing. However, they use very low-level similarity coefficients.

9.3. Necessary Adaptations

Traditional procedural and object-oriented program-debugging techniques cannot be directly applied in the spreadsheet domain for the following reasons: (1) In the spreadsheet paradigm, the concept of code coverage does not exist since there are no explicit lines of code like in traditional programming paradigms. (2) There is no concept of test execution. Therefore, in order to use traditional program-debugging techniques on spreadsheets, we have to perform some modifications: the lines of code in a traditional programming paradigm are mapped to the cells of a spreadsheet. There are cells designed to receive user input, cells to process data (using spreadsheet formulas), and cells intended to display the results. As an alternative to the code coverage of traditional programming paradigms, we compute so-called *cones* (data dependencies of each cell).

Definition 27 (The function CONE) *Given a spreadsheet Π and a cell $c \in \Pi$, we define the function* CONE *recursively as follows:*

$$\textsc{cone}(c) = c \cup \bigcup_{c' \in \rho(c)} \textsc{cone}(c') \tag{9.1}$$

The correctness of the output cells is determined either by the user, or by comparing the results of the current spreadsheet Π with another spreadsheet considered correct.

With these modifications, we are able to apply two traditional fault localization techniques on spreadsheets. In the following subsections, we explain these debugging techniques.

9.3.1. Spectrum-based Fault Localization

In traditional programming paradigms, Spectrum-based fault localization (SFL) [AZG07] uses code coverage data and the pass/fail result of each test execution of a given system under test (SUT) as input. This data is collected at runtime and is used to build a so-called hit-spectra matrix. In the spreadsheet paradigm, we cannot use the coverage data of test executions. Instead, we use the cones of the output cells (see Definition 27). From the cones, the hit-spectra matrix can be generated (each row of the matrix has the dependencies of one output cell). The error vector represents the correctness of the output cells. The hit-spectra matrix and the error vector allow the use of any similarity coefficient to compute the probability of being faulty for each spreadsheet cell. In the empirical evaluation, we use the Ochiai similarity coefficient, since Ochiai is known to be one of the most efficient similarity coefficients for Sfl [AZG06].

9.3.2. Spectrum-Enhanced Dynamic Slicing

In traditional programming paradigms, similar to Sfl, Sendys uses coverage data and the result of each test execution (pass/fail) of a given program as input. In addition, the slices of the negative test cases are required. In order to apply Sendys to the spreadsheet paradigm, we propose to make the same modifications as for the Sfl technique. In addition, we have to use cones instead of slices for the Mbsd part. The major difference between cones and slices are the types of dependencies. For slices, control and data dependencies are used. In contrast, cones only make use of data dependencies.

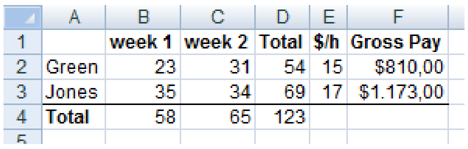
9.4. Example of usage

Figure 9.1 shows an example spreadsheet borrowed from the EUSES spreadsheet corpus [FR05]. For the sake of clarity, we have reduced the number of columns and rows of this example spreadsheet. Figure 9.1a illustrates the correct version of this spreadsheet and Figure 9.1b a faulty variant of the same spreadsheet. This spreadsheet is used to calculate the wages of the workers (cells `F2:F3`) and the total working hours (cell `D4`). Figure 9.1c shows the formula view of the faulty spreadsheet from Figure 9.1b. In this faulty spreadsheet, the computation of the total hours for the worker "Green" (cell `D2`) is faulty because the programmer of the spreadsheet unintentionally set a wrong area for the SUM formula. This happens for example when a programmer adds a new week but forgets to adapt some calculations. Because of this fault, the wage of the worker "Green" (cell `F2`) and the total hours (cell `D4`) are erroneous.

There exist two erroneous output cells (F2 and D4) and three correct output cells (B4, C4 and F3). For these cells, we compute the cones:

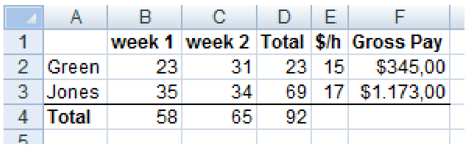
$$\textsc{Cone}(F2) = \{B2, D2, E2, F2\}$$

$$\textsc{Cone}(D4) = \{B2, D2, B3, C3, D3, D4\}$$



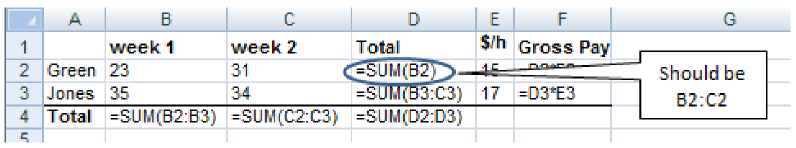
	A	B	C	D	E	F
1		week 1	week 2	Total	$/h	Gross Pay
2	Green	23	31	54	15	$810,00
3	Jones	35	34	69	17	$1.173,00
4	Total	58	65	123		
5						

(a) Correct spreadsheet



	A	B	C	D	E	F
1		week 1	week 2	Total	$/h	Gross Pay
2	Green	23	31	23	15	$345,00
3	Jones	35	34	69	17	$1.173,00
4	Total	58	65	92		
5						

(b) Faulty spreadsheet



	A	B	C	D	E	F	G
1		week 1	week 2	Total	$/h	Gross Pay	
2	Green	23	31	=SUM(B2)	15	=D2*E2	
3	Jones	35	34	=SUM(B3:C3)	17	=D3*E3	
4	Total	=SUM(B2:B3)	=SUM(C2:C3)	=SUM(D2:D3)			
5							

(c) Formula view of the spreadsheet from Figure 9.1b

Figure 9.1.: 'Workers' example borrowed from the EUSES spreadsheet corpus [FR05]

$$\textsc{Cone}(B4) = \{B2, B3, B4\}$$
$$\textsc{Cone}(C4) = \{C2, C3, C4\}$$
$$\textsc{Cone}(F3) = \{B3, C3, D3, E3, F3\}$$

The observation matrix that is build with that cone information is illustrated in Table 9.1. The faulty cell is ranked at position 1.

Table 9.1.: The spreadsheet 'Workers' - The observation matrix, the Ochiai coefficients and the subsequent ranking of the cells. The faulty cell is marked with •.

Line	F2	D4	B4	C4	F3	Coefficient	Ranking
B2	•	•	•			0.81	2
B3		•	•		•	0.40	7
B4			•			0.00	-
C2				•		0.00	-
C3		•		•	•	0.40	7
C4				•		0.00	-
•D2	•	•				1.00	1
D3		•			•	0.50	6
D4		•				0.70	3
E2	•					0.70	3
E3					•	0.00	-
F2	•					0.70	3
F3					•	0.00	-
Error	•	•					

For SENDYS, the hitting sets of the faulty cones are computed: There are two single fault diagnoses ($\{B2\}$ and $\{D2\}$) and eight double fault diagnoses ($\{B3, E2\}, \{B3, F2\}, \{C3, E2\}, \{C3, F2\}, \{D3, E2\}, \{D3, F2\}, \{D4, E2\}$, and $\{D4, F2\}$). The fault probabilities for the single faults are $p(\{B2\}) = 0.231$ and $p(\{D2\}) = 0.289$. Therefore, Sendys also ranks the faulty cell at position 1.

9.5. Empirical Evaluation

In this section, we compare the fault localization capabilities of Sfl and Sendys by applying them to the 622 spreadsheets of the modified EUSES spreadsheet corpus described in Section 8.3. In addition, we contrast these techniques with two primitive techniques, namely the union and intersection of the faulty cones. Table 9.2 summarizes the results of this comparison. The evaluation was performed on a processor powered by Intel Core2 Duo (2.67 GHz) and 4 GB RAM with Windows 7 Enterprise (64-bit) as the operating system and Java 7 as the runtime environment. Sendys performs slightly better than Sfl and the intersection of the cones. Since we only created first-order mutants, the intersection of the slices always contains the faulty cell. Please note that in case of higher-order mutants, the faulty cell could be absent in the intersection of the cones. This happens when two independent faults are contained in the same spreadsheet and both faults are revealed by different output cells. Therefore, the intersection of the cones is not the best choice. Concerning the computation time, Sfl has only a small overhead compared to the union and intersection of the cones. Sendys requires nearly five times more runtime for the computations.

Table 9.2.: Average ranking and computation time of Union, Intersection, Sfl, and Sendys. The 'Union' and the 'Intersection' sets are created from the cones of faulty output cells. The column 'Avg. relative ranking' shows the average ranking of the faulty cell normalized to the number of formula cells per spreadsheet.

Technique	**Avg. absolute ranking**	**Avg. relative ranking**	**Avg. comp. time (in ms)**
Union	41.1	27.3 %	15.6
Intersection	30.8	22.0 %	15.6
Sfl	26.3	20.3 %	16.9
Sendys	24.3	19.7 %	79.6

Figure 9.2 graphically compares the fault localization capabilities of the approaches for the 622 investigated faulty program versions. The x-axis represents the percentage of formula cells that are investigated. The y-axis represents the percentage of faults that are localized within that amount of cells. For Sfl and Sendys, the best and the worst case are indicated. The best case assumes that the faulty cell is found when investigating the first cell within a set of cells with the same ranking. The worst case assumes that the

faulty cell is found when investigating the last cell within a set of cells with the same ranking. Even in the worst case scenarios, Sfl and Sendys perform slightly better than the intersection and significantly better than the union of the cones. This means that faults can be detected earlier than when using the intersection or the union.

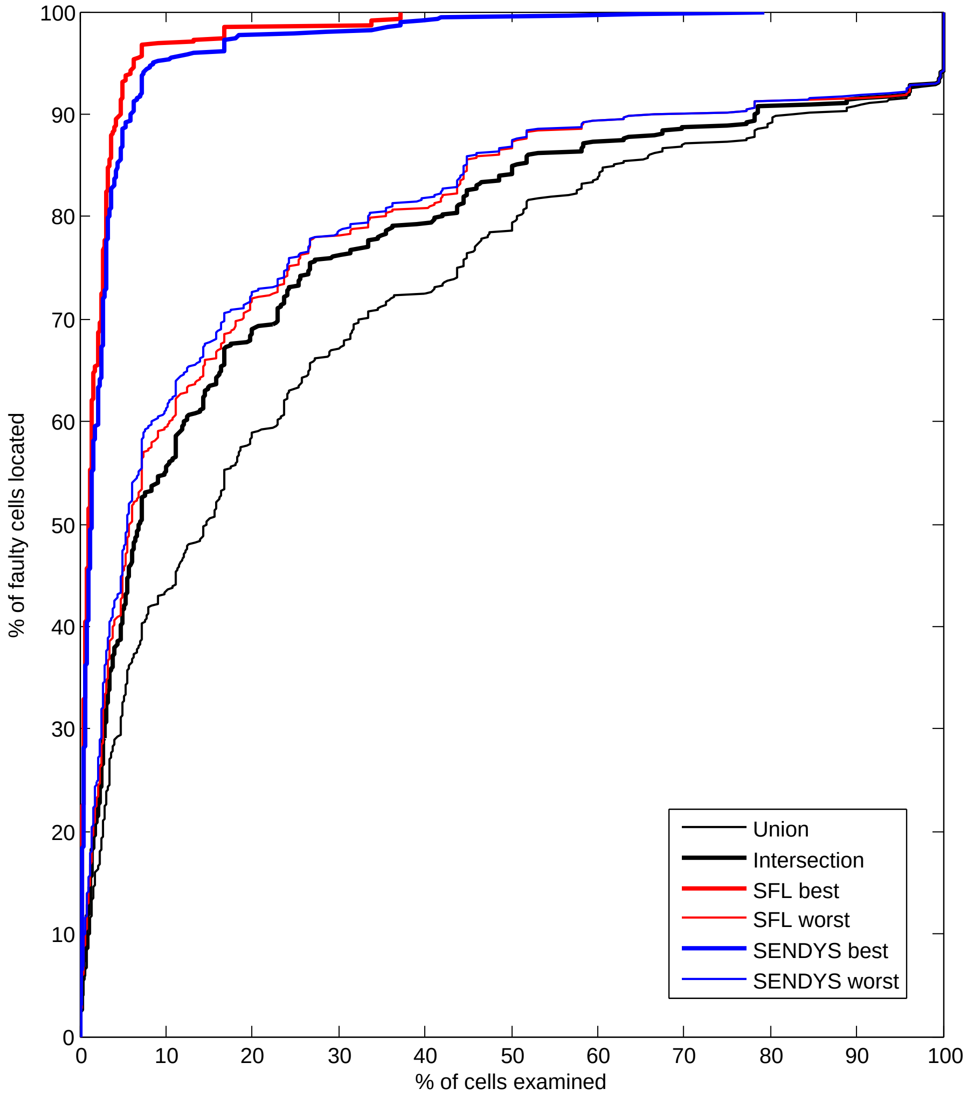
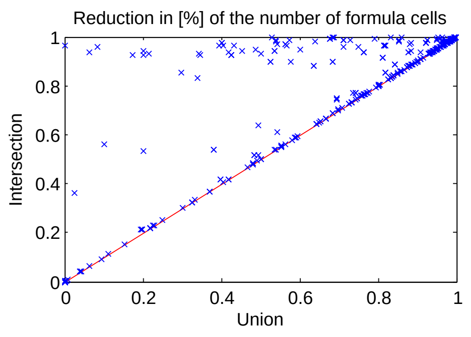
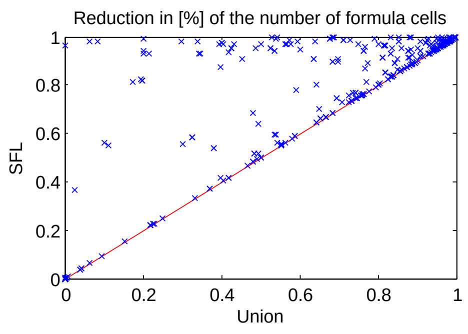


Figure 9.2.: Comparison of the Sfl, Sendys, the Union and Intersection of the cones in terms of the amount of formula cells that must be investigated.

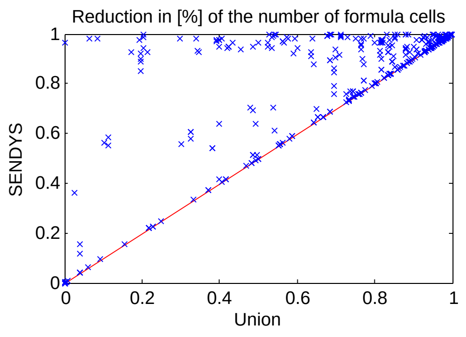
Figure 9.3 compares the approaches pairwise. For each spreadsheet and fault



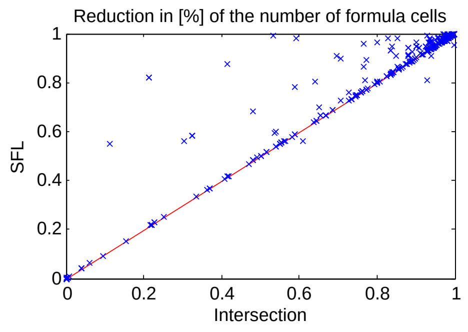
(a) Union versus Intersection



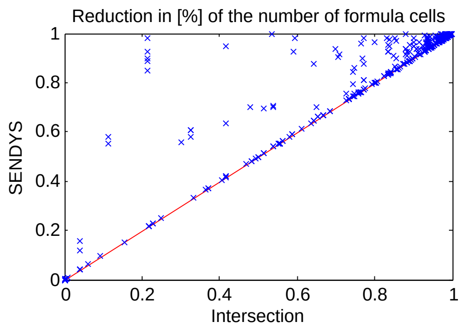
(b) Union versus SFL



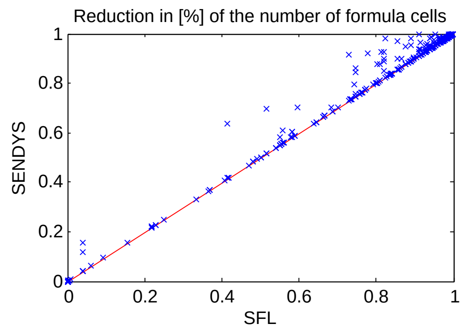
(c) Union versus SENDYS



(d) Intersection versus SFL



(e) Intersection versus SENDYS



(f) SFL versus SENDYS

Figure 9.3.: Pairwise comparison of the fault localization capabilities of Union, Intersection, SFL, and SENDYS in terms of the REDUCTION metric. Data points close to the red line indicate that the approaches perform equal. Data points above that line indicate that the approach labeled on the y-axis performs better.

localization technique, we computed the reduction metric as follows:

$$Reduction = (1 - \frac{invest}{formala}) \cdot 100\,\% \tag{9.2}$$

where *invest* indicates the number of cells that must be investigated until the first faulty cell is reached and *formula* indicates the total number of formulas contained in that spreadsheet. Please note that this analysis is a worst-case analysis: If several cells have the same ranking as the faulty cell, we assume that we have to examine all of them. Unsurprisingly, it can be seen in the Figures 9.3a, 9.3b and 9.3c that the union of the cones performs worst. Figure 9.3d shows that only in two cases the intersection performs better than SFL. In the other cases, SFL either performs as good as the intersection or better. Figures 9.3e and 9.3f show that SENDYS performs at least as good as the Intersection and SFL. In many cases, it performs better.

9.6. Conclusion

While spreadsheets are used by a considerable number of people, there is little support for automatic spreadsheet debugging. This chapter addresses this gap. In order to debug spreadsheets, we adapted and applied two popular debugging techniques designed for more traditional procedural or object-oriented programming languages. In addition, we explained what modifications to the traditional debugging techniques are necessary. The main modification is to use cones instead of execution traces and slices.

We evaluated the fault localization capabilities of the proposed techniques, SFL, and SENDYS, using a modified version of the well-known EUSES spreadsheet corpus. The evaluation shows that SFL and SENDYS are promising techniques. However, the evaluation needs to be extended in several aspects: (1) It is necessary to evaluate higher-order mutants. (2) The discussed techniques are only a small selection of the available traditional debugging techniques. Thus, other debugging techniques should be adapted to spreadsheets. Since our faulty versions of the spreadsheets are publicly available, we encourage new spreadsheet debugging techniques be compared to the techniques discussed in this chapter.

10. Constraint-based Debugging

Rui Abreu, André Riboira and Franz Wotawa developed an approach called ConBug [ARW12]. They invited me to participate in improving their approach. This chapter is based on unpublished work that was created with Rui Abreu, Alexandre Perez, André Riboira and Franz Wotawa and that is founded on [ARW12].

10.1. Introduction

In fault localization of programs written in 3rd generation languages, Mbsd with value-based models (e.g. [Wot+09]) is very successful. Therefore, it makes sense to apply this techniques for debugging spreadsheets. ConBug, short for Constraint-Based Debugging, is an approach that borrows ideas from Mbsd. It transforms a faulty spreadsheet and a given test case into a constraint satisfaction problem (Csp). The constraint solver is used to determine all diagnoses that could explain the observed misbehavior.

The basic idea of the conversion of the content of a spreadsheet into constraints is to use equations instead of assignments. For example, the cell F3 from our running example from Figure 9.1c (see page 104) contains the expression $\ell(F3) = D3 * E3$. Instead of using an assignment form $(F3 = D3 * E3)$, we use an equation: $F3 == D3 * E3$. The advantage of using equations instead of assignments is the direction of calculations: Assignments allow to deduce from the input to the output, but not vice versa. In contrast, equations allow to derive conclusions in both directions. Diagnosis candidates for the example spreadsheet from Figure 9.1 are cell D2 or the cells D4 and F2 together.

