Dissertation

On the Use of Constraints in Automated Program Debugging - From Foundations to Empirical Results.

Mihai Nica

Graz, 2010

Institute for Software Technology
Graz University of Technology

Supervisor/First reviewer: Univ.-Prof. Dipl.-Ing. Dr. techn. Franz Wotawa
Second reviewer: O.Univ.-Prof. Dipl.-Ing. Dr. Gerhard Friedrich

”Do not despise the small things, a candle will always can something that the sun will never can: lighten up the night”

-Octavian Paler

Abstract (English)

Software verification is one of the most tedious and time consuming tasks in the software development life-cycle. By verification it is often misunderstood only the process of software testing. It would worth nothing to the software developer if, after testing for correctness, the software would fail without any hint on what went wrong. That is, whenever a software fails a test case, a second phase of the verification process starts: ***debugging***, i.e., identifying the possible causes for the program's failure, and ***repair*** phase, i.e., correcting the bug.

The history of debugging began in 1947, when the operators of the Harvard Mark II computer identify the first fault in a software. The term BUG was for the first time used to describe a deviation from the expected behavior. Nowadays debugging remains a crucial activity in the software life cycle, being one of the most expensive tasks in the development of software [49].

This work focuses on the topic of software debugging provided the existence of at least one failing test case in a predefined test suite. The problem of software debugging, i.e., fault localization, in case of a detected failure is a time consuming and intricate task. The automation or at least partial automation of debugging is therefore highly desired. The two major drawbacks with respect to automation of debugging are ***computation time***, and ***suggesting a repair solution***. Another important challenge is prediction of the debugging complexity for a given faulty program.

The model-based approach we present here relies on a constraint representation of a program that is equivalent to the original program in terms of the input-output behavior under some reasonable assumptions. By using constraints for representing programs and subsequently test cases we are able to state the debugging problem as a constraint satisfaction problem that can be effectively solved using a modern constraint solver. We further extend this work by integrating loop invariants and showing how that can further improve our constraint based approach. Another extension of our work is the usage of mutation and distinguishing test cases for the purpose of debugging but also for repair. Last,

using the constraint based representation, we are able to predict the debugging complexity by using the structural property of the hypergraph associated to the constraint representation of the debugged program. The resulted debugger can be used for small embedded software systems (software for robots), but also in debugging of complex high level imperative programming languages.

Abstract (German)

Software-Verifikation ist eine der anspruchsvollsten und zeitaufwändigsten Aufgaben im Lebenszyklus der Softwareentwicklung. Der Begriff der Verifikation wird oft falsch interpretiert und nur mit dem Prozess des Software-Testens an sich in Verbindung gebracht. Wenn beim Testen einer Software auf Korrektheit ein Fehler auftritt, wäre es für den Software-Entwickler nur von geringem Nutzen, wenn es keinen weiteren Hinweis auf die Art des Fehlers gäbe. Aus diesem Grund wird, immer wenn eine Software bei einem Testfall einen Fehler liefert, eine zweite Phase des Verifikations-Prozesses gestartet: ***Debugging***, d.h. identifizieren der Möglichen Ursachen für das Auftreten des Fehlers im Programm, gefolgt von einer ***Reparatur***-Phase, d.h. das Beheben des Fehlers.

Die Geschichte von Debugging begann 1947, als die Bediener der Harvard Mark II Rechner den ersten Fehler in einer Software identifizierten. Der Begriff BUG wurde anfänglich dazu benutzt, um die Abweichung von einem erwarteten Verhalten zu beschreiben. Debugging ist heutzutage immer noch eine essentielle Tätigkeit im Lebenszyklus einer Software, und ist eine der kostenintensivsten Aufgaben bei der Softwareentwicklung [49].

Der Fokus dieser Arbeit liegt auf dem Debuggen von Software, unter der Annahme dass innerhalb einer Testreihe mindestens ein fehlerhafter Testfall existiert. Das Problem von Software-Debugging, d.h. der Fehlerlokalisierung, besteht darin, dass es eine sehr zeitaufwändige und komplizierte Tätigkeit ist. Daher ist es wünschenswert eine Automatisierung oder zumindest eine Teil-Automatisierung von Debugging zu erreichen. Die zwei Hauptnachteile von Debugging im Bezug auf Automatisierung sind die ***Berechnungszeit*** und das ***Vorschlagen einer Problemlösung***. Eine andere wichtige Herausforderung stellt die Vorhersage der Debugging-Komplexität bei einem vorliegenden fehlerhaften Programm dar.