The remainder of this chapter is organized as follows: Section 10.2 deals with the related work. Section 10.3 explains the conversion of the spreadsheet debugging problem into a constraint satisfaction problem. In addition, we introduce an algorithm for computing diagnosis candidates. The design and the results of the empirical evaluation are discussed in Section 10.4. This empirical evaluation comprises both single faults and double faults. Section 10.5 concludes the approach and discusses future work.

10.2. Related Work

ConBug is based on model-based diagnosis. It is derived from model-based software debugging approaches like [Wot+09]. Jannach and Engler [JE10] also present a model-based approach. In contrast to ConBug, their approach uses an extended hitting-set algorithm and user-specified or historical test cases and assertions.

10.3. Spreadsheets as Constraint Satisfaction Problem

In order to solve the debugging problem stated in Definition 26, we have to convert spreadsheets into constraints. There exist some differences between the conversion of ordinary sequential programs and the conversion of spreadsheets: As discussed in Section 3.2.1, the conversion of programs requires three steps: (1) execution of the program or unrolling the loops, (2) transformation into a static single assignment form and (3) final compilation to constraints. In the domain of spreadsheets, there are no loops allowed and every cell can only be defined once. Hence, there is no need for loop removal and the static single assignment form.

Function ConvertSpreadsheet (Algorithm 10.1) converts a spreadsheet Π and a test case t into a set of constraints CON. For each cell, its formula is converted into constraints using the function ConvertExpression (Line 3). In Line 4, a constraint is created that models the debugging behavior: either the cell is abnormal or the cell must have the value computed in the expression. In the Lines 7 to 14, the test case information is added to the constraint system.

The recursive function ConvertExpression is shown in Algorithm 10.2. A constant or referenced cell is represented by itself (Lines 1 to 3). An expression e that consists of an expression e_1 enclosed by parentheses is represented by the constraints of e_1 (Lines 4 to 7). For an expression of the form $e_1\ o\ e_2$, we convert e_1 and e_2 separately into constraints (Lines 9 and 10). In addition, we create a new constraint that corresponds to the operator o (Line 12). The results of the constraint con is stored in the new created intermediate variable $result$. Therefore, an expression $e \in \mathcal{L}$ might be translated into several constraints. For the conversion of conditionals (Lines 15 to 21) and sums (Lines 22 to 25), particular constraints are used that are available in most of today's constraint languages: Let $\Psi(cond, e_1, e_2, result)$ be a constraint that ensures the relationship of $cond, e_1, e_2$ and $result$ as follows: If $cond$ is true, $result$ must be equal to the value of e_1. Otherwise $result$ must be equal to the value of e_2. Let $\mathrm{SUM}(c_1, ..., c_2, result)$ be a constraint that

Algorithm 10.1 CONVERTSPREADSHEET(Π, t)

```
Require: Spreadsheet $\Pi$, a failing test case $t$ with input $I$ and output $O$
Ensure: Set of constraints CON
 1: $\text{CON}_\Pi = \{\}$
 2: for cell $c \in \Pi$ do
 3:     $[\text{CON}, aux] = \text{CONVERTEXPRESSION}(\ell(c))$
 4:     $\text{con} = \text{AB}(c) \vee (c == aux)$
 5:     $\text{CON}_\Pi = \text{CON}_\Pi \cup \text{CON} \cup \{\text{con}\}$
 6: end for
 7: $\text{CON}_T = \{\}$
 8: for all tuples $(c, v) \in I$ do
 9:     $\text{CON}_T = \text{CON}_T \cup \{c == v\}$
10: end for
11: for all tuples $(c, v_{exp}) \in O$ do
12:     $\text{CON}_T = \text{CON}_T \cup \{c == v_{exp}\}$
13: end for
14: return $\text{CON}_\Pi \cup \text{CON}_T$
```

ensures that the sum of the values contained in the area c_1:c_2 is equal to the value of *result*.

Since spreadsheets must be finite, the algorithm CONVERTSPREADSHEET terminates. The computational complexity of the algorithm is $O(|\text{CELLS}| \cdot L)$ where L is the maximum length of an expression. Obviously, the conversion does not change the underlying behavior of a spreadsheet.

Algorithm CONBUG (Algorithm 10.3) illustrates the debugging process. In Line 2, the spreadsheet $\Pi \in \mathcal{L}$ and the failing test case t are converted into their constraint representation. In the Lines 4 to 12, the diagnosis candidates are computed, i.e., cells of the spreadsheet that might cause the revealed misbehavior. As we are interested in minimal diagnoses, we create a constraint that restricts the solution size: In Line 5, the function GETSIZECONSTRAINT(CON, n) creates a constraint that ensures at most n of the abnormal variables contained in CON can be set to true. In Line 6, the algorithm calls a constraint solver. The constraint solver returns all possible combinations of values of the abnormal variables $\text{AB}(c)$ so that the constraints CON and CON_{AB} are not violated. The size of a solution corresponds to the size of the bug, i.e., the number of cells that must be changed in order to correct the fault. We assume that single cell bugs are more likely than bugs comprising more cells. Hence, we ask the constraint solver for smaller solutions first. If no solution of a particular size is found, the algorithm increases the size of the solutions to be searched for. At the latest, the algorithm terminates when the solution size is equal to the number of formula cells in Π.

Algorithm 10.2 CONVERTEXPRESSION(e)

```
Require: Expression $e \in \mathcal{L}$
Ensure: [CON, $var$] with CON as a set of constraints, and $var$ as the name
    of an auxiliary variable, a cell name or a constant
 1: if $e$ is a cell name or constant then
 2:     return [$\{\}$, $e$]
 3: end if
 4: if $e$ is of the form $\underline{(}e_1\underline{)}$ then
 5:     Let [CON,$aux$] = CONVERTEXPRESSION($e_1$)
 6:     return [CON, $aux$]
 7: end if
 8: if $e$ is of the form $e_1$ $o$ $e_2$ then
 9:     Let [CON$_1$,$aux_1$] = CONVERTEXPRESSION($e_1$)
10:     Let [CON$_2$,$aux_2$] = CONVERTEXPRESSION($e_2$)
11:     Generate a new variable $result$
12:     Create a new constraint con accordingly to the given operator $o$, which
        defines the relationship between $aux_1$, $aux_2$, and $result$
13:     return [CON$_1$ $\cup$ CON$_2$ $\cup$ $\{$con$\}$, $result$]
14: end if
15: if $e$ is of the form $\underline{\text{if(}}e_1\underline{;}e_2\underline{;}e_3\underline{)}$ then
16:     Let [CON$_1$,$aux_1$] = CONVERTEXPRESSION($e_1$)
17:     Let [CON$_2$,$aux_2$] = CONVERTEXPRESSION($e_2$)
18:     Let [CON$_3$,$aux_3$] = CONVERTEXPRESSION($e_3$)
19:     Generate a new variable $result$
20:     return [CON$_1$ $\cup$ CON$_2$ $\cup$ CON$_3$ $\cup$ $\{\Psi(aux_1, aux_2, aux_3, result)\}$, $result$]
21: end if
22: if $e$ is of the form $\underline{\text{sum(}}c_1\underline{:}c_2\underline{)}$ then
23:     Generate a new variable $result$
24:     return [$\{$SUM$(c_1, ..., c_2, result)\}$, $result$]
25: end if
```

Algorithm 10.3 ConBug(Π, t)

```
Require: A spreadsheet $\Pi$ and a failing test case $t$
Ensure: Minimal diagnoses
 1: $\Delta^S = \{\}$
 2: CON = ConvertSpreadsheet$(\Pi, t)$
 3: solutionSize = 1
 4: while solutionSize $\leq |\Pi|$ do
 5:   $CON_{AB} = \{$GetSizeConstraint(CON, solutionSize)$\}$
 6:   $\Delta^S$ = Solve$(CON \cup CON_{AB})$
 7:   if $\Delta^S \neq \{\}$ then
 8:     return $\Delta^S$
 9:   else
10:     solutionSize = solutionSize + 1
11:   end if
12: end while
13: return $\Delta^S$
```

10.4. Empirical Evaluation

For performing the empirical evaluation, we developed a prototype . This prototype uses Minion [GJM06] as constraint solver. As Minion is only able to handle Integers, we used the Integer Spreadsheet Corpus described in Section 8.3. We set a timeout of 20 minutes for computing the solutions. 142 spreadsheets ended in a timeout. We are aware that the number of spreadsheets resulting in a timeout is large. Therefore, we address this problem in Chapter 11. For the remaining 78 spreadsheets, we measured the diagnosis quality by means of the achieved reduction and the time required for computing the diagnoses. The evaluation was performed on an Intel Core2 Duo processor (2.67 GHz) with 4 GB RAM and Windows 7 as operating system. We used the Minion version 0.15. The computation time is the average time over 100 runs. We only computed the diagnoses with lowest cardinality, i.e., we only computed double fault diagnoses when Minion did not report any single fault diagnoses.

The Tables 10.1 and 10.2 show the results. The column 'Formula cells' indicates the number of formula cells. The column 'Cells in diag.' indicates the number of cells that are contained in any diagnosis. The column 'Reduction' is computed as follows:

$$\text{Reduction} = \frac{\textit{Cells in diagnoses}}{\textit{Formula cells}} \times 100\,\%. \tag{10.1}$$

On average, an reduction of 57.5 % of the formula cells was possible when using ConBug. The column 'Constr.' indicates the number of constraints of

Table 10.1.: Results of the empirical evaluation - Part 1

Spreadsheet	Formula cells	Cells in diag.	Reduction (%)	Constr.	Comput. time (ms)
amortization_1_1	16	15	6.3	16	62
amortization_2_1	16	13	18.8	16	52
amortization_2_2	16	13	18.8	16	51
amortization_2_3	16	10	37.5	16	66
area_2_1	81	30	63.0	17	1047
area_2_2	81	28	65.4	22	947
area_2_3	81	59	27.2	22	1033
arithmetics00_1_1	8	7	12.5	22	105
arithmetics00_1_3	8	5	37.5	23	87
arithmetics00_2_2	8	8	0.0	23	174
arithmetics00_2_3	8	4	50.0	28	110
arithmetics00_3_1	8	8	0.0	28	169
arithmetics01_1_1	11	1	90.9	28	9
arithmetics01_1_2	11	4	63.6	34	10
arithmetics01_1_3	11	11	0.0	34	586
arithmetics02_2_2	16	5	68.8	34	343
arithmetics02_2_3	16	14	12.5	34	4001
arithmetics02_3_1	16	13	18.8	34	1936
austrian_league_1_1	32	1	96.9	34	196
austrian_league_1_2	32	1	96.9	34	280
austrian_league_2_1	32	7	78.1	34	734
austrian_league_2_2	32	1	96.9	34	215
austrian_league_2_3	32	1	96.9	34	162
austrian_league_3_1	32	24	25.0	36	1929
birthdays_1_1	39	9	76.9	42	194
birthdays_1_3	39	1	97.4	43	81
birthdays_3_1	39	4	89.7	52	163
cake_1_1	69	44	36.2	52	7350
cake_2_1	69	43	37.7	52	7315
cake_2_2	69	20	71.0	52	7011
cake_2_3	69	17	75.4	53	6913
cake_3_1	69	35	49.3	53	33123
computer_shopping_1_1	36	1	97.2	53	138
computer_shopping_1_2	36	1	97.2	68	138
computer_shopping_2_1	36	2	94.4	68	261
computer_shopping_2_2	36	1	97.2	68	140
computer_shopping_2_3	36	34	5.6	69	628
computer_shopping_3_1	36	1	97.2	69	142
conditionals01_1_1	11	2	81.8	69	24
conditionals01_1_2	11	7	36.4	69	32
conditionals01_2_1	11	4	63.6	87	29
conditionals01_2_2	11	5	54.5	87	33
conditionals01_2_3	11	7	36.4	87	34

Table 10.2.: Results of the empirical evaluation - Part 2

Spreadsheet	Formula cells	Cells in diag.	Reduction (%)	Constr.	Comput. time (ms)
conditionals02_1_1	7	3	57.1	101	50
conditionals02_1_3	7	3	57.1	101	79
conditionals02_2_1	7	3	57.1	101	76
conditionals02_2_2	7	3	57.1	101	77
conditionals02_2_3	7	4	42.9	102	71
conditionals02_3_1	7	3	57.1	102	68
dice_rolling_1_1	21	6	71.4	159	205
dice_rolling_2_1	21	7	66.7	159	231
dice_rolling_2_2	21	6	71.4	159	232
dice_rolling_2_3	21	6	71.4	159	205
dice_rolling_3_1	21	7	66.7	160	231
matrix_1_1	13	6	53.8	160	37
matrix_1_2	13	7	46.2	160	35
matrix_2_1	13	1	92.3	161	31
matrix_2_2	13	1	92.3	188	31
matrix_2_3	13	6	53.8	190	31
matrix_3_1	13	1	92.3	195	32
prom_calculator_1_1	14	13	7.1	195	23
prom_calculator_2_1	14	13	7.1	195	88
prom_calculator_2_2	14	13	7.1	195	18
prom_calculator_2_3	14	13	7.1	195	16
prom_calculator_3_1	14	10	28.6	196	21
shares_1_1	39	1	97.4	207	455
shares_1_2	39	1	97.4	207	388
shares_1_3	39	1	97.4	207	402
shares_1_4	39	1	97.4	207	328
shares_1_5	39	1	97.4	209	387
shares_2_2	39	2	94.9	210	1190
shares_2_3	39	18	53.8	210	2456
shopping_bedroom1_1_2	32	15	53.1	211	154
shopping_bedroom1_2_1	32	15	53.1	267	163
shopping_bedroom1_2_2	32	15	53.1	268	99
shopping_bedroom1_2_3	32	16	50.0	302	93
shopping_bedroom1_3_1	32	31	3.1	302	803
shopping_bedroom2_1_2	64	1	98.4	302	246
Average	**27.5**	**9.9**	**57.5**	**107.6**	**1,116.7**
Median	21.0	6.0	57.1	69.0	158.0
Stdev	19.9	11.4	31.9	82.6	4,016.3

the corresponding Csp. The total computation time is given in the last column and is 1.1 seconds on average. This low computation time make the approach practicable. Figure 10.1 illustrates the reduction quality w.r.t. the spreadsheets evaluated in the Tables 10.1 and 10.2. Only for 4 spreadsheets, the reduction was between 0 % and 5 %. For 17 spreadsheets, ConBug yields a reduction of more than 95 %.

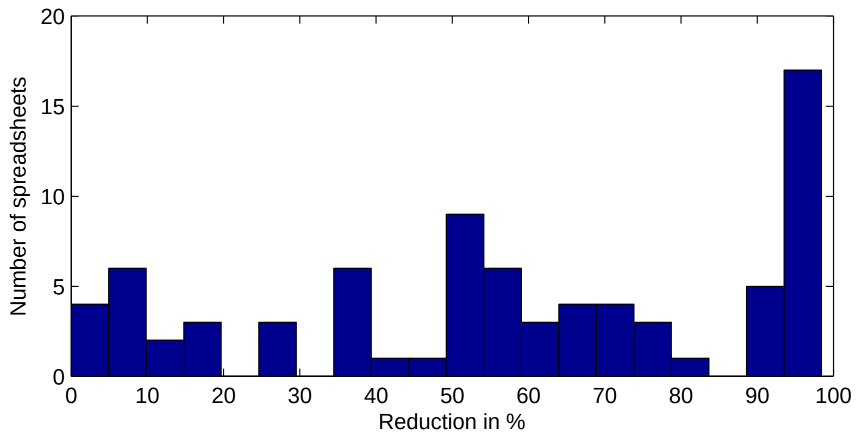


Figure 10.1.: Reduction histogram for the spreadsheets from the Tables 10.1 and 10.2

10.5. Conclusions and Future Work

ConBug is a constraint-based approach for fault localization in spreadsheets. The approach takes as input a spreadsheet and the set of user expectations (specifying the input and output cells and their expected values), and produces as output a set of diagnosis candidates. Diagnosis candidates are explanations for the misbehavior in user expectations. Our empirical investigation shows that ConBug is light-weight and efficient. On average, ConBug reduced the number of formula cells by 57.5 %. The average computation time is about 1.1 seconds. The current implementation comes with two major issues: (1) The use of Minion as constraint solver only allows variables of the types Integer and Boolean. (2) For the major part of the spreadsheets, ConBug results in a timeout. These issues are addressed in Section 11.

11. SMT versus Constraint Solving

This chapter is based on work published in [Auß+13].

11.1. Introduction

Currently, model-based debugging approaches for spreadsheets (e.g. ConBug and [JE10; ARW12]) use constraint solvers for dealing with the constraint satisfaction problem (Csp). While these approaches provide a profound background on the theoretic modeling of the spreadsheet debugging problem as a CSP, they are limited in their evaluations. The main reason for the limited evaluations seems to be the lack of constraint solvers being able to handle Real numbers. Therefore, we propose a novel approach that models the spreadsheet debugging problem as an SMT (satisfiability modulo theories) problem. The advantage of treating the spreadsheet debugging problem as a satisfiability problem is the availability of SMT solvers that are able to deal with Real numbers, e.g. Z3 [MB08]. In addition, SMT solvers are easily expandable with other theories and therefore the handling of additional data types is possible. Besides the enlargement of spreadsheets types that can be debugged, the usage of Z3 comes with a second advantage: a speedup in the computation time of diagnoses.

In particular, we compare the time performance of Z3 with those of the Choco[1] and Minion [GJM06] constraint solvers. We have chosen Z3 because it is one of the state-of-the-art SMT solvers, non-commercial and easy to integrate in Java. The range of constraint solvers is huge and there exist constraint solvers that are able to handle Real numbers, e.g. the Interval Constraint solver of Eclipse Prolog, and Ibm's Ilogs Cplex Cp Optimizer. We have chosen Minion and Choco since these constraint solvers have been used in comparable work, e.g. [JE10; ARW12]. However, Minion does not support Real numbers. In contrast, Choco officially supports Real numbers. However, we have learned that Choco does not support Real numbers to a complete extent. This thesis does not aim to make general assumptions about favoring SMT solvers over constraint solvers. Instead, it aims to show that SMT solvers

[1] http://www.emn.fr/z-info/choco-solver/

are a good alternative to constraint solvers for this particular application area, i.e., debugging of spreadsheets.

The remainder of this chapter is organized as follows: Section 11.2 deals with debugging using SMT solvers. In Section 11.3, the conversion into Minion and Choco constraints and Z3 formula clauses is demonstrated by means of an example. The empirical setup and the results of the empirical evaluation are discussed in Section 11.4. Finally, we conclude the approach in Section 11.5.

11.2. Debugging with Satisfiability Modulo Theories

Satisfiability Modulo Theories (SMT) is a technique that is based on Boolean satisfiability (SAT) enhanced by different theories, e.g. theory of Integers or Reals. An SMT problem is a decision problem: An SMT solver determines whether the given formulas are satisfiable according to the underlying theories. An SMT instance is a combination of first order logic formulas and background theories, where some predicates and functions have additional interpretations. For example, consider the clause $p \vee (x > y) \vee (y \leq 2z)$, where p is a Boolean variable and x, y and z are Integer variables. Predicates that are non-Booleans are evaluated according to the background theory. Such a theory $\mathcal{T}$ is defined over a signature Σ. Σ is a set of predicate and function symbols such as $\{0, 1, 2 \cdots \leq, \geq, \cdots +, -\}$. A formula φ is satisfiable w.r.t. $\mathcal{T}$ if there is an assignment that evaluates φ to true.

SMT solvers allow manifold strategies for determining the diagnoses. In this chapter, we use the MCSes [LS08] and the MCSes-U [LS09] algorithms to determine the diagnoses for a faulty spreadsheet. These algorithms aim to enumerate all minimal correction sets (MCSes) given an unsatisfiable constraint system. A correction set is a set of constraints which needs to be removed in order to make the constraint system satisfiable. A correction set cs is a minimal correction set (MCS) if there does not exist any proper subset of cs that is a correction set. In our case, finding all MCSes maps to finding all minimal diagnoses given a formula representing a faulty spreadsheet and a test case.

Algorithm 11.1 illustrates the MinimalCorrectionSets algorithm (MCSes) developed by Liffiton and Sakallah [LS08]. This algorithm requires a formula φ which is satisfiable if all variables contained in the set NAB are unassigned. NAB (short for not-abnormal) refers to a set of Boolean variables representing the decision whether the corresponding clause (or constraint) should be contained in the formula. Please note that NAB is the negated form of AB used in the Chapters 5 and 10. The algorithm computes all minimal

Algorithm 11.1 MinimalCorrectionSets(φ) [LS08]

```
Require: Formula $\varphi$ with unassigned not-abnormals (NAB)
Ensure: Minimal diagnoses MCSes
 1: MCSes = {}
 2: solutionSize = 1
 3: while Solve($\varphi$) == SAT do
 4:    $\varphi' = \varphi \wedge$ AtMost($\{nab \mid nab \in \text{NAB}\}$, solutionSize)
 5:    while solve($\varphi'$) == SAT do
 6:       MCS = GetNewMCS($\varphi'$)
 7:       MCSes = MCSes $\cup$ {MCS}
 8:       $\varphi' = \varphi' \wedge$ BlockingClause(MCS)
 9:       $\varphi = \varphi \wedge$ BlockingClause(MCS)
10:    end while
11:    solutionSize = solutionSize + 1
12: end while
13: return MCSes
```

diagnoses with increasing cardinality (solutionSize). The initial cardinality for the diagnoses is 1 (Line 2). In Line 3, an initial check is performed whether the solver is able to find any diagnosis. If the solver returns UNSAT, i.e., there does not exist any solution, the algorithm terminates immediately. Otherwise, the solutionSize constraint is added to a temporary copy φ' of the formula (Line 4). We use the function AtMost defined Liffiton and Sakallah [LS08] for the solution size:

$$\text{AtMost}(\{l_1, l_2, \ldots, l_n\}, k) \equiv \sum_{i=1}^{n} value(l_i) \leq k, \tag{11.1}$$

where $l_1, l_2, \ldots, l_n$ are Booleans, k is an Integer and $value(l_i)$ is 1 if l_i is assigned false, 0 otherwise. Please note that the function AtMost is analogous to the function GetSizeConstraint used in Algorithm 10.3.

The inner loop (Line 5) is necessary since an SMT solver does not allow to retrieve all models in contrast to constraint solvers. Therefore, an SMT solver has to be called several times to obtain all MCSes with the maximum cardinality solutionSize. In each iteration of the inner loop, the function GetNewMCS (Line 6) is called with φ'. This function calls the solver to retrieve a model and obtains an MCS as follows:

$$\text{MCS} = \{nab \in \text{NAB} : model(nab) = \text{true}\}\,. \tag{11.2}$$

In the Lines 8 and 9, a blocking clause for the newly found MCS is added to the formula φ' and to the original formula φ. The blocking clauses are necessary to avoid a repeated reporting of already found diagnoses or their supersets. A blocking clause for a diagnosis MCS is a disjunction of all not-abnormals (NAB) which are contained in the diagnosis:

$$\textsc{BlockingClause}(\mathrm{MCS}) = \bigvee_{nab \in \mathrm{MCS}} nab. \tag{11.3}$$

After all diagnoses with the desired maximum cardinality are found (i.e., the $\textsc{Solve}(\varphi') ==$ UNSAT), the solutionSize is incremented. In the outer loop, the solver checks whether the original formula φ extended by the blocking clauses is still satisfiable (i.e., solutions are missing). If it is satisfiable, the missing MCSes are computed. Otherwise, the algorithm returns the set MCSes.

Liffiton and Sakallah published an improved version of the MCSes algorithm, which increases the performance. This algorithm, namely the MCSes-U algorithm [LS09], uses the unsatisfiable core. The unsatisfiable core is an SMT solver's byproduct of the proof of unsatisfiability and is a set of variables which lead to the unsatisfiability of a given formula, i.e., a conflict set. Although not necessarily minimal, the unsatisfiable core can be used to limit the number of clauses which need to be considered during the calculation of minimal correction sets. At this, true can be assigned to all *nab* which are not included in the core set. This decreases the number of unassigned variables and reduces the solving time.

Algorithm 11.2 illustrates the pseudo code of the algorithm provided by Liffiton and Sakallah. In Line 3, all not-abnormals (NAB) are assumed to be true, which leads to the unsatisfiability of φ since the formula represents a faulty spreadsheet. The function GetCore retrieves the unsatisfiability core, a not necessarily minimal set of variables which lead to the unsatisfiability of φ. In Line 4, the solver is called without any assumption for the not-abnormal variables (NAB). The solver returns SAT if there exists at least one diagnosis which has not been reported yet. If the solver cannot satisfy the formula, the algorithm terminates, otherwise a new temporary formula is created by the function Instrument (Line 5). This function adds the AtMost cardinality constraint to a copy of φ. Moreover, each not-abnormal which is not included in the core, is set to true since it is not involved in the current conflict. Variables which are included in the core remain unassigned. The loop in Line 7 is responsible for reporting all MCSes, which can resolve the conflict corresponding to the current unsatisfiable core. This is done in the same manner as in the previously described MCSes algorithm. After finding all diagnoses with the given cardinality, a new core set is retrieved by solving φ' under the assumption that not-abnormals which are not contained in the core are true (Line 13). The resulting new core is added to *core*. Line 6 checks

Algorithm 11.2 MCSes-UnsatisfiableCores(φ) [LS09]

```
Require: Formula $\varphi$ with unassigned not-abnormals (NAB)
Ensure: Minimal diagnoses MCSes
 1: MCSes = {}
 2: solutionSize = 1
 3: $core$ = GetCore($\varphi$, NAB)
 4: while Solve($\varphi$) == SAT do
 5:   $\varphi'$ = instrument($\varphi$, $core$, solutionSize)
 6:   while Solve($\varphi'$) == SAT do
 7:     while Solve($\varphi'$) == SAT do
 8:       MCS = GetNewMCS($\varphi'$)
 9:       MCSes = MCSes $\cup$ {MCS}
10:       $\varphi' = \varphi' \wedge$ BlockingClause(MCS)
11:       $\varphi = \varphi \wedge$ BlockingClause(MCS)
12:     end while
13:     $core = core \cup$ GetCore($\varphi'$, NAB$\setminus core$)
14:     $\varphi'$ = Instrument($\varphi$, $core$, solutionSize)
15:   end while
16:   solutionSize = solutionSize + 1
17: end while
18: return MCSes
```

if the newly added not-abnormals in the core lead to further solutions with the same maximum solutionSize. In this case, the inner loop in Line 7 is executed at least one time, otherwise solutionSize is incremented.

In order to use SMT solvers instead of constraint solvers, we have to modify the conversion of the spreadsheet. Algorithm 11.3 shows the function ConvertSpreadsheetIntoSmtClauses. This function differs from the function ConvertSpreadsheet (Algorithm 10.1) in two aspects: (1) Instead of creating a set of constraints, the function ConvertSpreadsheetIntoSmtClauses creates a formula comprising all constraints combined by conjunctions. (2) In place of the abnormal statement $\mathrm{AB}(c)$, the negated version of $\mathrm{AB}(c)$, i.e., $nab(c)$, is used.

Example 11.1 *Let the formula ℓ of the cell B4 be "B1+B2". The constraint representation of this formula is $AB(B4) \vee B4 == (B1 + B2)$. The formula clause representation of this cell is $nab(B4) \implies B4 == (B1 + B2)$.* □

The function ConvertExpressionSMT (Line 3) is similar to the function ConvertExpression (Algorithm 10.2) but returns a formula instead of a set of constraints. In order to compute the diagnoses for a faulty spreadsheet, we

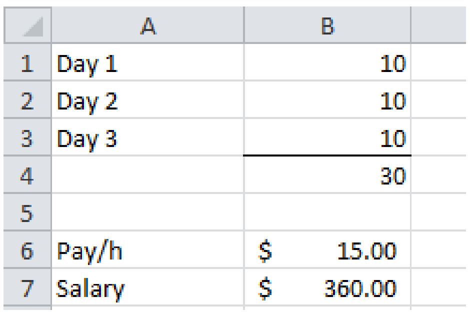
Algorithm 11.3 CONVERTSPREADSHEETINTOSMTCLAUSES(Π, t)

```
Require: Spreadsheet $\Pi$, a failing test case $t$ with input $I$ and output $O$
Ensure: Formula clause $\varphi$
1:  Let $\text{CON}_\Pi$ be empty
2:  for cell $c \in \Pi$ do
3:      $[\text{CON}, aux] = $ CONVERTEXPRESSIONSMT$(\ell(c))$
4:      con $= nab(c) \implies (c == aux)$
5:      $\text{CON}_\Pi = \text{CON}_\Pi \wedge \text{CON} \wedge con$
6:  end for
7:  Let $\text{CON}_T$ be empty
8:  for all tuples $(c, v) \in I$ do
9:      $\text{CON}_T = \text{CON}_T \wedge (c == v)$
10: end for
11: for all tuples $(c, v_{exp}) \in O$ do
12:     $\text{CON}_T = \text{CON}_T \wedge (c == v_{exp})$
13: end for
14: return $\text{CON}_\Pi \wedge \text{CON}_T$
```

apply the function MCSES-UNSATISFIABLECORES (Algorithm 11.2) on the formula obtained from the function CONVERTSPREADSHEETINTOSMTCLAUSES.

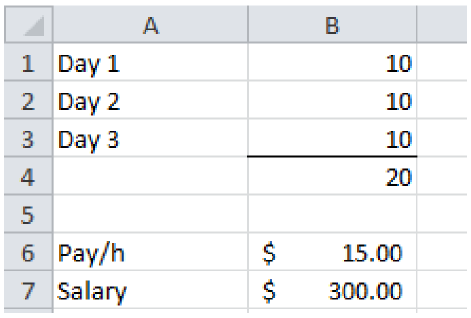
11.3. Example of application

In order to illustrate the basic concepts of our technique, we make use of the running example shown in Figure 11.1. The spreadsheet is used to collect the working hours for three days and to calculate the salary. The total working hours are contained in cell B4. The user can enter the wage per hour in cell B6. The overall salary (cell B7) is calculated by multiplying the total amount of working hours with the wage per hour. If an employee's total amount of working hours exceeds 24 hours, she is only paid 24 hours. Figure 11.1b illustrates the faulty version of this spreadsheet. In this faulty spreadsheet, the computation of the total hours only contains the amount of hours for the first two days. As a result, the salary (the value of cell B7) is erroneous. Figure 11.1c shows the formula view of this faulty spreadsheet. MBSD helps to identify the possible root causes given the observation that cell B7 contains a wrong value. Diagnoses for this faulty spreadsheet (i.e. explanations of the observed misbehavior) are the cells B4 and B7. Changing one of these cells allows to fix the fault. In the following subsections, we show the conversion for the constraint solvers Choco and MINION and for the SMT solver Z3.



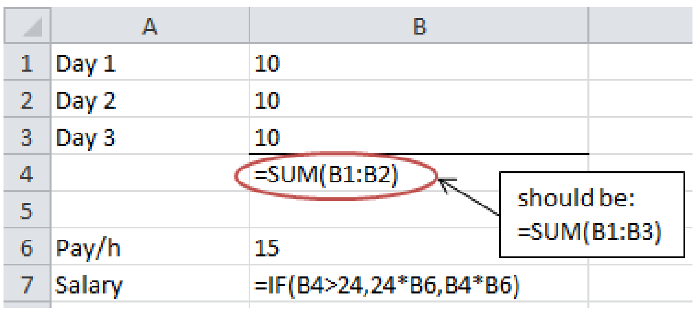
	A	B
1	Day 1	10
2	Day 2	10
3	Day 3	10
4		30
5		
6	Pay/h	$ 15.00
7	Salary	$ 360.00

(a) Correct spreadsheet



	A	B
1	Day 1	10
2	Day 2	10
3	Day 3	10
4		20
5		
6	Pay/h	$ 15.00
7	Salary	$ 300.00

(b) Faulty spreadsheet



	A	B
1	Day 1	10
2	Day 2	10
3	Day 3	10
4		=SUM(B1:B2)
5		
6	Pay/h	15
7	Salary	=IF(B4>24,24*B6,B4*B6)

should be:
=SUM(B1:B3)

(c) Formula view of the faulty spreadsheet

Figure 11.1.: Spreadsheet example 'Salary'

MINION

Listing 11.1 shows the MINION representation of the example from Figure 11.1. In the first part of the conversion, all required variables are defined (Lines 3 to 7). `tmp6` represents an auxiliary variable that was introduced during the conversion. The search strategy is determined in Line 9. Since we are interested in which variables might behave abnormal, we focus on the values of the abnormal array (`ab`). The Lines 10 to 17 represent the constraints of the cells' formulas. The MINION constraints `sumleq` and `sumgeq` are used to represent the plus operator, and `weightedsumleq` and `weightedsumgeq` together with the given list of signs are for representing the minus operator. Finally, the Lines 19 to 20 encode the test case and the Lines 22 and 23 determine the size of the diagnoses. In this case, the MINION solver computes single fault diagnoses. MINION restricts the domain of variables to Boolean and Integer. Therefore, it is not possible to use Real numbers in MINION. As a consequence, MINION can only be used for the debugging of spreadsheets containing Booleans and Integers.

```
MINION 3
**VARIABLES**
DISCRETE B1{-2000..5000}
BOOL ab[2]
BOOL tmp6
DISCRETE tmp7{-2000..5000}
...
**SEARCH**
VARORDER [ab]
watched-or({element(ab,0,1),weightedsumgeq([1,1],[B2,B1],B4)})
watched-or({element(ab,0,1),weightedsumleq([1,1],[B2,B1],B4)})
watched-or({element(ab,1,1),reifyimply(eq(B7,tmp8),tmp6)})
watched-or({element(ab,1,1),product(tmp7,B6,tmp8)})
watched-or({element(ab,1,1),diseq(tmp6,tmp11)})
watched-or({element(ab,1,1),reifyimply(eq(B7,tmp9),tmp11)})
watched-or({element(ab,1,1),product(B4,B6,tmp9)})
watched-or({element(ab,1,1),reifyimply(ineq(tmp5,B4,-1),tmp6)})
#TEST CASE
eq(B1,10)
...
#SIZE OF SOLUTION
watchsumgeq(ab,1)
watchsumleq(ab,1)
**EOF**
```

Listing 11.1: MINION representation of the running example

Choco

Listing 11.2 shows fragments of the conversion result for the Choco constraint solver. The first part (Lines 1 to 7) defines the variables and their bounds. Boolean variables are defined by an Integer with a range from 0 to 1. If the value of a cell is known (e.g. for constants and test case values), the range of the cell variable is set to this value. Lines 8 to 20 contain the transformed spreadsheet. Lines 21 to 23 encode the size of the diagnoses. In this case, we are looking for single fault diagnoses. In contrast to MINION, Choco allows constraints with variables of the type Real in principle. However, a reification of constraints containing Real numbers is not allowed in Choco. Therefore, it is not possible to model a cell c as $\mathrm{AB}(c) \vee con(c)$ if the cell references or computes Real numbers.

```
 1 varB1 [10,10]
 2 varB3 [-2000,5000]
 3 abB3 [0,1]
 4 aux0 [-2000,5000]
 5 ...
 6 upperBoundActive [0, 1]
 7 upperBound [0, 6]
 8 eq({aux0,INTEGER_EXPRESSION{varB1,varB2}})
 9 implies({eq({abB4,0}),eq({varB4,aux0})})
10 eq({varB1,10})
11 eq({varB7,300})
12 reifiedconstraint({aux1,varB4,24})
13 eq({aux2,INTEGER_EXPRESSION{24,varB6}})
```

```
eq({aux3,INTEGER_EXPRESSION{varB4,varB6}})
or({and({neq({aux1,0}),eq({aux4,aux2})}),and({not({neq({aux1,0})}),eq
    ({aux4,aux3})})})
implies({eq({abB7,0}),eq({varB7,aux4})})
eq({varB2,10})
implies({eq({abB3,0}),eq({varB3,10})})
eq({varB6,15})
eq({varB7,300})
implies({eq({upperBoundActive,1}),eq({INTEGER_EXPRESSION{abB4,abB7,
    abB3},upperBound})})
eq({upperBoundActive,1})
eq({upperBound,1})
```

Listing 11.2: Choco representation of the running example

Z3

In contrast to Choco and Minion, Z3 allows to use Real numbers in formula clauses without any restrictions. The running example encoded in Z3 is illustrated in Listing 11.3. In contrast to Minion and Choco, the 'health' state of the cells is represented by not-abnormals instead of abnormals.

```
(declare-fun varB1 () Int)
(declare-fun varB2 () Int)
(declare-fun varB3 () Int)
(declare-const nabB3 Bool)
...
(assert(= varB1 10))
(assert(= varB2 10))
(assert(=> nabB3 (= varB3 10)))
...
(assert(=> nabB7 (= varB7 (ite (> varB4 24) (* 24 varB6) (* varB4
    varB6)))))
```

Listing 11.3: SMT representation of the running example

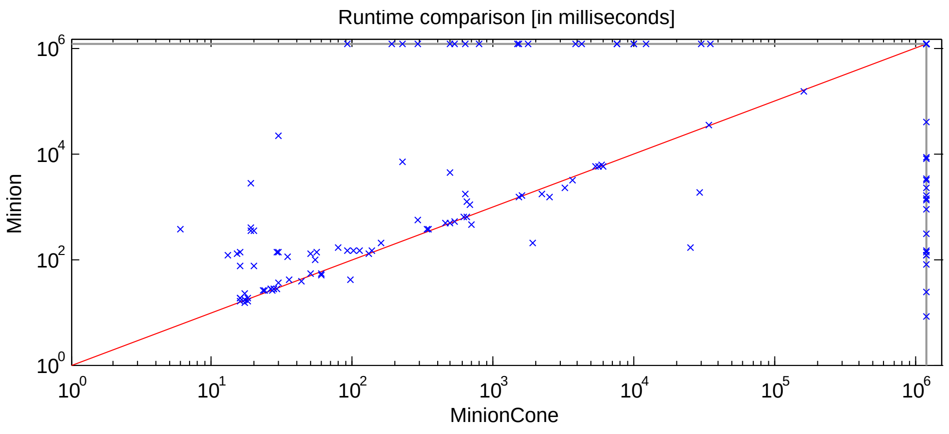
11.4. Empirical Evaluation

The empirical evaluation consists of three major parts: (1) the time comparison of Z3, Minion and Choco, (2) the time comparison of Z3 for the Integer and Real domains, and (3) the evaluation of the fault localization quality.