Der modellbasierte Ansatz den wir hier vorstellen beruht auf der eingeschränkten Darstellung eines Programms, welche unter realistischen Annahmen zu einem äquivalenten Eingangs-Ausgansverhalten

führt wie beim originalen Programm. Durch die Verwendung einer eingeschränkten Darstellung des Programms und der nachfolgenden Testfälle, sind wir in der Lage, das Debugging-Problem als Bedingungserfüllungsproblem darzustellen. Dieses Problem kann mit modernen Constraint-Solvern effizient gelöst werden. Darüberhinaus erweitern wir diese Arbeit noch durch Integration von Schleifeninvarianten und zeigen, dass dies unseren einschränkungsbasierenden Ansatz noch weiter verbessern kann. Eine andere Erweiterung unserer Arbeit ist die Anwendung von Mutationen und das Unterscheiden von Testfällen zum Debuggen und Reparieren von Programmen. Abschließend lässt sich sagen, dass wir mit Hilfe der einschränkungsbasierenden Darstellung und der Verwendung der strukturellen Eigenschaften des Hypographen, welcher der eingeschränkten Darstellung des zu debuggenden Programms zugehört, in der Lage sind, die Debugging-Komplexität vorherzusagen.

Der in dieser Arbeit entstandene Debugger kann nicht nur für kleine eingebettete Softwaresysteme (Software für Roboter) verwendet werden, sondern auch zum Debuggen komplexer Programme in imperativen Hochsprachen.

Acknowledgments

I thought thoroughly about how I should write my Acknowledgment section, it usually starts with *I would like to thank to....* Well, I intend to do so, but first I will tell a short story, which is very dear to me.

There was once a man who tried to change the world. He spent the best years of his life to do so. But after a time, he understood that he could not do it. He tried to change his country, and again after some years he had to give up. The same happened with his city, neighborhood and neighbors. Nothing. He was already old when his attention turned to his family. He tried to change them at least. Again, he failed. In the end he finally understood: If he would have only changed himself first and then try to change the others, perhaps, he would have succeeded... This is what I want to do in my life: before asking others to change, first change myself.

The last three years meant for me a permanent evolution, i.e., change by learning. This would have not been possible without the help of all the people that were around me and which are ”responsible” for the person I am today. So...

First, ***I would like to thank*** God for blessing me with all the people in my life: the good and not so good ones (without them I would have been depraved of life's most valuable lessons, given to me so far).

I would like to thank my ”Doktorvater” professor Franz Wotawa, for all his help, for his confidence in me, for his patience and honesty, for all that he taught me: about science and life, and especially for all those nice, fruitful and long conversion which we had. I would also like to thank the members of my defense committee, Prof. Gerhard Friedrich and Prof. Frank Kappe, for taking the time to review my thesis and to supervise my exam. I would also like to thank all my colleagues and students, for all our work together.

I would like to thank my best friend Valentin, because he knows me like ”the back of his pocket”, for listening to me when I was down, for always finding the right thing to say, for provoking me to be fair, when I was not... I would also like to thank Thomas, for all his help (and boy it was a lot), for our ”Biertermine” :), and because since I came in Austria he always was a true friend to me. I would also like to thank my friend and former colleague Willibald Krenn, i.e., Willi ;), for his help and for our, always pleasant, philosophical conversions. I would like to thank my colleague Jörg Weber for his critical eye when reading my papers, and for teaching me a thing or two about writing papers and formalisms.

I would like to thank my parents, Marioara and Iosif, for supporting me from the first moment of my life by consistently offering me all the love and confidence in the world, and for teaching me how to be strong and honorable but also kind and patient (I still have to work on that one though :)), to my grandmother, Maria, who loved me above all and taught me my religion and showed me the beautifulness and richness of the Romanian traditions, hence making me part of the spiritual heritage of my people. I would like to thank my wife Iulia, whom I dearly love, for her tenderness, confidence, support with MINION ;) and for making me smile :-) and feel good about myself. I would like to thank my sister, Simona, for her love and strong support, for she was always there for me, brave and ready to sacrifice a lot, just to help me. I would also like to thank Aniela for all her love, support, kindness and generosity, and for being like a grandmother to me.

Finally but surely not the least, I would like to thank all of my colleagues from the Institute of Software Technology, all of my collaborators and all of my friends who, due to space reasons :), where not mentioned here. Without all of you guys, I would have not become who I am today...

Oh, I almost forgot about it.... another special thanks goes to the ***Austrian Science Fund (FWF)***, grant P20199-N15, for founding my research.

Mihai Nica.
-Graz 2010.

In memory of my beloved
grandmother,
Maria

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, _____ _____

Place, Date Signature

Eidesstattliche Erklärung

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 entnommene Stellen als solche kenntlich gemacht habe.

Graz, am _____ _____

Ort, Datum Unterschrift

Contents

List of Figures

List of Tables

Chapter 1

Forward

"*Coding without a debugger makes you feel like a blind man in a dark room looking for a black cat that isn't there*" - [9].

1.1. Motivation

Through the history of software development, software bugs are held responsible for many financial and sometimes even human losses. For example, in October 2005, due to a bug in the embedded code inducing a problem to the warning light system and fuel indicator (the engine stalled while driving), Toyota is forced to call back more than 160.000 Prius hybrid vehicles. A software patch manages to solve this non-critical problem. Another well-known software bug is the Mars Rover robot which in 2004 freezes during a probing mission. The cause originated in opening too many files in the flash memory. Due to an error in an algorithm converting a 64-bit floating-point number into a 16-bit signed integer, on 4th of June 1994 Ariane 5 explodes 40 seconds after launching. Due to an error in the engine control computer, in 1994, a British Royal Air Force helicopter crashes killing 29 men. In 1991, due to a proximity error (software), a Patriot missile kills 29 people.