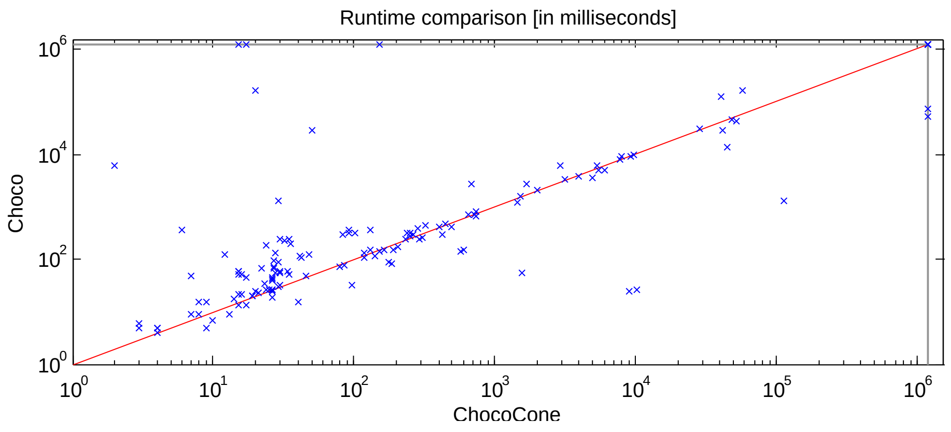
We implemented a prototype in Java that is able to transform a spreadsheet into Minion and Choco constraints or a Z3 formula respectively. This prototype supports the transformation of arithmetic and relational operations as well as of the functions `SUM` and `IF`. The transformation of other functions like `AVG` and `MAX` is technically feasible, but their transformation is not implemented yet. In order to be fair, this prototype uses similar search strategies for the solvers. All algorithms compute diagnoses of minimum cardinality. In particular, the algorithms are increasing the size of the diagnoses until they get first results. They only search for diagnoses containing two cells if there does not exist any diagnosis containing only a single cell. While Choco and Minion compute all possible values for the abnormal variables at once, Z3 must be called for each abnormal combination separately. All solvers work with the same domain for their variables. The domain is fixed to [-2000, 5000]. The prototype directly uses the API calls of Choco and Z3. Since Minion does not provide an application programming interface, the Minion constraints are written to a file and the Minion solver is started in a separate process. For the following runtime comparison, we only measure the time Minion, Choco and Z3 require for solving the given Csp. The time for writing to the file is not included. If a solver has to be called several times, the solving times are summed up.

In the first part of the evaluation, we compared the computation time of Z3, Choco and Minion. For Z3, we used the MCSes and MCSes-U algorithms for computing the diagnoses. Since the constraint solvers are only able to handle Integers, we used the Integer Spreadsheet Corpus from Section 8.3 for this part of the evaluation. This evaluation was performed on a computer with an Intel Core i5-2500T (2,3GHz quadcore) processor and 8GB of RAM. The prototype was running on a 32-Bit Java VM 1.7.0 Update 11 within a 64-Bit version of Windows 7. For each faulty spreadsheet, we computed the average solving time over 100 runs. In order to keep the time required for the evaluation small, we set a time limit of 5 minutes for Z3 and 20 minutes for the others. For Z3, less than 5 % of the spreadsheets did not finish. For the other algorithms, the range of timeouts was between 11 % (Choco with cone) and 36 % (Minion without cone).

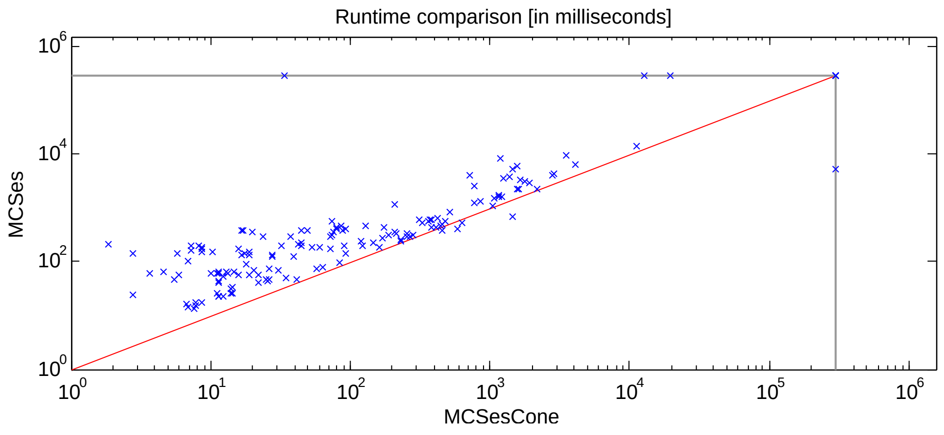
The Figures 11.2, 11.3 and 11.4 graphically compare the performance of the approaches. Figures 11.2a, 11.2b, 11.2c, and 11.3a compare the performance of the single solvers and methods w.r.t. the amount of information, the solver has available. Except for the MCSes-U algorithm (Figure 11.3a), the figures clearly show that using cones yields a shorter solving time. For the MCSes-U algorithm, the average runtimes (831.3 milliseconds without cone and 675.4 milliseconds with cone information) indicate that using only the cone information increases the performance. However, using only the cone information in the MCSes-U algorithm more often results in timeouts than using the whole information of the spreadsheet. Figure 11.2a shows that Minion often cannot finish the computations. The Figures 11.3b, 11.3c, 11.4a, and 11.4b compare the different solvers and methods. Z3 using MCSes-U is on average the fastest approach. However, the difference between MCSes and MCSes-U (Figure 11.3c) is marginal compared to the differences to the constraint solvers (Minion: 4,340.3 milliseconds, Choco: 4,029.7 milliseconds on average). This means that Z3 is on average six times faster than the



(a) Minion without versus Minion with cones

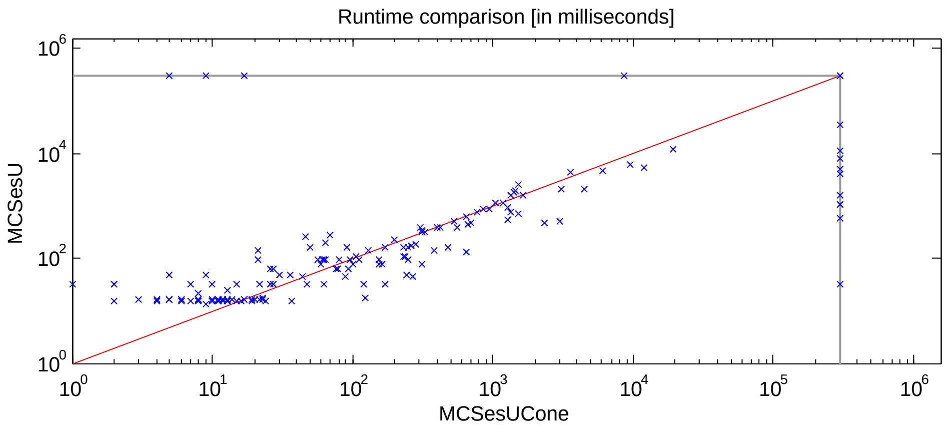


(b) Choco without versus Choco with cones

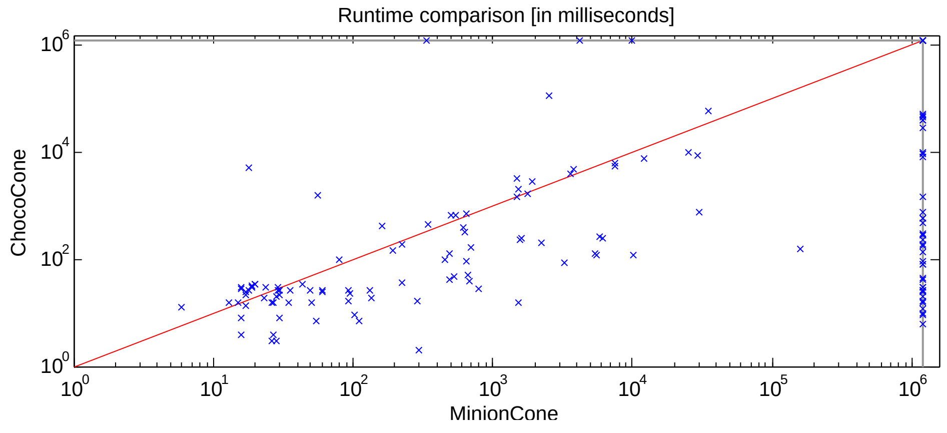


(c) Z3 (MCSes) without versus Z3 (MCSes) with cones

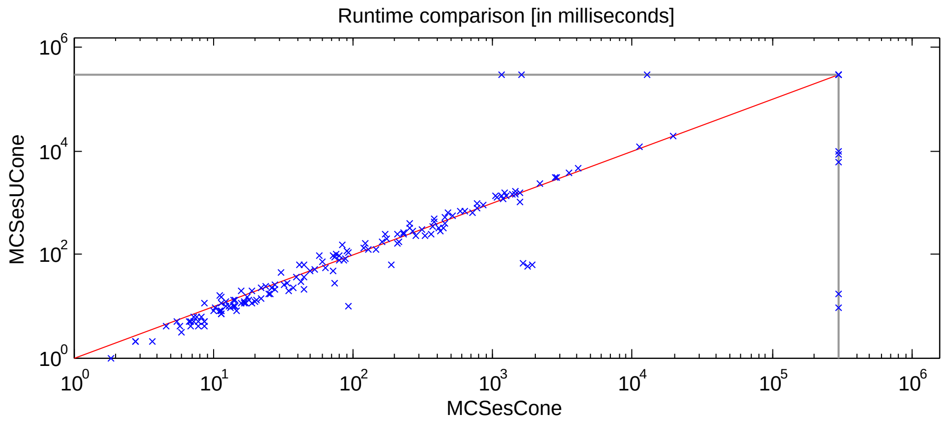
Figure 11.2.: Runtime comparison on basis of the Integer spreadsheet corpus - Part 1. Data points near to the red line indicate that the approaches perform equal (w.r.t. runtime). Data points above the red line indicate that the approach labeled on the x-axis performs better. The grey lines indicate the timeout limits.



(a) Z3 (MCSes-U) without versus Z3 (MCSes-U) with cones

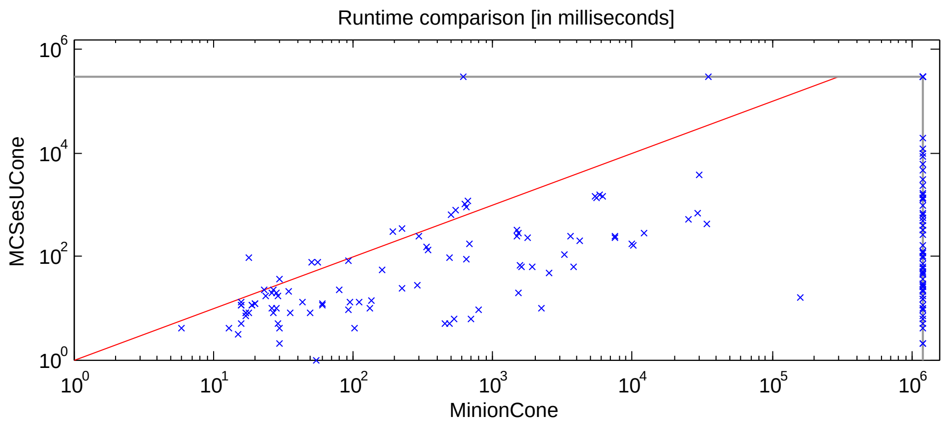


(b) MINION versus Choco

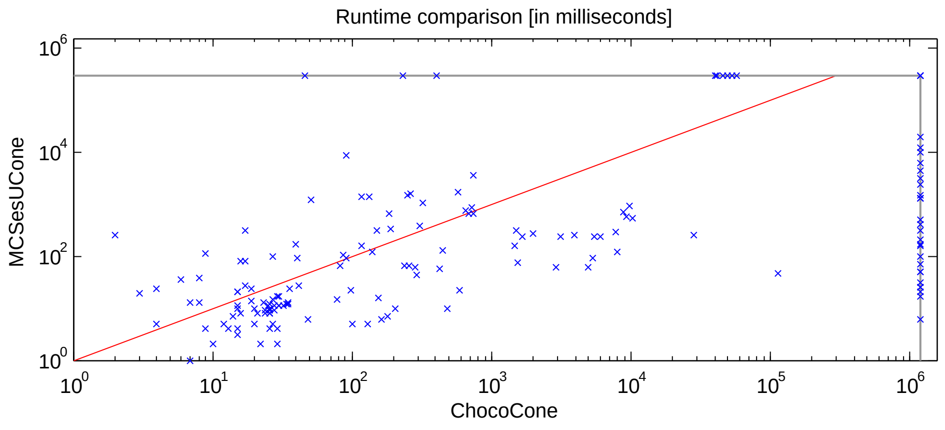


(c) Z3 (MCSes) versus Z3 (MCSes-U)

Figure 11.3.: Runtime comparison on basis of the Integer spreadsheet corpus - Part 2. Data points near to the red line indicate that the approaches perform equal (w.r.t. runtime). Data points above the red line indicate that the approach labeled on the x-axis performs better. The grey lines indicate the timeout limits.



(a) MINION versus Z3 (MCSes-U)



(b) Choco versus Z3 (MCSes-U)

Figure 11.4.: Runtime comparison on basis of the Integer spreadsheet corpus - Part 3. Data points near to the red line indicate that the approaches perform equal (w.r.t. runtime). Data points above the red line indicate that the approach labeled on the x-axis performs better. The grey lines indicate the timeout limits.

constraint solvers. When considering only the constraint solvers (Figure 11.3b), the performance difference of Minion and Choco is marginal. When using the cone information (Figure 11.3b), Choco performs slightly better than Minion, otherwise Minion performs better than Choco.

In this part of the empirical evaluation, we showed that Z3 outperforms Minion and Choco concerning runtime and modeling abilities. Our prototype contains optimizations for all three solvers. However, the following threats to validity remain: Restricting the bounds of Integers is essential for the runtime of constraint solvers. Currently, we used fixed bounds, which ensures both the support for our evaluation examples and a reasonable runtime. However, relaxing the bound might increase the computation times.

In the second part of the evaluation, we investigate the performance w.r.t. the runtime of Z3 when dealing with Real numbers. Therefore, we replaced all input values in the spreadsheets of the Integer spreadsheet corpus (Integer SC) with Real numbers. We refer to this new corpus as Real numbers spreadsheet corpus (or Real SC). We run Z3 with the MCSes-U algorithm using cones, on both the Integer SC and the Real SC 25 times. Table 11.1 shows the results of this comparison. Debugging of spreadsheets containing Real numbers lasts on average 2.6 times longer than debugging of Integer spreadsheets. Please note that the number of investigated spreadsheets in Table 11.1 differs from the average given in the above discussion because we excluded spreadsheets leading to timeouts from the evaluation.

Table 11.1.: Analysis of runtime performance of Z3 using cones with MCSes-U for the Integer spreadsheet corpus and the Real numbers spreadsheet corpus

	Integer SC [ms]	**Real SC [ms]**
avg	332	872
median	12	46
stdev	1,530	2,710

Besides the time analysis, we show that model-based debugging is a valuable help when debugging spreadsheets. Therefore, we take a look at the quality of the debugging results. We express quality by means of the Reduction metric defined in Equation 10.1.

For this part of the evaluation, we used spreadsheets from the modified Euses spreadsheet corpus described in Section 8.2. Since this corpus contains spreadsheets with Real numbers, this evaluation is only performed with Z3. As already mentioned, our prototype does not support all functionality provided by standard spreadsheet tools. Therefore, the evaluation is only performed on 183 spreadsheets of the corpus. From these spreadsheets, 12 spreadsheets resulted in a timeout (5 minutes). The results for the other

171 spreadsheets are summarized in Table 11.2. The given runtime is the average over 25 runs.

Table 11.2.: Analysis of execution for Z3 using cones with MCSes-U for the modified EUSES spreadsheet corpus

	Runtime [ms]	**Reduction [%]**
average	488.6	79.7
median	5.0	97.0
stdev	2,763.6	68.2

From these empirical evaluations, we conclude that using Z3 for debugging is on average faster than using Minion or Choco. In addition, Z3 is able to handle Real numbers. Furthermore, we showed that such an model-based approach is able to reduce the amount of cells that must be inspected.

11.5. Conclusion

In this chapter, we proposed to use SMT solvers for solving the spreadsheet debugging problem. We showed the conversion of spreadsheet formulas into constraints or formula clauses respectively for Choco, Minion and Z3. In addition, we performed a case study showing the superiority of Z3 over Choco and Minion w.r.t. the runtime. Z3 with the MCSes-U algorithm is six times faster than Choco and Minion for finding all diagnoses with lowest cardinality. This runtime improvement makes it possible to use Mbsd for interactive debugging. Another important advantage of Z3 over Minion and Choco is the handling of Real numbers. Z3 supports the usage of Real numbers without any restrictions. However, when using Z3, the computation of diagnoses for spreadsheets containing Real numbers requires 2.6 times more runtime than the computation of diagnoses for similar spreadsheets with Integer values only. In addition, we evaluated the performance of our SMT prototype on the modified EUSES spreadsheet corpus w.r.t. runtime and reduction quality. On average, the prototype requires 488.6 milliseconds for computing the diagnoses for a EUSES spreadsheet. With the help of our prototype the amount of cells that must be manually inspected is reduced by 79.7 % on average.

12. Mutation Supported Spreadsheet Fault Diagnosis

This chapter is based on work that is submitted for publication [Auß+].

12.1. Introduction

Constraint-based fault localization techniques (e.g. [JE10; ARW12]) determine which cells might be faulty. When the user changes all cells that are contained in a diagnosis in the right way, the user could repair the spreadsheet. Unfortunately, the amount of diagnoses can be huge. In addition, such diagnoses cannot give the user hints how to change the cells. We are confident that the information how the cells should be changed is an important support for the user. Therefore, we propose a novel approach that automatically creates versions, i.e., mutants, of the spreadsheet that satisfy the given test case. This approach is called MuSSCO, short for Mutation Supported Spreadsheet Correction. As the number of the generated mutants can be large, MuSSCO automatically computes distinguishing test cases to eliminate mutants that are invalid corrections. A test case is a distinguishing test case if there is at least one output variable where the computed value of two versions of a spreadsheet differ on the same input. We provide a prototype implementation of MuSSCO[1] which handles the basic functionality of spreadsheets.

The remainder of this chapter is organized as follows. In Section 12.2, we discuss the related work. The mutants creation process is explained in Section 12.3. The algorithms for creating diagnoses and distinguishing test cases are explained in Section 12.4. We provide an empirical evaluation in Section 12.5 and discuss threats to validity and future work in Section 12.6. In Section 12.7, we conclude the approach.

[1] http://mussco.ist.tugraz.at/

12.2. Related Work

MuSSCO is based on work of Nica *et al.* [NNW10; NNW12]. Nica *et al.* compute distinguishing test cases for debugging imperative programs. Their approach relies on μ-Java [MOK06] for generating the correction suggestions. In contrast to their approach, MuSSCO encodes the mutation operators directly into the constraint satisfaction problem.

GoalDebug [AE07a; AE05; AE08] employs a similar constraint-based approach and computes a list of changes to fix the spreadsheet. GoalDebug relies upon a set of possible, pre-defined change (repair) inference rules. The fault localization approach is done by mutating the spreadsheet using the set of rules and ascertain that the user expectations are met. GoalDebug uses heuristics to select the most suited mutants. In contrast, MuSSCO generates distinguishing test cases to filter out those mutants that are not valid corrections.

Weimer *et al.* [Wei+09] introduced genetic programming for repairing C programs. Similar to them, we make assumptions how to restrict the search space. For example, we perform mutations on the cone and Weimer *et al.* make mutations on the weighted path. In addition, Weimer *et al.* assume that the correct statement exists somewhere in the code. We assume that when a spreadsheet programmer referenced the wrong cell, the correct cell is in the surrounding of the referenced cell. However, we differ from their genetic programming approach as we do not use crossover and randomness for selecting mutations.

12.3. Mutation creation

A primitive way to compute mutants is to clone the spreadsheet and change arbitrary operators and operands in all formulas of the cells contained in one diagnosis. If the created mutant satisfies the given test case we present the mutant to the user. Otherwise we discard the mutant and create another mutant. The problem with this approach is that too many mutants have to be computed until the first mutant passes the given test case. Therefore, we propose a more sophisticated approach which includes the mutation creation process in the Csp. Instead of only transforming cell formulas into a value-based constraint model, we also include the information how the cells could be mutated. We allow the following mutation operations:

- replace a constant with a reference or another constant
- replace a reference with a constant or another reference
- replace arithmetical operators with other arithmetical operators

- replace relational operators with other relational operators
- replace function with other functions that take the same number of arguments
- resize areas

We are aware that these mutation operators are not able to correct all faulty spreadsheets. In particular, the creation of completely new formulas is up to future work.

When creating mutants, we have to face two challenges: (1) We have to ensure that the created mutant is a feasible spreadsheet, i.e., the DDG of the mutant must not contain any cycles. (2) Theoretically, an infinite number of mutations can be created. Therefore, we have to restrict the search space for the mutant creation.

In order to handle the first challenge, we propose the following solution: Each cell that is represented in the CSP gets an additional Integer variable within the domain $\{1, |\Pi|\}$. The constraint solver has to assign values to these variables in such a way that each cell gets a number that is higher than the numbers assigned to the cells this cell references. This constraint ensures that the created mutant is still a feasible spreadsheet.

In order to reduce the search space for mutations, our approach makes the following assumptions:

- Mutations are only indicated for cells that are contained in the cone of any erroneous output cell.
- When replacing references with constants, we do not immediately compute the concrete constant. Instead, we just use the information that there exists a constant that could eliminate the observed misbehavior. Only if we present a mutant to the user, we compute a concrete value for that constant. The reason for this delayed computation is the fact that there often exist many constants that satisfy the primary test case. During the distinguishing test case creation process, we gain additional information. Therefore, we can reduce the number of constants.
- In cases where we have to change a reference or resize an area, we make use of the following assumption: If the user made a mistake when indicating the reference or area, the intended reference(s) might be in the surrounding of the originally indicated reference(s). In case of a single reference, we define the surrounding of a cell c as follows:

$$\textsc{Surrounding}(c) \equiv_{def}$$

$$\left\{ c_1 \in \text{CELLS} \,\middle|\, \begin{array}{l} \varphi_x(c) - 2 \leq \varphi_x(c_1) \leq \varphi_x(c) + 2 \;\; \& \\ \varphi_y(c) - 2 \leq \varphi_y(c_1) \leq \varphi_y(c) + 2 \end{array} \right\} \tag{12.1}$$

We model into our Csp that the reference to the cell is either correct or that it should be replaced by one of the cells in the surrounding. In case of an area, we define the surrounding of the area as follows:

$$\textsc{Surrounding}(c_1 : c_2) \equiv_{def}$$

$$\left\{ c_3 \in \text{CELLS} \,\middle|\, \begin{array}{l} \varphi_x(c_1) - 2 \leq \varphi_x(c_3) \leq \varphi_x(c_2) + 2 \;\; \& \\ \varphi_y(c_1) - 2 \leq \varphi_y(c_3) \leq \varphi_y(c_2) + 2 \end{array} \right\} \tag{12.2}$$

For areas, we allow to select/deselect any cell in the surrounding. This allows shrinking and enlargement of areas on the one hand, and non-continuous areas on the other hand.

These assumptions on the search space do not allow to find suited mutants for all given faulty spreadsheets. Furthermore, only one mutation per cell is allowed. However, these restrictions enable to use the approach in practice.

12.4. Computing distinguishing test cases

In the previous section, we have discussed how to generate mutants for a given faulty spreadsheet Π that satisfy a given test case t. Usually, there exists more than one possible correction. In practice, a large number of repair suggestions overwhelms the user. Consequently, there is a strong need for distinguishing such variants. One way to distinguish explanations is to use distinguishing test cases. Nica *et al.* [NNW12] define a distinguishing test case for two variants of a program as input values that lead to the computation of different output values for the two variants. When translating this definition to the spreadsheet domain, we have to search for constants that are assigned to inputs, which lead to different output values for the different explanations. The user (or another oracle) has to clarify which output values are correct.

Algorithm 12.1 describes our approach. The algorithm takes a faulty spreadsheet and a failing test case as input and determines possible solutions with increasing cardinality, starting with a solutionSize of 1 (Line 1). Since input cells are considered correct, the upper bound of the solutionSize is equal to the amount of non-input cells. In Line 2, the set T initialized with the given failing test case. In the Lines 5 and 6, we create the sets eqMut and undesMut

Algorithm 12.1 Mussco(Π, t)

```
Require: A spreadsheet $\Pi$, a test case $t$
Ensure: A set of possible corrections
 1: solutionSize = 1
 2: T = $\{t\}$
 3: while solutionSize $\leq (|\Pi| - |$GetInputCells$(\Pi)|)$ do
 4:   $M = \{\}$
 5:   eqMut ={}
 6:   undesMut ={}
 7:   CON =ConvertSpreadsheet($\Pi$, T)
 8:   CON = CON $\cup$ GetSizeConstr(CON, solutionSize)
 9:   while HasSolution(CON) do
10:     m = GetMutant(CON)
11:     $M = M \cup \{m\}$
12:     CON = CON $\cup\{\neg m\}$
13:     while $|M| \geq 2 \wedge \exists((m_1, m_2) \in M : (m_1, m_2) \notin$ eqMut $\wedge$ $(m_1, m_2) \notin$
        undesMut) do
14:       Select two mutants $m_1, m_2$ from $M$ where $(m_1, m_2) \notin$ eqMut $\wedge$
          $(m_1, m_2) \notin$ undesMut
15:       $t'$ = GetDistinguishingTestcase($\Pi, m_1, m_2$)
16:       if $t'$ = UNSAT then
17:         eqMut = eqMut $\cup \{(m_1, m_2)\}$
18:       else
19:         if $t'$ = UNKNOWN then
20:           undesMut = undesMut $\cup \{(m_1, m_2)\}$
21:         else
22:           $t' = t' \cup$ GetExpectedOutput($\Pi, t$)
23:           T = T $\cup \{t'\}$
24:           CON = CON $\cup$ ConvertSpreadsheet($t'$)
25:           $M'$ = Filter($\Pi, t', M$)
26:           $M = M \setminus M'$
27:         end if
28:       end if
29:     end while
30:   end while
31:   if User accepts any solution in $M$ then
32:     return $M$
33:   end if
34:   solutionSize = solutionSize + 1
35: end while
36: return no solution
```

to store the pairs of equivalent and undecidable mutants. The faulty spreadsheet and the given test cases are converted into constraints in Line 7. At this, the function ConvertSpreadsheet is similar to the function described in Algorithm 10.1. However, instead of only converting an expression into its constraint representation, also possible mutations are encapsulated in the constraint representation. The function GetSizeConstraint(CON, n) creates a constraint that ensures at most n of the abnormal variables contained in CON can be set to true (Line 8).

In Line 9, the function HasSolution checks if the solver can compute any mutants that satisfy the given constraint system. In Line 10, the function GetMutant returns a mutant that satisfies the given constraint system. This mutant is added to the list of mutants M (Line 11) and is blocked in the constraint system (Line 12). If M contains at least two mutants that are not equivalent or undecidable, such a pair of mutants is selected (Line 14). In Line 15, we call the function GetDistinguishingTestcase. If this function returns UNSAT, the pair m_1, m_2 is added to the set eqMut (Line 17). If the function returns UNKNOWN, the pair m_1, m_2 is added to the set undesMut (Line 20). Otherwise, the function returns a new test case. The function GetExpectedOutput is used to determine the expected output for the given test case (Line 22). This function either asks the user or another oracle, e.g. a correct implementation of the spreadsheet. The test case is added to the set of test cases (Line 23). Furthermore, it is converted into constraints and added to the constraint system (Line 24). The function Filter checks for each remaining mutant in M, if it passes the new test case (Line 25). This function returns the set of mutants that fail this test case. Those mutants are removed from the set of mutants (Line 26). After retrieving all mutants for the given solutionSize, the user is presented with the remaining solutions M. If the user accepts at least one found mutant, the algorithm terminates. Otherwise, the solutionSize is incremented (Line 34) in order to determine possible mutants with an increased size.

Algorithm 12.2 describes the creation of distinguishing test cases. This algorithm takes as input a spreadsheet and two mutated versions of that spreadsheet. In the Lines 1 and 2, the functions GetInputCells and GetOutputCells are called. These functions return the set of input and output cells for the given spreadsheet. In Line 3, mutant m_1 is converted into its constraint representation. When creating a distinguishing test case, we have to exclude the input cells from the spreadsheet. Therefore, we only hand over the spreadsheet without the input cells to the function ConvertSpreadsheet. This function slightly differs from the ConvertSpreadsheet function used in Algorithm 12.1, because it takes two additional parameters: (1) the particular mutant in use and (2) a constant that acts as postfix for variables. This postfix is necessary to distinguish the constraint representation of m_1 from that of

m_2: Each variable in the constraint system for mutant m_1 gets the postfix "_1". Mutant m_2 is converted in Line 4 into its constraint representation using the postfix "_2". In Line 5, a constraint is created that ensures that the input of the m_1 is equal to the input of m_2. In Line 6, a constraint is created that ensures that at least one output cell of m_1 has a different value than the same output cell in m_2. The function GetSolution calls the solver with these constraints (Line 8). This function either returns a distinguishing test case, UNSAT (in case of equivalent mutants) or UNKOWN (in case of undecidability).

Algorithm 12.2 GetDistinguishingTestCase(Π, m_1, m_2)

```
Require: A spreadsheet $\Pi$, mutants $m_1, m_2$
Ensure: A distinguishing test case or UNSAT/UNKOWN
 1: inputCells = GetInputCells($\Pi$)
 2: outputCells = GetOutputCells($\Pi$)
 3: CON1 =ConvertSpreadsheet($\Pi \setminus$ inputCells, $m_1$, "_1")
 4: CON2 =ConvertSpreadsheet($\Pi \setminus$ inputCells, $m_2$, "_2")
 5: inputCon = $\bigwedge_{c \in inputCells} c\_1 = c\_2$
 6: outputCon = $\bigvee_{c \in outputCells} c\_1 \neq c\_2$
 7: CON = CON1 $\cup$ CON2 $\cup$ inputCon $\cup$ outputCon
 8: return GetSolution(CON)
```

The overall worst-case time complexity of Algorithm 12.1 is exponential in the number of cells ($O\left(2^{|\text{CELLS}|}\right)$). In practice, only solutions up to a certain size, i.e., single or double fault solutions, are relevant. Obviously, Algorithm 12.1 terminates. The outer while-loop (Line 3) is bound to the size of the spreadsheet. The while-loop in Line 9 is limited since there only exists a limited number of mutants that can be created and we do not allow to report mutants twice (Line 12). The inner-most loop (Line 13) is limited since the number of mutants in M has to be greater or equal to two and the selected pair must not have been proven to be equivalent or undecidable. In each iteration of this loop, either a new pair is added to the equivalent or undecidable set (Lines 17 and 20) or the set M shrinks (Line 26). M must shrink because the return set of the function Filter (Line 25) contains at least one element, since the mutants m_1 and m_2 must compute different output values for the given test case.

12.5. Empirical Evaluation

We implemented a prototype in Java that uses Apache POI[2] to access spreadsheets and Z3 [MB08] as solver. This prototype supports the conversion of

[2] http://poi.apache.org/index.html

spreadsheets with basic functionality into Z3 formula clauses. Arithmetic and relational operators as well as the functions 'IF', 'SUM', 'AVERAGE', 'MIN', and 'MAX' are supported.

For the empirical evaluation, we used an extended version of the Integer Spreadsheet corpus described in Section 8.3. The evaluated spreadsheets can be found on the MuSSCO website[3]. We have to exclude some spreadsheets of the corpus: For 102 mutated spreadsheets, the algorithm did not terminate within 20 minutes. For 54 mutated spreadsheets, MuSSCO could not compute the required corrections. In most of these cases, MuSSCO was able to generate mutants that satisfy the given initial test case. However, MuSSCO was not able to generate a mutant that equals to the original non-faulty spreadsheet because of the following reasons: (1) The correction requires more than one mutation within a single cell, which is currently not supported by our approach. (2) The required mutation operator is not implemented in MuSSCO or the mutation operator does not cover all particular constellations. For 73 mutated spreadsheets, MuSSCO was able to compute the required mutation in order to correct the fault. In the following empirical evaluation, we only consider these 73 mutated spreadsheets.

The smallest spreadsheet used in this evaluation contains 8 formulas and the largest contains 69 formulas cells. On average, a spreadsheet contains 31.2 formula cells. The faulty spreadsheet variants have on average 1.14 erroneous output cells. 52 mutated spreadsheets contain single faults. 20 mutated spreadsheets contain double faults, i.e., two cells with wrong formulas. One mutated spreadsheets contains three faults. The evaluation was performed using a PC with an Intel Core i7-3770K CPU and 16GB RAM. The evaluation machine runs a 64-bit Windows 7, at which MuSSCO is executed in the Oracle Java Virtual Machine version 1.7.0_17. The evaluation results are averaged over 100 runs.

Our approach is designed to interact with the user. In order to investigate a larger amount of spreadsheets, we decided to simulate the user interactions. Therefore, we use the original correct spreadsheets as oracles to determine the output values for the the generated distinguishing test cases. Figure 12.1 shows the amount of correction suggestions that are returned to the user. For 49 spreadsheets, only the correct mutation is returned to the user. On average, 3.2 mutants are reported to the user. For one faulty spreadsheet containing two faulty cells, MuSSCO determines 27 correction suggestions. Moreover, applying the algorithm to a spreadsheet with three faults results in 94 correction suggestions. The evaluation shows that in case of double or triple faults, MuSSCO finds a higher amount of equivalent solutions.

[3]http://mussco.ist.tugraz.at/

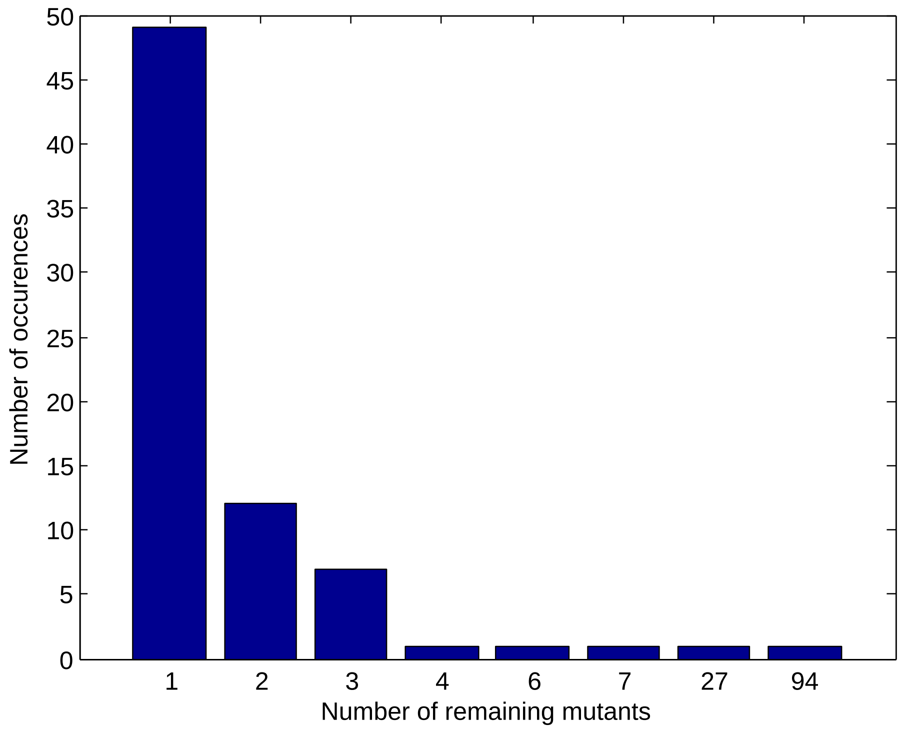


Figure 12.1.: Amount of correction suggestions returned to the user

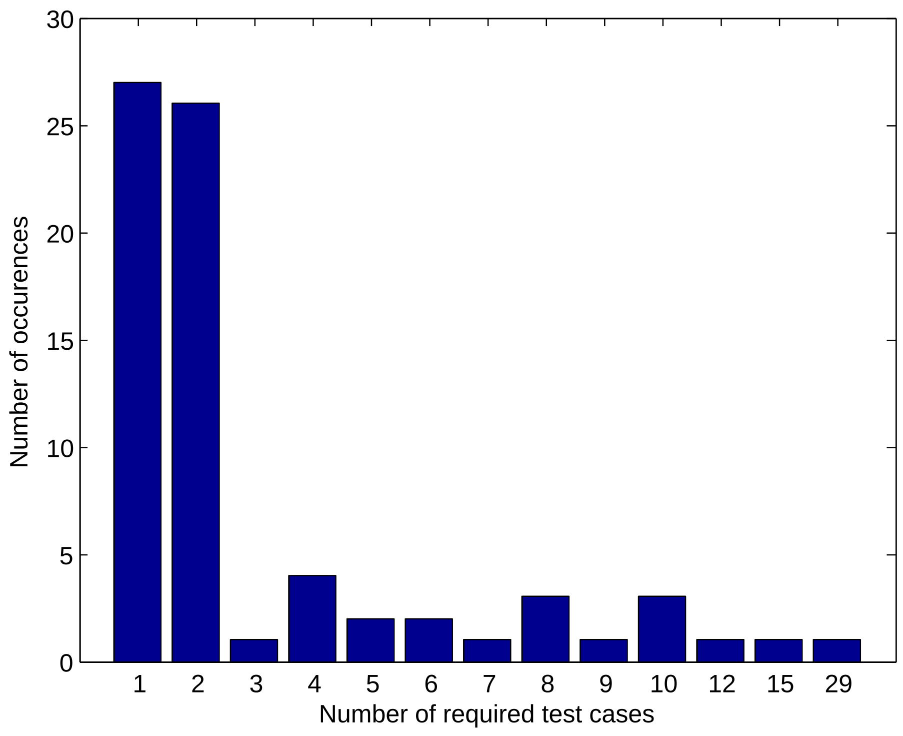


Figure 12.2.: Amount of generated distinguishing test cases

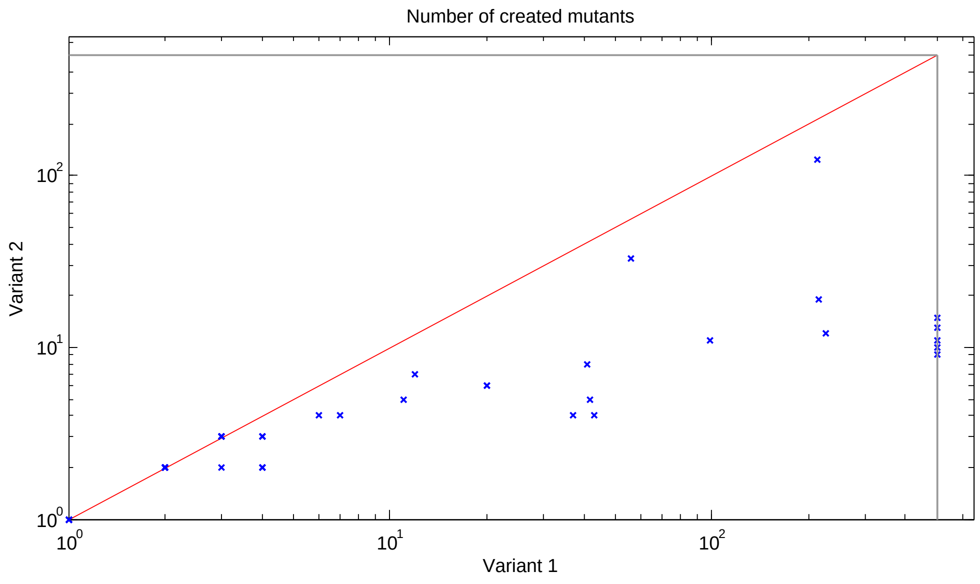
Figure 12.2 illustrates the number of generated distinguishing test cases. For 27 spreadsheets, only a single distinguishing test case is required. For 26 spreadsheets, two distinguishing test cases are necessary. For one spreadsheet, 29 distinguishing test cases have to be generated. This spreadsheet contains a double fault. Therefore, MuSSCO creates many mutants which have to be removed by the distinguishing test cases. On average, 3.1 distinguishing test cases are required.