Ideally intelligent systems should provide self-reasoning and reflection capabilities in order to react on internal faults as well as on unexpected interaction with their environment. Reflection capabilities are highly recommended for systems with strong robustness constraints, like space exploration probes or even mobile robots. A scenario, for example, is a robot where the software fails because of a bug.

In this situation a robot should recover and ideally repair itself. Note that even exhaustive testing does not prevent a program from containing bugs which might cause an unexpected behavior in certain situations.

Making systems self-aware and giving them those self-healing capabilities increase their autonomy, which is especially important for software critical systems, e.g., space exploration or peace-makers, where the system cannot be directly controlled.

There exist thousands of examples where software bugs are the reason for important losses. Starting from the harmless smart phones software, e.g., email clients, navigation software, and ending with safety critical systems like airplane or life support medical devices, bugs are a present problem which has to be addressed and overcome.

Another issue that contributes to the complexity of finding bugs is the software development model. Nowadays it is very popular for a software company to divide its product into modules which are then outsourced. Very often when these modules are integrated the software is error prone and the need of debugging arises. In this situation an automated debugging tool would spare the developer the tedious and complicated work of having to understand the whole source code.

Hence, whether we want it or not, debugging must be integrated in the software life-cycle. There are two ways for performing debugging:

1. The hard way: Manual debugging which implies either
   - the usage of the classical ”print-statements” throughout the whole program or
   - the usage of a symbolic debugger. Symbolic debuggers are part of almost all major integrated development environments, e.g., Eclipse, Microsoft Visual Studio, Delphi.
2. The smart way: The usage of an automated debugger, like [85, 73, 71]

This thesis contributes to the research of automated debugging by providing a constraint based method for fault localization in programs. The methodology requires the availability of the source code and test cases. It compiles the program into their equivalent constraint representation and uses a failure-revealing test case to compute diagnosis candidates. We further improve our results with respect to the cardinality of the fault candidates set, by integrating loop invariants. Additionally we extend our work by making use of mutation and distinguishing test-cases techniques to discriminate between possible explanations for an occurred fault.

The work presented here was conducted within the MoDReMAS (Model-based Diagnosis and Reconfiguration in Mobile Autonomous Systems) project at Graz University of Technology, Institute for Software Technology. My task was to develop a debugging engine that smoothen up or even allows software systems to perform automated debugging and, under special circumstances, self-repair.

The most important feature of a debugging engine, i.e., debugger, is given by the ratio between computation time and the quality of the debugging results. The result outputted by a debugger running against a faulty program, is called the fault candidates set. The smaller this set is, the higher the quality of the debugger. However, automated debugging requires small computation time, even if, the resulting fault candidates set is sometimes larger.

To undergo this challenge, our work relies on the powerful mechanism of constraint solving. In order to compute the fault candidates we follow the model-based diagnosis approach, [92] but do not rely on logical models, instead we use constraints for representing programs. The obtained constraint representation can be directly used for computing diagnosis, e.g., by using specialized diagnosis algorithms like the one described in [38, 103, 104]. By implying constraint representation we manage to compute the fault candidates set within excellent times, outperforming by far other debugging techniques. Furthermore, the cardinality of the fault candidates set is similar to the one computed with other debugging techniques.

Additionally we study the relation between the structural properties of the debugged program and its debugging complexity.

The work presented here focuses exclusively on debugging at method level. The same technique(s) can be used to debug small software systems (up to 1300 lines of code) which typically run on autonomous systems. Extending this work for larger programs can be done by integration of pre- and postconditions at method levels.

1.2. Problem Statement

Although, reflection and debugging capabilities are a desired functionality of a system they provoke additional computational complexity which can hardly be handled by the system directly because of the lack of computational power. Note that model-based diagnosis is NP complete. Hence, a distributed architecture would be required which separates the running control program from the debugging capabilities, i.e., from the debugger. In our proposed framework the debugging module, i.e.,

the debugger, takes the source code of the original software system and at least one error revealing test-case, and uses them to localize and repair the fault. The changed source code is compiled and transferred back to the original software system.

Our **problem statement** is summarized as follows: Given a software system and at least one error revealing test case, i.e., where the program's output contradicts the predicted output, we must identify those components, i.e., statements, that individually or together with other components, explain for the failure of the program. The quality requirements are:

- The faulty statement(s) must be included in the fault candidates set.
- The cardinality of the fault candidates set must be the smallest possible.
- The computation time should allow self reasoning of the software system, in our case we require:
  - For small systems (circa 100-200 LOC) $\leq$ 5s and
  - For larger systems (circa 800-1000 LOC) $\leq$ 20s.

1.3. Contribution

The model based diagnosis approach presented in [92] can be easily adapted to be used for debugging of programs. Further, by using the constraint representation and a state-of-the-art constraint solver we manage to obtain considerable gains with respect to computation time. The integration of specification knowledge and the usage of distinguishing test cases are further used to reduce the number of fault candidates. Additionally we make use of the constraint representation to study the debugging complexity of the analyzed programs.