The average runtime is 49.1 seconds, at which the runtime is less than 10 seconds for 23 of the spreadsheets. The average runtime for single faults is 25.1 seconds. The average runtime for double and triple faults is 108.6 seconds. Most of the runtime, i.e. 95.5 % is consumed by the mutation creation process. The creation of the distinguishing test cases requires on average 1.4 % of the total run time. The remaining 3.1 % share out between the time required for filtering the mutants and setting up MuSSCO (read spreadsheet data in, convert spreadsheet).

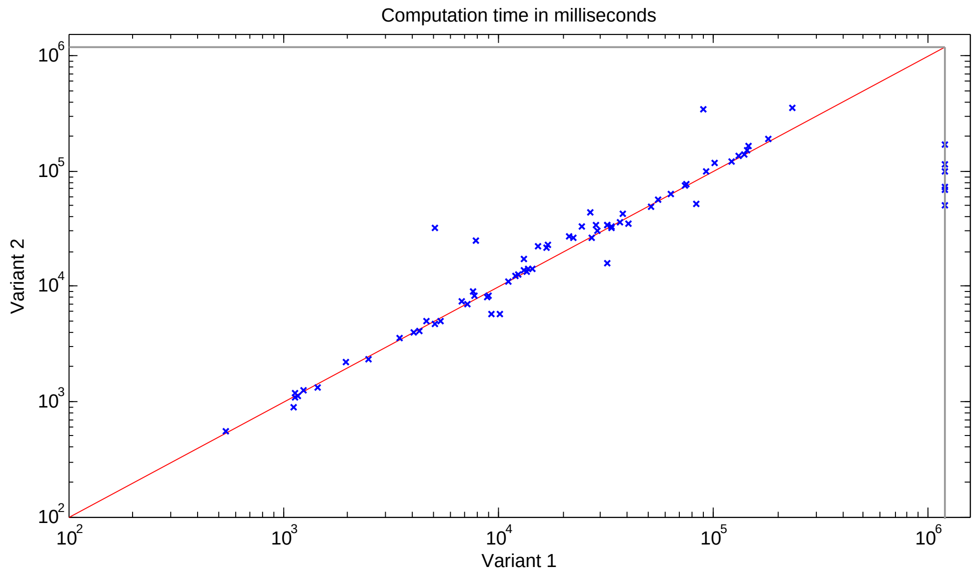
We create a distinguishing test case as soon as we have two mutants available. Another possibility is to immediately compute all possible mutants of a particular size and afterwards generate the test cases. Does the implemented method perform better with respect to runtime? We suppose that more adding more test cases to the constraint system decreases the number of mutants that are created and therefore decreases the total computation time. For clarifying our assumptions, we compare the two methods with respect to the number of generated mutants and the total computation time in Figure 12.3. Variant 1 denotes the version where we first compute all possible mutants. Variant 2 denotes the version described in Algorithm 12.1. For six spreadsheets, Variant 1 results in a timeout. From Figure 12.3a, we can see that Variant 1 obviously creates more mutants. On average, Variant 1 creates 17.2 mutants while Variant 2 creates 5.2 mutants (when comparing only those spreadsheets without timeouts). When comparing the computation time (Figure 12.3b), the two variants only slightly differ (expect for the six spreadsheets yielding a timeout when using Variant 1). It turns out, that decreasing number of computed mutants through more test cases, increases the computation time per mutant. Nevertheless, we favor Variant 2 over Variant 1 since the user gets a first response earlier.

12.6. Threats to validity and future work

The largest threat to validity is the user oracle. In this empirical evaluation, we had the original spreadsheet as oracle at our disposal and therefore complete and correct knowledge. This is not the case in real life where the user acts as oracle. Unfortunately, to err is human. Similar as the user made a mistake



(a) Comparison of two computation variants with respect to the number of created mutants



(b) Comparison of two computation variants with respect to the total computation time

Figure 12.3.: Comparison of two computation variants with respect to the number of created mutants and the total computation time. Data points close to the red line indicate that the variants behave equal. Data points below the red line indicate that Variant 1 creates more mutants (Subfigure 12.3a). The grey lines indicate the timeout.

in a formula, he might indicate wrong values as expected output values. Currently, our approach is not robust enough to handle wrong user input. In future work, we will expand our approach by an heuristic model. This heuristic model takes into consideration that user answers might be wrong.

Another important topic is the solving time. As mentioned in the empirical evaluation, we had to exclude several spreadsheets from the evaluation because the solving process results in a timeout. In future work, we are going to improve our mutation strategy. We are confident that an improved mutation strategy paves off for a reduced solving time. In addition, we want to predict the solving time. For the prediction of the solving time, we are going to rely on previous work on estimating the complexity of constraint satisfaction problems [Wot+09]. With a predicted solving time, we are able to advice the user for/against using this approach for a concrete spreadsheet. In case of a large solving time, we could ask the user for more information and some intermediate values in order to restrict the search space.

In addition, we plan to extend the empirical evaluation by conducting a user study to ascertain the effectiveness of our approach. The user study is important to study usability aspects in order to efficiently communicate the diagnostic results to the end-users.

12.7. Conclusions

In this chapter, we present an approach for fault localization and correction in spreadsheets. Our approach, coined MuSSCO, converts the spreadsheet under analysis into a set of constraints. Moreover, mutations are added to the constraint system. This allows to generate mutants as possible repair suggestions that satisfy the given test case. As the number of mutants that fulfill a primary test case can be huge, distinguishing test cases are computed. This allows to narrow the set of possible repair suggestions.

Beside the theoretical foundations and the algorithms we also discuss the results obtained from an empirical evaluation where we are able to show that distinguishing test cases improve diagnosis of spreadsheets substantially. In particular, results show that on average 3.1 distinguishing test cases are generated and 3.2 mutants are reported as possible fixes. On average, the generation of the mutants and distinguishing test cases requires 47.9 seconds in total, rendering the approach applicable as a real-time application.

Part IV.

Future Work and Conclusion

13. Future Work and Conclusion

In this thesis, we have proposed several approaches that improve the state-of-the-art of fault localization in both, 3rd generation languages and the spreadsheet domain. Most of these approaches are model-based software debugging approaches. Sfl for spreadsheets from Chapter 9 is the only technique that does not rely on a model. The other approaches are illustrated in Figure 13.1. For 3rd generation languages, we have proposed two approaches, namely Sendys and Conbas. Conbas relies on a value-based model. It is computationally demanding and therefore only suited for debugging small programs. In contrast, Sendys relies on a dependency-based model. This makes Sendys to a lightweight approach that can also be used for debugging large programs. While Conbas returns a set of diagnosis candidates, Sendys returns a ranked list of suspicious statements. For the spreadsheet domain, we have proposed three model-based approaches, namely Sendys for spreadsheets, ConBug, and MuSSCO. While Sendys for spreadsheets relies on a dependency-based model, ConBug, and MuSSCO make use of a value-based model. Therefore, ConBug and MuSSCO are computationally more demanding than Sendys for spreadsheets. Sendys for spreadsheets returns a ranked list of suspicious cells. ConBug returns a set of diagnosis candidates. MuSSCO not only returns a set of diagnosis candidates but also correction suggestions.

Domain Model	3rd Generation Programs	Spreadsheets
Dependency-Based	SENDYS	Sendys for Spreadsheet
Value-Based	CONBAS	ConBug MuSSCO

Figure 13.1.: Classification of the model-based approaches presented in this thesis

We have shown in the empirical evaluations that the previously presented techniques are a valuable support when debugging programs or spreadsheets. However, there exists many open challenges.

For both domains, the integration of the introduced approaches into tools is vital. For the 3rd languages, debugging tools have to be integrated into IDEs like Eclipse. For the spreadsheet domain, the debugging tools have to be directly integrated into the spreadsheet tools, e.g. Mircosoft's Excel. The technical integration is easy. The greatest challenge lies in human-computer interaction. Most of the effort has to be spent for designing the interfaces. In addition, user studies are necessary to show if such debugging tools are a valuable help for users.

Another challenge is the detection of missing code / formulas. The proposed methods do not explicitly point out that the error might be caused by missing code or missing formulas. However, most researches (e.g. Abreu and van Gemund [AG10] and Wong *et al.* [WDC10]) argue that ranking based methods can still be a valuable help when debugging: When examining suspicious statement, the programmer (spreadsheet developer) may realize that some code in the neighborhood of the suspicious statement is missing. In future work, we are going to investigate by means of a user-study if users realizes that the error is caused by missing code or missing formulas when presenting diagnoses to the user.

For all approaches that rely on value-based models, the computational complexity is a hot topic. Unfortunately, all of these approaches suffer from a scalability problem. They can only be used for small programs and spreadsheets. In contrast, dependency-based model can be used for large programs, but they cannot compute as good results as value-based models. In this thesis, we have made first steps towards improving the fault localization of lightweight debugging approaches without dramatically increasing their computational complexity. However, there is still a long way to go for providing good automated debugging techniques. But the combination of spectrum-based techniques with model-based techniques seems to be a good path in the right direction.

From this thesis, we have learned that it makes sense to use debugging techniques developed for 3rd generation languages in the spreadsheet domain. In the domain of 3rd generation languages, there exist many techniques whose complexity prevents programmers to use these techniques on real-life programs. As the spreadsheet domain is more restricted (e.g. there exist no loops), it is possible to apply these approaches to spreadsheets. In the spreadsheet domain, we investigated the impact of using different solvers (constraint solvers versus SMT solvers) for solving the debugging problem. We have learned that Z3, a state-of-the-art SMT solver, performs best w.r.t. runtime. In future work, we are going to investigate if Z3 is also successful when debugging programs written in a 3rd generation language.

Appendix

Appendix A.

List of publications

A.1. Published

1. Birgit Hofer, André Riboira, Franz Wotawa, Rui Abreu, and Elisabeth Getzner. "On the Empirical Evaluation of Fault Localization Techniques for Spreadsheets." In: *Fundamental Approaches to Software Engineering - 16th International Conference, FASE 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013*, pp. 68–82.
   I had the idea for this paper. Franz Wotawa wrote the part about the basic definitions. André Riboira and Rui Abreu wrote about Spectrum-based fault localization. Elisabeth Getzner helped me to implement the prototype and to perform the empirical evaluation. The Chapters 7, 8, and 9 are based on this paper.
2. Birgit Hofer and Franz Wotawa. "Reducing the Size of Dynamic Slicing with Constraint Solving." In: *Proceedings of the 12th International Conference on Quality Software*, pp. 41-–48.
   Franz Wotawa had the basic idea for this paper. I refined the algorithm and implemented a prototype based on the proof-of-concept implementation of a student. In addition, I performed the empirical evaluation. Chapter 5 is based on this paper.
3. Birgit Hofer and Franz Wotawa. "Combining Slicing and Constraint Solving for Better Debugging: The CONBAS Approach." In: *Advances in Software Engineering, vol. 2012*, Article ID 628571, 18 pages, 2012.
   This is an extension of the previously mentioned paper. Franz Wotawa focused on the basic definitions, while I wrote about the algorithm and extended the empirical evaluation. The Chapters 3 and 5 are based on this paper.
4. Birgit Hofer and Franz Wotawa. "Spectrum Enhanced Dynamic Slicing for better Fault Localization." In: *Proceedings of the 20th European Conference on Artificial Intelligence (ECAI 2012)*. Vol. 242. Frontiers in Artificial Intelligence and Applications. IOS Press, pp. 420–425. ISBN: 978-1-61499-097-0.

I had the idea for this paper and implemented the prototype. Franz Wotawa helped in writing and proof-reading the paper. The Chapters 3 and 4 are based on this paper.

5. Birgit Hofer, Franz Wotawa, and Rui Abreu. "AI for the win: improving spectrum-based fault localization." In: *ACM SIGSOFT Software Engineering Notes 37*.6, pp. 1–8.
   Rui Abreu contributed the results of Barinel and Deputo and the text about them and Sfl. I contributed the results and the description of Sendys and performed the comparison of the approaches. Franz Wotawa motivated the paper. All authors equally contributed to the discussion section. The results of the evaluation are used in Chapter 4.
6. Birgit Hofer and Franz Wotawa. "How to combine slicing-hitting-set-computation with spectrum-based fault localization." In: *22nd International Workshop on Principles of Diagnosis*, pp. 114–121.
   I had the idea for this paper and implemented the prototype. Franz Wotawa helped in writing and proof-reading the paper. The Chapters 3 and 4 are based on this paper.
7. Simon Außerlechner, Sandra Fruhmann, Wolfgang Wieser, Birgit Hofer, Raphael Spörk, Clemens Mühlbacher, and Franz Wotawa. "The right choice matters! SMT solving substantially improves model-based debugging of spreadsheets." In: *13th International Conference on Quality Software*, in press.
   Simon Außerlechner had the idea for this paper. I wrote the introduction, the basic definitions (expect of the description of the MCSes and MCSes-U algorithms) and the part about the empirical evaluation. Simon Außerlechner, Sandra Fruhmann, Wolfgang Wieser, Raphael Spörk and Clemens Mühlbacher performed the empirical evaluation. Franz Wotawa helped in writing and proof-reading the paper. Chapter 11 is based on this paper.

A.2. Unpublished work

1. Simon Außerlechner, Birgit Hofer, Franz Wotawa, and Rui Abreu. "Mutation Supported Spreadsheet Fault Diagnosis." Submitted for publication.
   Franz Wotawa and Rui Abreu had the basic idea for this paper. Simon Außerlechner and I worked on the technical realization of the idea and on solving the problems of the infinite search space. Rui Abreu wrote the introduction, the related work and the conclusion. I wrote about the Csp and the mutation creation process. Simon Außerlechner and I wrote the Sections about the distinguishing test cases and the empirical evaluation. Franz Wotawa helped in writing and proof-reading the

paper. Chapter 12 is based on this paper.

2. Rui Abreu, Birgit Hofer, Alexandre Perez, André Riboira, and Franz Wotawa. “Using Constraints to Debug Spreadsheets.” Unpublished. The basic idea for this paper came from Rui Abreu, André Riboira, and Franz Wotawa and have already been published [ARW12]. I helped them to improve their approach by making a profound empirical evaluation and by extending the basic definitions of their work. Chapter 10 is based on this unpublished work.

Bibliography

[Abr+09a] Rui Abreu, Wolfgang Mayer, Markus Stumptner, and Arjan J. C. van Gemund. “Refining Spectrum-based Fault Localization Rankings.” In: *Proceedings of the 24th Annual ACM Symposium on Applied Computing (SAC’09)*. Honolulu, Hawaii, USA: ACM Press, Aug. 2009, pp. 409–414 (cit. on pp. 17, 18, 42).

[Abr+09b] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan J. C. van Gemund. “A practical evaluation of spectrum-based fault localization.” In: *Journal of Systems & Software (JSS)* 82.11 (2009), pp. 1780–1792. ISSN: 0164-1212. DOI: 10.1016/j.jss.2009.06.035 (cit. on pp. 16, 38, 58).

[Abr09] Rui Abreu. “Spectrum-based Fault Localization in Embedded Software.” PhD thesis. Delft University of Technology, Nov. 2009. ISBN: 978-90-79982-04-2 (cit. on p. 39).

[AE05] Robin Abraham and Martin Erwig. “Goal-Directed Debugging of Spreadsheets.” In: *Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*. VLHCC ’05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 37–44. ISBN: 0-7695-2443-5 (cit. on pp. 87, 134).

[AE06] Robin Abraham and Martin Erwig. “AutoTest: A Tool for Automatic Test Case Generation in Spreadsheets.” In: *Proceedings of the 2006 IEEE Symposium on Visual Languages and Human-Centric Computing*. VLHCC ’06. Brighton, UK, 2006, pp. 43–50 (cit. on p. 88).

[AE07a] Robin Abraham and Martin Erwig. “GoalDebug: A Spreadsheet Debugger for End Users.” In: *Proceedings of the 29th International Conference on Software Engineering*. ICSE ’07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 251–260. ISBN: 0-7695-2828-7. DOI: 10.1109/ICSE.2007.39 (cit. on pp. 87, 134).

[AE07b] Robin Abraham and Martin Erwig. “UCheck: A spreadsheet type checker for end users.” In: *Journal of Visual Languages and Computing* 18 (1 Feb. 2007), pp. 71–95. ISSN: 1045-926X. DOI: 10.1016/j.jvlc.2006.06.001 (cit. on p. 87).

[AE08] Robin Abraham and Martin Erwig. "Test-driven goal-directed debugging in spreadsheets." In: *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2008, Herrsching am Ammersee, Germany, 15-19 September 2008*. IEEE, 2008, pp. 131–138. DOI: 10.1109/VLHCC.2008.4639073 (cit. on pp. 87, 134).

[AE09] Robin Abraham and Martin Erwig. "Mutation operators for spreadsheets." In: *IEEE Transactions on Software Engineering (TSE)* 35 (2009), pp. 94–108 (cit. on pp. 87, 98).

[AG10] Rui Abreu and Arjan J. C. van Gemund. "Diagnosing multiple intermittent failures using maximum likelihood estimation." In: *Artificial Intelligence* 174 (18 Dec. 2010), pp. 1481–1497. ISSN: 0004-3702. DOI: 10.1016/j.artint.2010.09.003 (cit. on pp. 18, 42, 150).

[AM03] Yirsaw Ayalew and Roland Mittermeir. "Spreadsheet Debugging." In: *Bilding Better Business Spreadsheets - from the ad-hoc to the quality-engineered. Proceedings of EuSpRIG 2003, Dublin, Ireland, July 24th-25th 2003* (2003), pp. 67–79 (cit. on pp. 88, 101).

[Arc08] Andrea Arcuri. "On the automation of fixing software bugs." In: *ICSE Companion '08: Companion of the 30th international conference on Software engineering*. Leipzig, Germany: ACM, 2008, pp. 1003–1006. ISBN: 978-1-60558-079-1. DOI: 10.1145/1370175.1370223 (cit. on p. 17).

[ARW12] Rui Abreu, André Riboira, and Franz Wotawa. "Constraint-based Debugging of Spreadsheets." In: *Proceedings of the XV Iberoamerican Conference on Software Engineering, Buenos Aires, Argentina, April 24-27, 2012*. Ed. by Renata S. S. Guizzardi, Claudia Pons, and Alejandro Oliveros. 2012, pp. 1–14. URL: http://cibse.inf.puc-rio.br/CIBSEPapers/artigos/artigos_CIBSE12/paper_46.pdf (cit. on pp. 109, 117, 133, 155).

[Auß+] Simon Außerlechner, Birgit Hofer, Franz Wotawa, and Rui Abreu. *Mutation Supported Spreadsheet Fault Diagnosis*. Submitted for publication (cit. on p. 133).

[Auß+13] Simon Außerlechner, Sandra Fruhmann, Wolfgang Wieser, Birgit Hofer, Raphael Spörk, Clemens Mühlbacher, and Franz Wotawa. "The right choice matters! SMT solving substantially improves model-based debugging of spreadsheets." In: *13th International Conference on Quality Software*. In press. 2013 (cit. on pp. 97, 117).