The following is a selection of journals, papers and workshops publications that contributed to the creation of the present thesis.

- *Converting Programs into Constraint Satisfaction Problems* [121];
- *From constraint representations of sequential code and program annotations to their use in debugging* [86];
- *On the Compilation of Programs into their equivalent Constraint Representation* [122];

- *How to debug sequential code by means of constraint representation* [84];
- *On the complexity of program debugging using constraints for modeling the programs syntax and semantics* [126];
- *On the use of Specification Knowledge in Program Debugging* [85];
- *Representing Program Debugging as Constraint Satisfaction Problems* [78];
- *Generating Distinguishing Tests using the MINION Constraint Solver* [123];
- *Does testing help to reduce the number of potentially faulty statement in debugging?* [68];
- *Automated debugging based on a constraint model of the program and a test case* [125] ;

The thesis is organized as follows. In Chapter 2 we present an overview of the available state-of-the-art techniques for debugging programs. Additionally we introduce here the terminology used throughout this thesis. Chapter 3 presents our constraint-based framework for solving the debugging problem. The most important definitions used throughout this thesis are to be found here (see Section 3.1). In Chapter 4 we present the two extensions of our debugging engine: integration of *specification knowledge* and the usage of *mutation and distinguishing test cases*, and study their impact on the debugging results. In Chapter 5 we analyze the correspondence between the structure of the program's constraint representation given as a hypergraph and the complexity of finding a solution. Finally, in Chapter 6 I conclude this work with a personal view over my work within this project and present some open issues which should be addressed in a future research.

Chapter 2

Related Approaches

"*Testing proves a programmer's failure. Debugging is the programmer's vindication.*" – Boris Beizer.

In the past decades there was an increased interest for software debugging. As a result, a number of debugging techniques appeared in the pursuit of providing a good solution to the debugging problem. In this chapter we present some of the latest state of the art debugging methodologies, showing both the weak and strong spots of each particular approach.

This chapter is divided as follows. In Section 2.1 we present and explain the unified terminology used in testing and debugging. In Section 2.2 we present the preliminaries for analyzing the debugging techniques. Section 2.3 deals with presenting the model based approach, whereas Section 2.4 presents the slicing approaches to debugging, e.g., dynamic, static slicing. In Section 2.5 we present the spectrum based techniques, whereas Section 2.6 refers to the other debugging approaches. We then conclude in Section 2.7.

2.1. Terminology

Software validation refers to the process of verifying if the software specifications meet the actual stakeholder requirements.

Software verification is the process of proving the correctness of software with respect to the provided set of specifications. Without some form of specification, it is not possible to perform verification. In [22], for example, the authors present an approach which tests the constraint representation of a program against its negated specification. They convert both the negate specifications and the program to a constraint system. They use an SMT solver to test if a solution to this constraints system exists. If yes, then they obtain an inconsistency. Generally, verification is based on model theoretic approaches [37] or axiomatic theoretic approaches [87]. More recent approaches include [79] (from Microsoft) or [97].

A fault is the trigger of a ***software failure***, i.e., the fault is the cause and the failure is the effect. The existence of a fault is revealed, i.e., detected, only by the existence of a failing test case which contradicts the program's behavior. If there is no available error revealing test case, the existence of a fault cannot be exposed. We can have faults in a program which do not trigger any failures. In this situation the system is said to be tolerant to this type of errors.

Testing on the other hand is only then "successful", when it reveals errors in an existent program. Specification knowledge is not always necessary, e.g., monkey testing, but desired. However the existence of an oracle which indicates under certain circumstances the correct behavior of a program is a necessary condition. The most common classification of testing procedure is:

- *black box testing*, the source code is not available to the tester. The testing process is guided only by the software's specifications.
- *gray box testing*, information about the structure of the program are known but no information about the actual source code, e.g., pre- and postconditions for a method.
- *white box testing*, also called glass-box testing, has the property that the tester has access to the complete source code of the program.

The reader must understand that on a theoretical level, testing, contrary to verification, cannot prove that a program is correct. It can only try to identify the existence of faults (by attempting to trigger failures), but not guarantee their absence. For more information about testing we refer the interested reader to [88].

Debugging is the process of localizing one or more faults in case of a detected failure. This is a time consuming and intricate task and sometimes manually hard to perform. For example, in software maintenance, when the code owner is not the same with the one performing maintenance, debugging is almost impossible to be done manually. The automation or at least partial automation of debugging

is therefore highly desired. To perform debugging the existence of an error revealing test case is mandatory.

Debugging of software is not software verification! This statement is, sadly, not for everybody a commonly "accepted truth". Throughout our work it happened, even by "heavy" conferences, e.g., IJCAI, that the reviewer often confused verification with debugging. The main difference is that verification is the process of proving correctness of a software (which can or cannot be correct) with respect to a given set of specifications, whereas debugging deals with identifying a failure in a "proven to be wrong" program.

At the moment, the debugging techniques can be classified into two main categories:

1. Based on the static analysis of the program, i.e., static debugging, which relies on the statically representation of the program and on the test case;
2. Based on the dynamic analysis of the program, i.e., dynamic debugging, which relies on the test case and the program's trace (computed from running the test case).