[Aye+07] Nathaniel Ayewah, William Pugh, J. David Morgenthaler, John Penix, and YuQian Zhou. “Evaluating static analysis defect warnings on production software.” In: *Proceedings of the 2007 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. PASTE ’07. San Diego, California, USA, 2007, pp. 1–8 (cit. on p. 11).

[AZG06] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. “An Evaluation of Similarity Coefficients for Software Fault Localization.” In: *Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing*. PRDC ’06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 39–46. ISBN: 0-7695-2724-8. DOI: 10.1109/PRDC.2006.18 (cit. on pp. 3, 37, 38, 58, 103).

[AZG07] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. “On the Accuracy of Spectrum-based Fault Localization.” In: *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*. TAICPART-MUTATION ’07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 89–98. ISBN: 0-7695-2984-4. URL: http://dl.acm.org/citation.cfm?id=1308173.1308264 (cit. on pp. 101, 102).

[Bei90] Boris Beizer. *Software testing techniques (2nd ed.)* New York, NY, USA: Van Nostrand Reinhold Co., 1990. ISBN: 0-442-20672-0 (cit. on p. 23).

[BF85] F. Brglez and H. Fujiwara. “A neutral netlist of 10 combinational benchmark circuits and a target translator in Fortran.” In: *Proceedings of the IEEE International Symposium on Circuits and Systems*. June 1985, pp. 663–698 (cit. on p. 79).

[Bil89] C.W. Billings. *Grace Hopper: Navy admiral and computer pioneer*. Contemporary Women Series. Enslow Publishers, 1989. ISBN: 9780894901942. URL: http://books.google.at/books?id=6mEYbEeB0gsC (cit. on p. 3).

[BM94] Marc M. Brandis and Hanspeter Mössenböck. “Single-pass generation of static assignment form for structured languages.” In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16(6) (Nov. 1994), pp. 1684–1698 (cit. on p. 29).

[BP94] Gregory W. Bond and Bernie Pagurek. “Declarative Error Diagnosis as Consistency-Based Diagnosis.” In: *Symposium on Principles of Programming Languages*. 1994, p. 673 (cit. on p. 14).

[Bre08] Andrej Bregar. “Complexity Metrics for Spreadsheet Models.” In: *The Computing Research Repository (CoRR)* abs/0802.3895 (2008). URL: http://arxiv.org/abs/0802.3895 (cit. on p. 90).

[Bur+03] Margaret M. Burnett, Curtis R. Cook, Omkar Pendse, Gregg Rothermel, Jay Summet, and Chris S. Wallace. "End-User Software Engineering with Assertions in the Spreadsheet Paradigm." In: *Proceedings of the 25th International Conference on Software Engineering, May 3-10, 2003, Portland, Oregon, USA*. Ed. by Lori A. Clarke, Laurie Dillon, and Walter F. Tichy. ICSE '03. IEEE Computer Society, 2003, pp. 93–105. URL: `http://dl.acm.org/citation.cfm?id=776816` (cit. on p. 89).

[BZ11] Martin Burger and Andreas Zeller. "Minimizing Reproduction of Software Failures." In: *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ISSTA '11. Toronto, Ontario, Canada: ACM, July 2011, pp. 221–231. ISBN: 9781450305624. DOI: `10.1145/2001420.2001447` (cit. on p. 18).

[CC93] T. Y. Chen and Y. Y. Cheung. "Dynamic Program Dicing." In: *Proceedings of the Conference on Software Maintenance, ICSM 1993, Montréal, Quebec, Canada, September 1993*. Ed. by David N. Card. IEEE Computer Society, 1993, pp. 378–385. ISBN: 0-8186-4600-4 (cit. on p. 13).

[CDT91] Luca Console, Daniele Theseider Dupré, and Pietro Torasso. "On the Relationship Between Abduction and Deduction." In: *Journal of Logic and Computation* 1.5 (1991), pp. 661–690 (cit. on p. 29).

[CFD93] Luca Console, Gerhard Friedrich, and Daniele Theseider Dupré. "Model-Based Diagnosis Meets Error Diagnosis in Logic Programs." In: *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. Chambery, Aug. 1993, pp. 1494–1499 (cit. on p. 14).

[CKM05] Michael J. Coblenz, Andrew Jensen Ko, and Brad A. Myers. "Using Objects of Measurement to Detect Spreadsheet Errors." In: *2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2005), 21-24 September 2005, Dallas, TX, USA*. IEEE Computer Society, 2005, pp. 314–316. ISBN: 0-7695-2443-5. DOI: `10.1109/VLHCC.2005.67` (cit. on p. 88).

[CKR01] David Chadwick, Brian Knight, and Kamalasen Rajalingham. "Quality Control in Spreadsheets: A Visual Approach using Color Codings to Reduce Errors in Formulae." In: *Software Quality Journal* 9.2 (2001), pp. 133–143 (cit. on p. 4).

[Cun+12] Jácome Cunha, João Paulo Fernandes, Jorge Mendes, and João Saraiva. "MDSheet: A framework for model-driven spreadsheet engineering." In: *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*. Ed. by Martin Glinz, Gail C. Murphy, and Mauro Pezzè. IEEE, 2012, pp. 1395–

1398. ISBN: 978-1-4673-1067-3. DOI: 10.1109/ICSE.2012.6227239 (cit. on p. 89).

[CZ05] Holger Cleve and Andreas Zeller. "Locating Causes of Program Failures." In: *Proceedings of the 27th International Conference on Software Engineering*. ICSE '05. St. Louis, MO, USA: ACM Press, May 2005, pp. 342–351. ISBN: 1595939632 (cit. on p. 17).

[Dec03] Rina Dechter. *Constraint processing*. Elsevier Morgan Kaufmann, 2003, pp. I–XX, 1–481. ISBN: 978-1-55860-890-0. URL: http://www.elsevier.com/wps/find/bookdescription.agents/678024/description (cit. on p. 29).

[DER05] Hyunsook Do, Sebastian G. Elbaum, and Gregg Rothermel. "Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact." In: *Empirical Software Engineering: An International Journal* 10.4 (2005), pp. 405–435 (cit. on pp. 52, 58).

[DJ12] Nicholas DiGiuseppe and James A. Jones. "Concept-based failure clustering." In: *20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-20), SIGSOFT/FSE'12, Cary, NC, USA - November 11 - 16, 2012*. Ed. by Will Tracz, Martin P. Robillard, and Tevfik Bultan. ACM, 2012, p. 29. ISBN: 978-1-4503-1614-9, 978-1-4503-0443-6. DOI: 10.1145/2393596.2393629 (cit. on p. 17).

[Dow97] Mark Dowson. "The Ariane 5 software failure." In: *SIGSOFT Software Engineering Notes* 22.2 (Mar. 1997), p. 84. ISSN: 0163-5948. DOI: 10.1145/251880.251992 (cit. on p. 3).

[DPS96] Richard A. DeMillo, Hsin Pan, and Eugene H. Spafford. "Critical slicing for software fault localization." In: *Proceedings of the 1996 ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA '96. San Diego, California, USA: ACM, 1996, pp. 121–134. ISBN: 0-89791-787-1. DOI: 10.1145/229000.226310 (cit. on p. 13).

[Duc93] Mireille Ducassé. "A Pragmatic Survey of Automated Debugging." In: *Proceedings of Automated and Algorithmic Debugging, 1th International Workshop, AADEBUG'93, Linköping, Sweden, May 3-5, 1993*. Ed. by Peter Fritzson. Vol. 749. Lecture Notes in Computer Science. Springer, 1993, pp. 1–15. ISBN: 3-540-57417-4. DOI: 10.1007/BFb0019397 (cit. on p. 11).

[DW10] Vidroha Debroy and W. Eric Wong. "Using mutation to automatically suggest fixes for faulty programs." In: *Third International Conference on Software Testing, Verification and Validation (ICST 2010)*. Paris, France: IEEE, 2010 (cit. on p. 17).

[Fel+04] Alexander Felfernig, Gerhard Friedrich, Dietmar Jannach, and Markus Stumptner. "Consistency-based diagnosis of configuration knowledge bases." In: *Artificial Intelligence* 152.2 (2004), pp. 213–234. ISSN: 0004-3702. DOI: 10.1016/S0004-3702(03)00117-6 (cit. on p. 14).

[Fis+02] Marc Fisher, Mingming Cao, Gregg Rothermel, Curtis R. Cook, and Margaret M. Burnett. "Automated Test Case Generation for Spreadsheets." In: *Proceedings of the 24th International Conference on Software Engineering*. ICSE '02. ACM Press, 2002, pp. 141–151 (cit. on p. 88).

[FMS10] Gerhard Friedrich, Wolfgang Mayer, and Markus Stumptner. "Diagnosing Process Trajectories Under Partially Known Behavior." In: *Proceedings of the 19th European Conference on Artificial Intelligence (ECAI 2010), Lisbon, Portugal, August 16-20, 2010*, ed. by Helder Coelho, Rudi Studer, and Michael Wooldridge. Vol. 215. Frontiers in Artificial Intelligence and Applications. IOS Press, 2010, pp. 111–116. ISBN: 978-1-60750-605-8. URL: http://www.booksonline.iospress.nl/Content/View.aspx?piid=17724 (cit. on p. 14).

[FR05] Marc II Fisher and Gregg Rothermel. "The EUSES Spreadsheet Corpus: A Shared Resource for Supporting Experimentation with Spreadsheet Dependability Mechanisms." In: *1st Workshop on End-User Software Engineering*. 2005, pp. 47–51 (cit. on pp. 97, 101, 103, 104).

[FSW99] Gerhard Friedrich, Markus Stumptner, and Franz Wotawa. "Model-based diagnosis of hardware designs." In: *Artificial Intelligence* 111.1-2 (1999), pp. 3–39. ISSN: 0004-3702 (cit. on p. 14).

[Gal+93] Dennis F. Galletta, Dolphy Abraham, Mohamed El Louadi, William Lekse, Yannis A. Pollalis, and Jeffrey L. Sampler. "An empirical study of spreadsheet error-finding performance." In: *Accounting, Management and Information Technologies* 3.2 (1993), pp. 79–95 (cit. on p. 89).

[GBF99] Tibor Gyimóthy, Árpád Beszédes, and István Forgács. "An Efficient Relevant Slicing Method for Debugging." In: *Proceedings of the 7th European Software Engineering Conference (ESEC/FSE'99)*. Ed. by Oscar Nierstrasz and Michel Lemoine. Vol. 1687. Lecture Notes in Computer Science. Springer, 1999, pp. 303–321. ISBN: 3-540-66538-2. DOI: 10.1007/3-540-48166-4_19 (cit. on pp. 13, 24, 26, 35).

[GJM06] Ian P. Gent, Chris Jefferson, and Ian Miguel. “MINION: A Fast, Scalable, Constraint Solver.” In: *Proceedings of the 17th European Conference on Artificial Intelligence (ECAI 2006) August 29 – September 1, 2006, Riva del Garda, Italy* (2006), pp. 98–102. URL: `http://dl.acm.org/citation.cfm?id=1567016.1567043` (cit. on pp. 75, 113, 117).

[GMP12] Martin Glinz, Gail C. Murphy, and Mauro Pezzè, eds. *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*. IEEE, 2012. ISBN: 978-1-4673-1067-3.

[GSW89] Russell Greiner, Barbara A. Smith, and Ralph W. Wilkerson. “A Correction to the Algorithm in Reiter's Theory of Diagnosis.” In: *Artif. Intell.* 41.1 (1989), pp. 79–88. DOI: `10.1016/0004-3702(89)90079-9` (cit. on p. 33).

[Gup+05] Neelam Gupta, Haifeng He, Xiangyu Zhang, and Rajiv Gupta. “Locating Faulty Code Using Failure-Inducing Chops.” In: *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*. Ed. by David F. Redmiles, Thomas Ellman, and Andrea Zisman. ASE '05. New York, NY, USA: ACM, Nov. 2005, pp. 263–272. DOI: `10.1145/1101908.1101948` (cit. on pp. 18, 65, 68).

[Ham08] Clemens Hammacher. *Design and Implementation of an Efficient Dynamic Slicer for Java*. Bachelor's Thesis. Nov. 2008 (cit. on p. 52).

[Har+98] Mary Jean Harrold, Gregg Rothermel, Rui Wu, and Liu Yi. “An empirical investigation of program spectra.” In: *Proceedings of the 1998 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. PASTE '98. Montreal, Quebec, Canada: ACM, 1998, pp. 83–90. ISBN: 1-58113-055-4. DOI: `10.1145/277631.277647` (cit. on pp. 16, 37).

[Her+13] Felienne Hermans, Ben Sedee, Martin Pinzger, and Arie van Deursen. “Data clone detection and visualization in spreadsheets.” In: *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*. Ed. by David Notkin, Betty H. C. Cheng, and Klaus Pohl. IEEE / ACM, 2013, pp. 292–301. ISBN: 978-1-4673-3076-3. URL: `http://dl.acm.org/citation.cfm?id=2486827` (cit. on p. 90).

[Hof+13] Birgit Hofer, André Riboira, Franz Wotawa, Rui Abreu, and Elisabeth Getzner. “On the Empirical Evaluation of Fault Localization Techniques for Spreadsheets.” In: *Proceedings of 16th International Conference on Fundamental Approaches to Software Engineering, FASE 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-*

24, 2013. Ed. by Vittorio Cortellessa and Dániel Varró. Vol. 7793. Lecture Notes in Computer Science. Springer, 2013, pp. 68–82. ISBN: 978-3-642-37056-4. DOI: 10.1007/978-3-642-37057-1_6 (cit. on pp. 91, 97, 101).

[HPD10] Felienne Hermans, Martin Pinzger, and Arie van Deursen. "Automatically Extracting Class Diagrams from Spreadsheets." In: *Proceedings of the 24th European Conference on Object-Oriented Programming (ECOOP 2010), Maribor, Slovenia, June 21-25, 2010.* Ed. by Theo D'Hondt. Vol. 6183. Lecture Notes in Computer Science. Springer, 2010, pp. 52–75. ISBN: 978-3-642-14106-5. DOI: 10.1007/978-3-642-14107-2_4 (cit. on p. 90).

[HPD11] Felienne Hermans, Martin Pinzger, and Arie van Deursen. "Breviz: Visualizing Spreadsheets using Dataflow Diagrams." In: *The Computing Research Repository (CoRR)* abs/1111.6895 (2011). DOI: http://arxiv.org/abs/1111.6895 (cit. on p. 90).

[HPD12a] Felienne Hermans, Martin Pinzger, and Arie van Deursen. "Detecting and visualizing inter-worksheet smells in spreadsheets." In: *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*. Ed. by Martin Glinz, Gail C. Murphy, and Mauro Pezzè. IEEE, 2012, pp. 441–451. ISBN: 978-1-4673-1067-3. DOI: 10.1109/ICSE.2012.6227171 (cit. on p. 90).

[HPD12b] Felienne Hermans, Martin Pinzger, and Arie van Deursen. "Detecting code smells in spreadsheet formulas." In: *28th IEEE International Conference on Software Maintenance, ICSM 2012, Trento, Italy, September 23-28, 2012*. IEEE Computer Society, 2012, pp. 409–418. ISBN: 978-1-4673-2313-0 (cit. on p. 90).

[HPD12c] Felienne Hermans, Martin Pinzger, and Arie van Deursen. "Measuring Spreadsheet Formula Understandability." In: *The Computing Research Repository (CoRR)* abs/1209.3517 (2012) (cit. on p. 90).

[HRB88] Susan Horwitz, Thomas W. Reps, and David Binkley. "Interprocedural slicing using dependence graphs." In: Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation (PLDI '88) (1988), pp. 35–46. DOI: 10.1145/53990.53994 (cit. on p. 12).

[HS06] Harry Howe and Mark G. Simkin. "Factors Affecting the Ability to Detect Spreadsheet Errors." In: *Decision Sciences Journal of Innovative Education* 4.1 (2006), pp. 101–122 (cit. on p. 89).

[Hut+94] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. “Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria.” In: *Proceedings of the 16th international conference on Software engineering*. ICSE ’94. Sorrento, Italy: IEEE Computer Society Press, 1994, pp. 191–200. ISBN: 0-8186-5855-X. URL: http://dl.acm.org/citation.cfm?id=257734.257766 (cit. on p. 52).

[HW11] Birgit Hofer and Franz Wotawa. “How to combine slicing-hitting-set-computation with spectrum-based fault localization.” In: *Proceedings of the 22nd International Workshop on Principles of Diagnosis*. 2011, pp. 114–121 (cit. on p. 41).

[HW12a] Birgit Hofer and Franz Wotawa. “Combining Slicing and Constraint Solving for Better Debugging: The CONBAS Approach.” In: *Advances in Software Engineering, vol. 2012, Article ID 628571,18 pages* (2012), pp. 1–8. DOI: 10.1155/2012/628571 (cit. on pp. 19, 65).

[HW12b] Birgit Hofer and Franz Wotawa. “Reducing the Size of Dynamic Slicing with Constraint Solving.” In: *12th International Conference on Quality Software, Xi’an, Shaanxi, China, August 27-29, 2012*. Ed. by Antony Tang and Henry Muccini. IEEE, 2012, pp. 41–48. ISBN: 978-1-4673-2857-9. DOI: 10.1109/QSIC.2012.44 (cit. on p. 65).

[HW12c] Birgit Hofer and Franz Wotawa. “Spectrum Enhanced Dynamic Slicing for better Fault Localization.” In: *Proceedings of 20th European Conference on Artificial Intelligence (ECAI 2012). Including Prestigious Applications of Artificial Intelligence (PAIS-2012) System Demonstrations Track, Montpellier, France, August 27-31 , 2012*. Ed. by Luc De Raedt, Christian Bessière, Didier Dubois, Patrick Doherty, Paolo Frasconi, Fredrik Heintz, and Peter J. F. Lucas. Vol. 242. Frontiers in Artificial Intelligence and Applications. IOS Press, 2012, pp. 420–425. ISBN: 978-1-61499-097-0. DOI: 10.3233/978-1-61499-098-7-420 (cit. on pp. 19, 41, 101).

[HW12d] Birgit Hofer and Franz Wotawa. *Spectrum Enhanced Dynamic Slicing for Fault Localization*. Tech. rep. IST-DR-2012-01. Institute for Software Technology, Graz University of Technology, 2012 (cit. on p. 41).

[HWA12] Birgit Hofer, Franz Wotawa, and Rui Abreu. “AI for the win: improving spectrum-based fault localization.” In: *ACM SIGSOFT Software Engineering Notes* 37.6 (2012), pp. 1–8. DOI: 10.1145/2382756.2382784 (cit. on pp. 19, 41).

[JAG09] Tom Janssen, Rui Abreu, and Arjan J.C. van Gemund. "Zoltar: a spectrum-based fault localization tool." In: *Proceedings of the 2009 ESEC/FSE Workshop on Software Integration and Evolution and Runtime (SINTER '09)*. Amsterdam, The Netherlands: ACM, 2009, pp. 23–30. ISBN: 978-1-60558-681-6. DOI: 10.1145/1596495.1596502 (cit. on pp. 37, 41, 49).

[JE10] Dietmar Jannach and Ulrich Engler. "Toward model-based debugging of spreadsheet programs." In: *9th Joint Conference on Knowledge-Based Software Engineering (JCKBSE'10) August 25-27, 2010, Kaunas, Lithuania*. Kaunas, Lithuania, 2010, pp. 252–264 (cit. on pp. 88, 110, 117, 133).

[JGG08] Dennis Jeffrey, Neelam Gupta, and Rajiv Gupta. "Fault localization using value replacement." In: *Proceedings of the 2008 International Symposium on Software Testing and Analysis*. ISSTA '08. Seattle, WA, USA: ACM, 2008, pp. 167–178. ISBN: 978-1-60558-050-0. DOI: 10.1145/1390630.1390652 (cit. on p. 68).