Correction is the second phase of program debugging and deals with suggesting repairs for the identified faults. In the last years there was a growing interest for automating this last step of debugging. Techniques like genetic debugging [112], could be used to support the process of repair. However, as we show in Chapter 4, this can be computational expensive and inaccurate. In Chapter 4 we also suggest an extension [68], relying on testing and mutation, that is both computational feasible and accurate with respect to repair suggestions.

Program debugging, i.e., the detection, localization, and correction of programs, is generally considered a hard problem especially after program deployment, but of huge practical value. Support for program debugging helps to keep direct and indirect costs of software development low. Faults that are corrected early in the development process cause less costs than faults revealed after shipping the software to the customers. Therefore, most of the research activities since the beginnings of software engineering have focused on verification and validation in order to ensure program correctness. Only a little research effort has been spent in developing tools for debugging. Debugging is sometimes seen as a consequence of poor testing [13]. The reality is however that testing and debugging are the two sides of one coin. In the testing phase, if a test fails, debugging starts, and vice versa if no available failing test case exists, one cannot start debugging. Hence debugging is the natural extension to testing, whereas testing is the *necessary but not sufficient* precondition for debugging.

In this chapter, we discuss some of the most recent approaches for debugging namely spectrum-

based, slicing-based, genetic and model-based debugging. Our work focuses on the latter and will be detailed in Chapter 3. Moreover, we briefly compare the four approaches and suggest a combination of them in order to improve the results and the overall necessary running time.

2.2. Introduction

In the context of this thesis, program debugging is defined as the activity carried out by humans or a program itself to localize a root cause in the source code of a program, responsible for an observed behavior deviating from the expected one. Obviously after finding the root cause we are also interested in making the corrective changes, but this part of debugging as a whole is not in the focus of this thesis. The given definition of debugging is a very general one. Until now we have not introduced any restriction regarding how to observe a deviation. For example, such a deviation might come from user demands that are not implemented in the deployed program. Such a root-cause requires adding functionality to the program and has to do with re-design. Another reason for inconsistencies is that the program does not pass all test cases. As a consequence, verification reveals a faulty behavior and the cause usually has to be tracked down to parts of the source code responsible for the misbehavior. Note that a program might fail passing a test case because the program computes a wrong value for a variable or raises an exception, e.g., division-by-zero exception.

The approaches presented here rely on *some restrictions*. It is assumed that the source code of the program as well as the test suite comprising at least one failing test case is given. A further restriction of debugging is: the situations where the original program computes a wrong value for at least one program variable (error identified by wrong output). We assume that the program to be debugged is syntactically correct, does not comprise any type of errors and infinite loops, and that the corrected program is a close variant of the original one.

In order to point out the technical differences between various debugging approaches we use the small program fragment depicted in Figure 2.1. This fragment computes the minimum and maximum of a collection of integers stored in an array, as well as the sum of the minimum and maximum under the precondition that the array comprises at least one element. Otherwise, an *Out of Bounds* exception would be raised when accessing the first element of the array in Line 2. Note, that changing this program in order to avoid the exception is simple, but increases the program size, which is less appropriate for explanation purposes. Moreover, due to the same reason and our assumptions, we exclude all definitions and type information from the source code.

```
1.          i = 1;
2.          min = input[0];
3.          max = input[0];
4.          while (i < length) {
5.               if (input[i] < min) {
6.                    min = input[i];
7.               }
8.               if (input[i] > max) {
9.                     max = input[i];
10.              }
11.              i = i + 1;
12.         }
13.         result = min * max;   // BUG should be result = min + max;
```

Figure 2.1.: A program fragment computing the minimum and maximum from an array of integers, and then the sum of both

The program fragment comprises a bug in Line 13. In order to detect the faulty behavior, we introduce a test suite comprising 5 different test cases (see Table 2.1). Each of them specifies values for the input variables and the expected output. When running our example program on each test case, the fragment returns unexpected values. So how to obtain the root cause for this detected misbehavior?

Test case	Input / `input`	Expected output
A	`[ 1 ]`	`result = 2, min = 1, max = 1`
B	`[ 1, 2 ]`	`result = 3, min = 1, max = 2`
C	`[ 2, 1, 3, 0 ]`	`result = 3, min = 0, max = 3`
D	`[ 0, 1, 2, 3 ]`	`result = 3, min = 0, max = 3`
E	`[ 2, 1 ]`	`result = 3, min = 1, max = 2`

Table 2.1.: The test suite used to verify the program from Fig. 2.1

2.3. Model Based Debugging

The concept of model based debugging (MBDe) is borrowed from model based diagnosis. Model based diagnosis (MBD) was first introduced by [28] and afterwards improved and refined by [92] and [29]. MBD refers to identification of failures within hardware systems. Such a system comprises simple hardware components, e.g., inverters, gates, wired together, such that when applying an input, a specific output can be observed. The information flow is done via actuators (responses) and sensors (triggers). In the case of an error, in MBD, the behavior extracted from the model of the system, is always presumed to be correct, whereas the observed behavior is not. Opposite to this reasoning, in MBDe the system's model is presumed to be error-prone whereas the test case, i.e., observation in MBD terminology, is describing the correct behavior of the program.

In MBD every component C is formally modeled by the relation $\neg AB(C) \rightarrow C$. Predicate $\neg AB(C) = true$ states the correctness of component C. If $false$, i.e., $AB(C)$ is $true$, component C exhibits an unpredictable faulty behavior. MBD reasons about finding an instantiation of the $AB(C)$ which can account for a faulty observation.

Borrowing the modeling from MBD, in MBDe each program statement is seen as a separate component communicating with other components via the dependency given by the statement's variables. A similar abnormal predicate, AB, is introduced, stating the correctness of a statement. By implying a MBDe engine we can compute the statements composing the conflict set explaining the failure of the program on a certain test case.

Model-based diagnosis techniques can be effectively applied to this field. Friedrich et al. [39], Stumptner and Wotawa [102], Mateis et al. [70], Mayer et al. [74], and most recently Köb and Wotawa [61] are examples for the application of model-based diagnosis to program debugging. There are also a lot of papers dealing with debugging which originate from other fields. For example, Staber et al. [95] and Griesmayer et al. [1] incorporate model-checking techniques into debugging. These techniques use ideas from model-based diagnosis and allow for the integration of debugging and verification to some extent. Many of these approaches rely on test cases defining the correct (expected) input/output-behavior of the program.

Past works on the application of model-based diagnosis to program debugging have also investigated different modeling approaches, in particular dependency-based models (e.g., [39]) and value-based models (e.g., [74]).

Let us explain model-based debugging for extracting the root cause of a failure. For this purpose

let us consider the test case `A` from Table 2.1. When running the program on test case `A`, only the following statements are executed:

```
 1.          i = 1;
 2.          min = input[0];
 3.          max = input[0];
 4.          while (i < length) {
12.          }
13.          result = min * max;
```

Let us now assume that each of these statements is represented by a relation (or mathematical equation) $R(v_1,\ldots,v_k)$ over the used variables $v_1,\ldots,v_k$ in that statement. Moreover, let us assume that each relation has an unique corresponding predicate $\neg AB_R$. A relation is used in a derivation if its corresponding variable is true. Formally, we represent this using the horn clause $\neg AB_R \rightarrow R(v_1,\ldots,v_k)$. For example, we represent statement `1.  i = 1;` using rule $\neg AB_1 \rightarrow i = 1$, where $i = 1$ is a relation stating that i has to be 1. We obtain similar rules for the other assignment statements. For simplicity and because of the fact that the while-statement is not executed, we ignore it. We describe the handling of such statements later on in this chapter.

The idea of model-based debugging is to use the set of obtained rules for debugging. We automatically obtain this set, which is the model, from the source code of the program. Hence, there is no need to manually construct the model. An explanation, i.e., a root cause, for a test case, which is called a diagnosis, is an assignment of truth values to the $\neg AB_R$ predicates such that the model together with the test case is satisfiable. Note that we represent the test case itself as a set of relations.

For our example, we have the following model SD and the following set of observations OBS:

$$SD = \left\{ \begin{array}{c} \neg AB_1 \rightarrow i = 1, \neg AB_2 \rightarrow min = input[0], \\ \neg AB_3 \rightarrow max = input[0], \neg AB_{13} \rightarrow result = min \cdot max \end{array} \right\}$$
$$OBS = \{input[0] = 1, min = 1, max = 1, result = 2\}$$

Debugging in the model-based debugging approach is reduced to finding a truth assignment to $\neg AB_R$ predicates that does not lead to a contradiction. If we assume that $\neg AB_2$ is false and all other $\neg AB_R$ predicates to be true, we obtain an inconsistency. From the truth of $\neg AB_{13}$ and the model we obtain that $result = min \cdot max$ must hold. We know that $result = 2$ and $max = 1$ and we are, therefore,

able to compute the value 2 for *min*, which contradicts the expected values in *OBS*, i.e., *min* is expected to be 1. Consequently, the assumed truth assignment cannot be a diagnosis. When applying the same procedure for every truth assignment with one predicate to be false and the other to be true, we only obtain one satisfiable assignment: $\neg AB_{13}$ is false and the other predicates have to be true. We are able to conclude that statement 13 is the only root cause comprising only one statement. Note that in the model-based debugging terminology *AB* stands for *abnormal*. If $\neg AB$ is false for a statement, then the statement has to be *abnormal*.

Automated and algorithmic debugging has a long tradition in research. Shapiro [96] was one of the first presenting an algorithm that guides the user when searching for bugs in Prolog programs. Weiser [113, 114] introduced the theory behind identifying subsets of programs called slices that are responsible for computing an unexpected variable value. He argued that programmers themselves use slices for debugging. Since the beginnings of automated debugging, many other approaches have been presented. We refer the interested reader to Ducassé [34] presenting a general survey on automated debugging, and to Stumptner and Wotawa [100]. In the rest of this chapter we give a brief overview on other current debugging techniques, focusing on program slicing and spectrum-based approaches for debugging.