[JH05] James A. Jones and Mary Jean Harrold. "Empirical evaluation of the tarantula automatic fault-localization technique." In: *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*. ASE '05. Long Beach, CA, USA: ACM, 2005, pp. 273–282. ISBN: 1-59593-993-4. DOI: 10.1145/1101908.1101949 (cit. on pp. 16, 58).

[JHB07] James A. Jones, Mary Jean Harrold, and James F. Bowring. "Debugging in Parallel." In: *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2007, London, UK, July 9-12, 2007*. Ed. by David S. Rosenblum and Sebastian G. Elbaum. ACM, 2007, pp. 16–26. ISBN: 978-1-59593-734-6. DOI: 10.1145/1273463.1273468 (cit. on p. 16).

[Kam95] Mariam Kamkar. "An overview and comparative classification of program slicing techniques." In: *Journal of Systems and Software* 31.3 (1995), pp. 197–214. DOI: 10.1016/0164-1212(94)00099-9 (cit. on p. 14).

[KL88] Bogdan Korel and Janusz Laski. "Dynamic program slicing." In: *Information Processing Letters* 29 (3 Oct. 1988), pp. 155–163. ISSN: 0020-0190. DOI: 10.1016/0020-0190(88)90054-3 (cit. on pp. 12, 24, 35).

[KMR92] Johan de Kleer, Alan K. Mackworth, and Raymond Reiter. "Characterizing Diagnoses and Systems." In: *Artificial Intelligence* 56.2-3 (1992), pp. 197–222. DOI: 10.1016/0004-3702(92)90027-U (cit. on p. 29).

[KR98] Bogdan Korel and Juergen Rilling. “Dynamic program slicing methods.” In: *Information & Software Technology* 40.11-12 (1998), pp. 647–659. DOI: 10.1016/S0950-5849(98)00089-5 (cit. on p. 14).

[Lib+05] Ben Liblit, Mayur Naik, Alice X. Zheng, Alexander Aiken, and Michael I. Jordan. “Scalable statistical bug isolation.” In: *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*. Ed. by Vivek Sarkar and Mary W. Hall. ACM, 2005, pp. 15–26. ISBN: 1-59593-056-6. DOI: 10.1145/1065010.1065014 (cit. on p. 16).

[Liu+06] C. Liu, L. Fei, X. Yan, J. Han, and S.P. Midkiff. “Statistical Debugging: A Hypothesis Testing-Based Approach.” In: *IEEE Transactions on Software Engineering (TSE)* 32.10 (2006), pp. 831–848. ISSN: 0098-5589. DOI: 10.1109/TSE.2006.105 (cit. on p. 16).

[LS08] Mark H. Liffiton and Karem A. Sakallah. “Algorithms for Computing Minimal Unsatisfiable Subsets of Constraints.” In: *Journal of Automated Reasoning (JAR)* 40.1 (Jan. 2008), pp. 1–33. ISSN: 0168-7433. DOI: 10.1007/s10817-007-9084-z (cit. on pp. 118, 119).

[LS09] Mark H. Liffiton and Karem A. Sakallah. “Generalizing Core-Guided Max-SAT.” In: *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing*. SAT ’09. Swansea, UK: Springer-Verlag, 2009, pp. 481–494. ISBN: 978-3-642-02776-5. DOI: 10.1007/978-3-642-02777-2_44 (cit. on pp. 118, 120, 121).

[LW87] J.R. Lyle and M.D. Weiser. “Automatic Program Bug Location by Program Slicing.” In: *Proceedings of 2nd International Conference on Computers and Applications, Peking, China*. June 1987, pp. 877–882 (cit. on p. 13).

[Mat+00] Cristinel Mateis, Markus Stumptner, Dominik Wieland, and Franz Wotawa. “Model-Based Debugging of Java Programs.” In: *Proceedings of the 4th International Workshop on Automated Debugging, AADEBUG 2000, Munich, Germany, August 28-30th, 2000*. 2000 (cit. on p. 15).

[May+08] Wolfgang Mayer, Rui Abreu, Markus Stumptner, and Arjan J. C. van Gemund. “Prioritising Model-Based Debugging Diagnostic Reports.” In: *Proceedings of the 19th International Workshop on Principles of Diagnosis*. Blue Mountains, Sydney, Australia, Sept. 2008 (cit. on pp. 39, 41).

[MB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. “Z3: An Efficient SMT Solver.” In: *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008*. Vol. 4963. Lecture Notes in Computer Science. Springer, 2008, pp. 337–340. ISBN: 978-3-540-78799-0. DOI: 10.1007/978-3-540-78800-3_24 (cit. on pp. 117, 139).

[MC02] Roland Mittermeir and Markus Clermont. “Finding High-Level Structures in Spreadsheet Programs.” In: *9th Working Conference on Reverse Engineering (WCRE 2002), 28 October - 1 November 2002, Richmond, VA, USA*. Ed. by Arie van Deursen and Elizabeth Burd. IEEE Computer Society, 2002, pp. 221–232. ISBN: 0-7695-1799-4. DOI: 1799/17990221abs.htm (cit. on p. 89).

[ME03] Madanlal Musuvathi and Dawson R. Engler. “Some Lessons from Using Static Analysis and Software Model Checking for Bug Finding.” In: *Electronic Notes in Theoretical Computer Science* 89.3 (2003), pp. 378–404. DOI: 10.1016/S1571-0661(05)80002-7 (cit. on p. 14).

[MOK06] Yu-Seung Ma, Jeff Offutt, and Yong Rae Kwon. “MuJava: a mutation system for java.” In: *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006*. Ed. by Leon J. Osterweil, H. Dieter Rombach, and Mary Lou Soffa. ACM, 2006, pp. 827–830. ISBN: 1-59593-375-1. DOI: 10.1145/1134425 (cit. on pp. 13, 134).

[MS07] Wolfgang Mayer and Markus Stumptner. “Model-Based Debugging – State of the Art And Future Challenges.” In: *Electronic Notes in Theoretical Computer Science* 174.4 (May 2007), pp. 61–82. ISSN: 1571-0661. DOI: 10.1016/j.entcs.2006.12.030 (cit. on p. 15).

[MS08] Wolfgang Mayer and Markus Stumptner. “Evaluating Models for Model-Based Debugging.” In: *Proceedings of the 23rd IEEE/ACM Int. Conference on Automated Software Engineering*. ASE ’08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 128–137. ISBN: 978-1-4244-2187-9. DOI: 10.1109/ASE.2008.23 (cit. on pp. 15, 36).

[MSW00] Cristinel Mateis, Markus Stumptner, and Franz Wotawa. “Locating Bugs in Java Programs - First Results of the Java Diagnosis Experiment Project.” In: *Proceedings of the 13th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems, IEA/AIE 2000, New Orleans, Louisiana,* *USA, June 19-22*. Ed. by Rasiah Loganantharaj and Günther Palm. Vol. 1821. Lecture Notes in Computer Science. Springer, 2000, pp. 174–183. ISBN: 3-540-67689-9. DOI: 10.1007/3-540-45049-1_21 (cit. on p. 15).

[NNW10] Mihai Nica, Simona Nica, and Franz Wotawa. "Using Distinguishing Tests to Reduce the Number of Fault Candidates." In: *Proceedings of the 21st International Workshop on the Principles of Diagnosis* (2010) (cit. on p. 134).

[NNW12] Mihai Nica, Simona A. Nica, and Franz Wotawa. "On the use of mutations and testing for debugging." In: *Software : practice & experience* (2012). DOI: 10.1002/spe.1142 (cit. on pp. 15, 29, 68, 83, 134, 136).

[NWW09] Mihai Nica, Jörg Weber, and Franz Wotawa. "On the use of Specification Knowledge in Program Debugging." In: *Proceedings of 20th International Workshop on Principles of Diagnosis, Schweden*. 2009, pp. 35–42 (cit. on p. 15).

[PA10] Raymond R. Panko and Salvatore Aurigemma. "Revising the Panko-Halverson taxonomy of spreadsheet errors." In: *Decision Support Systems* 49.2 (2010), pp. 235–244. ISSN: 0167-9236 (cit. on p. 89).

[Pan98] Raymond R. Panko. "What we know about spreadsheet errors." In: *Journal of End User Computing* 10.2 (1998), pp. 15–21. URL: http://panko.shidler.hawaii.edu/My%20Publications/Whatknow.htm (cit. on p. 89).

[PJ96] Raymond R. Panko and Richard P. Halverson Jr. "Spreadsheets on Trial: A Survey of Research on Spreadsheet Risks." In: *Proceedings of the 29th Annual Hawaii International Conference on System Sciences (HICSS-29), January 3-6, 1996, Maui, Hawaii*. 1996, pp. 326–335 (cit. on p. 89).

[PP12] Raymond R. Panko and Daniel N. Port. "End User Computing: The Dark Matter (and Dark Energy) of Corporate IT." In: *Proceedings of the 45th Hawaii International Conference on Systems Science (HICSS-45 2012), 4-7 January 2012, Grand Wailea, Maui, HI, USA*. IEEE Computer Society, 2012, pp. 4603–4612. ISBN: 978-0-7695-4525-7. DOI: 10.1109/HICSS.2012.244 (cit. on pp. 4, 87).

[RCK08] Kamalasen Rajalingham, David R. Chadwick, and Brian Knight. "Classification of Spreadsheet Errors." In: *The Computing Research Repository (CoRR)* abs/0805.4224 (2008). URL: http://arxiv.org/abs/0805.4224 (cit. on p. 89).

[Rei87] Raymond Reiter. "A Theory of Diagnosis from First Principles." In: *Artificial Intelligence* 32.1 (Apr. 1987), pp. 57–95. DOI: 10.1016/0004-3702(87)90062-2 (cit. on pp. 14, 28, 29, 32).

[Rot+00] Karen J. Rothermel, Curtis R. Cook, Margaret M. Burnett, Justin Schonfeld, T. R. G. Green, and Gregg Rothermel. "WYSIWYT testing in the spreadsheet paradigm: an empirical evaluation." In: *Proceedings of the 22nd international conference on Software engineering*. ICSE '00. Limerick, Ireland: ACM, 2000, pp. 230–239. ISBN: 1-58113-206-9. DOI: 10.1145/337180.337206 (cit. on p. 88).

[RR03] Manos Renieris and Steven P. Reiss. "Fault Localization With Nearest Neighbor Queries." In: *Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE'03)*. Ed. by John Grundy and John Penix. Montreal, Canada: IEEE Computer Society, June 2003, pp. 30–39 (cit. on p. 16).

[Rut+03] J. Ruthruff, E. Creswick, M. Burnett, C. Cook, S. Prabhakararao, M. Fisher II, and M. Main. "End-user software visualizations for fault localization." In: *Proceedings of the 2003 ACM symposium on Software visualization*. SoftVis '03. San Diego, California: ACM, 2003, pp. 123–132. ISBN: 1-58113-642-0. DOI: 10.1145/774833.774851 (cit. on pp. 88, 101, 102).

[Sev87] Rudolph E. Seviora. "Knowledge-Based Program Debugging Systems." In: *IEEE Software* 4.3 (May 1987), pp. 20–32. ISSN: 0740-7459 (print), 0740-7459 (electronic) (cit. on p. 11).

[SFB07] Manu Sridharan, Stephen J. Fink, and Rastislav Bodík. "Thin slicing." In: *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*. ACM, 2007, pp. 112–122. ISBN: 978-1-59593-633-2. DOI: 10.1145/1250734.1250748 (cit. on p. 13).

[Sha83] Ehud Shapiro. *Algorithmic Program Debugging*. Cambridge, Massachusetts: MIT Press, 1983 (cit. on p. 14).

[Shc+12] Kostyantyn Shchekotykhin, Gerhard Friedrich, Philipp Fleiss, and Patrick Rodler. "Interactive ontology debugging: Two query strategies for efficient fault localization." In: *Journal of Web Semantics* 12–13 (2012), pp. 88–103 (cit. on p. 14).

[SW99] Markus Stumptner and Franz Wotawa. "Debugging Functional Programs." In: *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI'99)*. Stockholm, Sweden, 1999, pp. 1074–1079 (cit. on p. 14).

[Tip95] Frank Tip. "A Survey of Program Slicing Techniques." In: *Journal of Programming Languages* 3.3 (Sept. 1995), pp. 121–189 (cit. on p. 14).

[WDC10] W. Eric Wong, Vidroha Debroy, and Byoungju Choi. "A family of code coverage-based heuristics for effective fault localization." In: *Journal of Systems and Software* 83.2 (2010), pp. 188–208. DOI: 10.1016/j.jss.2009.09.037 (cit. on pp. 16, 150).

[Wei+09] Westley Weimer, Thanh Vu Nguyen, Claire Le Goues, and Stephanie Forrest. "Automatically Finding Patches Using Genetic Programming." In: *ACM/IEEE International Conference on Software Engineering*. ICSE '09. 2009, pp. 512–521 (cit. on pp. 17, 134).

[Wei+10] Westley Weimer, Stephanie Forrest, Claire Le Goues, and ThanhVu Nguyen. "Automatic program repair with evolutionary computation." In: *Communications of the ACM* 53.5 (2010), pp. 109–116. ISSN: 0001-0782. DOI: 10.1145/1735223.1735249 (cit. on p. 17).

[Wei82] Mark Weiser. "Programmers use slices when debugging." In: *Communications of the ACM* 25.7 (1982), pp. 446–452. ISSN: 0001-0782. DOI: 10.1145/358557.358577 (cit. on pp. 12, 24, 35).

[Wei84] Mark Weiser. "Program Slicing." In: *IEEE Transactions on Software Engineering* 10.4 (1984), pp. 352–357. DOI: 10.1109/TSE.1984.5010248 (cit. on p. 12).

[Wen+11] Wanzhi Wen, Bixin Li, Xiaobing Sun, and Jiakai Li. "Program slicing spectrum-based software fault localization." In: *Proceedings of the 23rd International Conference on Software Engineering & Knowledge Engineering (SEKE'2011), Eden Roc Renaissance, Miami Beach, USA, July 7-9, 2011*. Knowledge Systems Institute Graduate School, 2011, pp. 213–218. ISBN: 1-891706-29-2 (cit. on p. 18).

[Wen12] Wanzhi Wen. "Software fault localization based on program slicing spectrum." In: *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*. Ed. by Martin Glinz, Gail C. Murphy, and Mauro Pezzè. IEEE, 2012, pp. 1511–1514. ISBN: 978-1-4673-1067-3. DOI: 10.1109/ICSE.2012.6227049 (cit. on p. 18).

[WN08] Franz Wotawa and Mihai Nica. "On the Compilation of Programs into their equivalent Constraint Representation." In: *Informatica (Slovenia)* 32.4 (2008), pp. 359–371 (cit. on p. 15).

[WNM12] Franz Wotawa, Mihai Nica, and Iulia Moraru. "Automated debugging based on a constraint model of the program and a test case." In: *The Journal of Logic and Algebraic Programming* 81.4 (2012), pp. 390–407 (cit. on pp. 15, 29, 32, 68, 83).

[Won+08] Eric Wong, Tingting Wei, Yu Qi, and Lei Zhao. "A Crosstab-based Statistical Method for Effective Fault Localization." In: *Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*. ICST '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 42–51. ISBN: 978-0-7695-3127-4. DOI: 10.1109/ICST.2008.65 (cit. on p. 16).

[Wot+09] Franz Wotawa, Jörg Weber, Mihai Nica, and Rafael Ceballos. "On the Complexity of Program Debugging Using Constraints for Modeling the Program's Syntax and Semantics." In: *Proceedings of the 13th Conference of the Spanish Association for Artificial Intelligence, CAEPIA 2009, Seville, Spain, November 9-13, 2009*. 2009, pp. 22–31 (cit. on pp. 15, 72, 109, 110, 145).

[Wot02] Franz Wotawa. "On the Relationship between Model-Based Debugging and Program Slicing." In: *Artificial Intelligence* 135 (1-2 Feb. 2002), pp. 125–143. ISSN: 0004-3702. DOI: 10.1016/S0004-3702(01)00161-8 (cit. on pp. 15, 32, 33).

[Wot10] Franz Wotawa. "Fault Localization Based on Dynamic Slicing and Hitting-Set Computation." In: *Proceedings of the 10th International Conference on Quality Software*. QSIC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 161–170. ISBN: 978-0-7695-4131-0. DOI: 10.1109/QSIC.2010.51 (cit. on pp. 34, 35, 41, 45).

[Wot11] Franz Wotawa. "On the Use of Constraints in Dynamic Slicing for Program Debugging." In: *4th International IEEE Conference on Software Testing, Verification and Validation, ICST 2012, Berlin, Germany, 21-25 March, 2011, Workshop Proceedings*. IEEE Computer Society, 2011, pp. 624–633. DOI: 10.1109/ICSTW.2011.61. URL: http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=5954009 (cit. on p. 68).

[Xu+11] Jian Xu, W. K. Chan, Zhenyu Zhang, T. H. Tse, and Shanping Li. "A Dynamic Fault Localization Technique with Noise Reduction for Java Programs." In: *Proceedings of the 11th International Conference on Quality Software*. QSIC 2011. 2011, pp. 11–20 (cit. on pp. 16, 42).

[Zel02] Andreas Zeller. "Isolating cause-effect chains from computer programs." In: *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering 2002, Charleston, South Carolina, USA, November 18 - 22, 2002*. 2002, pp. 1–10. DOI: 10.1145/587051.587053 (cit. on p. 17).

[ZGG06] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. "Pruning dynamic slices with confidence." In: *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Im-*

plementation, Ottawa, Ontario, Canada, June 11-14, 2006. Ed. by Michael I. Schwartzbach and Thomas Ball. ACM, 2006, pp. 169–180. ISBN: 1-59593-320-4 (cit. on pp. 13, 68).

[ZGG07] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. "A study of effectiveness of dynamic slicing in locating real faults." In: *Empirical Software Engineering* 12.2 (Apr. 2007), pp. 143–160. ISSN: 1382-3256. DOI: 10.1007/s10664-006-9007-3 (cit. on p. 13).

[ZGZ04] Xiangyu Zhang, Rajiv Gupta, and Youtao Zhang. "Efficient Forward Computation of Dynamic Slices Using Reduced Ordered Binary Decision Diagrams." In: *Proceedings of the 26th International Conference on Software Engineering*. ICSE '04. Los Alamitos, CA, USA: IEEE Computer Society, 2004, pp. 502–511. DOI: 10.1109/ICSE.2004.1317472 (cit. on p. 13).

[ZH02] Andreas Zeller and Ralf Hildebrandt. "Simplifying and Isolating Failure-Inducing Input." In: *IEEE Transactions on Software Engineering* 28.2 (2002), pp. 183–200. DOI: 10.1109/32.988498 (cit. on p. 17).

[Zha+07] Xiangyu Zhang, Sriraman Tallam, Neelam Gupta, and Rajiv Gupta. "Towards locating execution omission errors." In: *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation* 42.6 (2007), pp. 415–424. ISSN: 0362-1340. DOI: 10.1145/1273442.1250782 (cit. on p. 13).

[Zoe+07] Peter Zoeteweij, Rui Abreu, Rob Golsteijn, and Arjan J.C. van Gemund. "Diagnosis of Embedded Software Using Program Spectra." In: *IEEE International Conference on the Engineering of Computer-Based Systems* (2007), pp. 213–220. DOI: 10.1109/ECBS.2007.31 (cit. on pp. 16, 58).

[ZTG06] Xiangyu Zhang, Sriraman Tallam, and Rajiv Gupta. "Dynamic slicing long running programs through execution fast forwarding." In: *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. SIGSOFT '06/FSE-14. Portland, Oregon, USA: ACM, 2006, pp. 81–91. ISBN: 1-59593-468-5. DOI: 10.1145/1181775.1181786 (cit. on p. 13).