2.4. Program slicing

Work on program slicing started with Weiser's initial papers [113, 114]. The idea behind slicing is to use the dependence information in a program to find the statement responsible for the computation of wrong variable values. From computational point of view there exist two types of program slicing: static slicing which does not consider the execution trace corresponding to the test case and dynamic slicing which operates solely on the trace of the test case.

The dependence information can be *statically* obtained during program compilation. For this purpose the data and control dependencies in a program have to be taken into account. The drawback of the static slicing approach is that all statements together with the corresponding dependencies are taken into account even when the statements are not executed for a certain test case. For large programs this is of course not feasible.

In order to reduce the size of slices Korel and Laski [62] introduced *dynamic slicing* that makes use of a test case to reduce the size of the slice to only those statements that are executed on the given test

case. Unfortunately, dynamic slices may not contain the root cause and therefore improvements like critical slicing [33] or relevant slicing [132] have been suggested. Gupta and colleagues [48] suggested combining Delta Debugging [131] with forward and backward slicing, which leads to smaller program fragments, i.e., chops, to be considered during a debugging session. Other more recent works on slicing include work by Krinke [63], and Ranganath and colleagues [91]. For a general introduction into slicing we refer the interested reader to Kamkar [58] and Tip [108]. Other important work include Kusomoto et al. [65] where the authors report on an empirical analysis of the applicability of slicing for software debugging. For a more general survey on empirical results and studies about slicing we recommend Binkley and Harman's work [12].

The combination of slicing and other approaches for debugging, testing and validation can be found in many publications. Krinke [64] combined slicing and constraint solving in order to improve accuracy. They used the resulting slicer for validation of measurement software. Kamkar [59] presented an approach that brings together slicing and algorithmic program debugging [96]. The idea is to use slicing in order to reduce heavy user interactions, which is a drawback of algorithmic debugging. [35] uses slicing for reducing the size of models for the purpose of formal verification.

There are also publications introducing the combination of slicing with model-based diagnosis, e.g., [119] and [120]. [119] proves that static slicing and model-based diagnosis based on a dependence model lead to the same results. In [120] the author presents the initial ideas and concepts regarding the combination of dynamic slicing, model-based, and probabilistic reasoning. In [11] the authors discuss the integration of static slicing and probability theory. In particular this work makes use of Bayesian reasoning to learn failure probabilities of statements from execution runs and enhance the corresponding nodes of a program dependence graph with the obtained probabilities. Moreover, the authors discuss the use of the new program dependence graph for debugging.

Other approaches use slicing for debugging of visual programing languages [60]. Here the authors introduce the concept of *Interrogative Debugging* for the visual programing language Alice [6]. They basically track down events connected to certain objects in order to provide an answer to the question of *"why?"* or *"why did not?"* an event occur. The applicability of this method for large complex system was however not thoroughly studied.

We illustrate debugging with relevant slicing and model-based diagnosis [119, 120], using a variant of the program depicted in Fig. 2.1. We assume Line 1 to be faulty, i.e., `i = 2;` instead of `i = 1`, and Line 13 to be correct, i.e., `result = min + max;`. When applying the test cases from Table 2.1, we get the failing test cases B and E. In the first test case *max* and *result* have wrong values, and

in the second *min* and *result*. So what we have to do is to compute the relevant slices for *min*, *max*, and *result* and to combine them appropriately. In [119, 120] a hitting set algorithm is used for this purpose, but for this example the intersection of the slices is sufficient.

A relevant slice [132] for *max*, using test case *B*, is computed by executing the program on *B*. The executed statements form an execution trace from which we extract a directed graph. The edges are the data and control dependencies. For our example, the obtained execution trace looks like follows:

```
 1.          i = 2;
 2.          min = input[0];
 3.          max = input[0];
 4.          while (i < length) {
12.          }
13.          result = min + max;
```

Obviously, Statement 3 and Statement 13 are connected via data dependency, because *min* is defined in 3 and used in 13. Moreover, we also have a data dependency between Statement 1 and 4. When computing the relevant slice we mark the node where *max* is defined the last time and go backwards the data and control dependence edges. For this example, only Statement 3 is marked. However, in relevant slicing also potential influences are considered.

The while-statement is not executed, but executing this statement would also allow defining *max*. Hence, the while statement has a potential influence over the value of *max*, and has to be considered for slicing. We mark Line 4 and go backwards again and therefore mark Statement 1. Hence, in summary, the relevant slice for *max* and test case *B* comprise the statements $\{1,3,4\}$.

For the variable *result* and both test cases, i.e., *B* and *E*, we obtain $\{1,2,3,4,13\}$. And for variable *min* and test case *E* we obtain $\{1,2,4\}$. The intersection of all these relevant slices is the set $\{1,4\}$, which is also the debugging result presented to the user. Hence, in this example we are able to reduce the search space from 13 statements to 2 for debugging. Moreover, the bug location is also included in the result.

2.5. Spectrum-based debugging

Another very promising debugging approach is spectrum-based fault-localization (SFL) [57, 3]. Unlike slicing based techniques or model based techniques, SFL implies no modeling of the program in order to deduce the conflict set. One of the first and most popular tools based on SFL is Tarantula [57]. Recently, Microsoft integrated in its popular IDE, Microsoft Visual Studio 2010, a spectrum-based debugging reasoning engine [93] which proved to be quite efficient in isolating single bugs in large programs. Another difference between model based debugging techniques and SFL, is that SFL makes use of the knowledge about both passing and failing test cases whereas MBDe takes advantage of only failing test cases. In particular, these approaches count how often a statement is executed for passing and failing runs. From this knowledge, a ***similarity coefficient*** is obtained, which allows for ranking statements. The approach requires a test suite that should comprise a large number of test cases. A combination of spectrum-based approaches and model-based approaches for fault localization is discussed in [72]. Their approach focuses first on computing a minimal conflict set using a Model based software debugging (MBSD) engine. The second part implies ranking these components based on the similarity coefficient. This has the advantage of focusing the precise technique of MBDe only on "highly probable" faults. In [4] the authors discuss the use of probabilistic reasoning in spectrum-based debugging for ranking the obtained diagnoses. One of the most recent works on spectrum-based debugging is [5], where the authors focus on multiple faults.

We now illustrate the spectrum-based debugging approach using the modified example program from Figure 2.1, where Line 6 is changed to `max = input[i];` and Line 13 is corrected to `result = min + max;`. The first step of spectrum-based debugging is the computation of a block-hit matrix. A block-hit matrix (or spectrum) h is a $n \times m$ matrix if the considered program Π has n statements, and the test suite TS has m test cases. One entry $h(i, j)$ is set to 1 if the statement i is executed in test case j. Otherwise, $h(i, j)$ is set to 0. Because we want to obtain a ranking of statements based on the program executions and the information regarding passing and failing test cases, we also need an error vector e, with $e(j) = 1$, if test case j causes the program to fail, and $e(j) = 0$, otherwise. The given information can be depicted as:

$$\begin{array}{lccc} & \text{Test cases} & & \\ & \longrightarrow & & \\ \text{Statements} \downarrow & \begin{bmatrix} h_{11} & \dots & h_{1m} \\ \vdots & & \vdots \\ h_{n1} & \dots & h_{nm} \end{bmatrix} & \begin{bmatrix} r_1 \\ \vdots \\ r_n \end{bmatrix} & \downarrow \text{Rank} \\ \text{Error vector} & \begin{bmatrix} e_1 & \dots & e_m \end{bmatrix} & & \end{array}$$

What we want to compute is the rank. A statement is more likely to fail, if it is executed only in failing runs. A statement that is executed only in passing runs is not likely to fail. Hence, we want to reflect these facts formally. For this purpose, in spectrum-based debugging a value $a_{pq}(i)$ for statement i is introduced, where $p,q \in \{0,1\}$, stating the number of passing and failing test cases (parameter q) where a statement i is executed or not executed (parameter p).

$$a_{pq}(i) = \left|\left\{ j | x_{ij} = p \wedge e_j = q \right\}\right|$$

The value of $a_{pq}(i)$ is used to compute a rank. Abreu et al. [3] introduced the Ochiai coefficient used in the molecular biology domain for this purpose. The Ochiai coefficient is defined as follows:

$$S_o(i) = \frac{a_{11}(i)}{\sqrt{(a_{11}(i)+a_{01}(i)) \cdot (a_{11}(i)+a_{10}(i))}}$$

For our running example, the block-hit spectrum and the rank using the test cases A to E from Table 2.1 is given as follows:

Stmnt \ Test case		A	B	C	D	E	Rank S_o
1.	`i = 1;`	1	1	1	1	1	0.63
2.	`min = input[0];`	1	1	1	1	1	0.63
3.	`max = input[0];`	1	1	1	1	1	0.63
4.	`while (i < length) {`	1	1	1	1	1	0.63
5.	`if (input[i] < min) {`	0	1	1	1	1	0.71
6.	`max = input[i];`	0	0	1	0	1	1.00
7.	`}`						
8.	`if (input[i] > max) {`	0	1	1	1	1	0.71
9.	`max = input[i];`	0	1	1	1	0	0.41
10.	`}`						
11.	`i = i + 1;`	0	1	1	1	1	0.71
12.	`}`						
13.	`result = min + max;`	1	1	1	1	1	0.63
Error vector		0	0	1	0	1	

With respect to the rank S_0, Statement 6 is the most likely bug candidate! Hence, in this example the real bug is identified using spectrum-based debugging.

The real power of the SFL lays in the used similarity coefficient. There exist situations when, even after running the test suite against the program, not one but more components (in worst case scenario all) have the highest ranking. In this case the method fails to provide a valuable indicator of the real bug. Another major drawback of this method, indirectly connected to the first one, is that SFL is as strong as the available test suite and, if only one or two test cases are available, SFL cannot be properly exploited. Usually, due to this drawback, SFL is more suited in the testing phase as in the development phase when most of the programmers have access only to one test case. To overcome this problem, Dallmeier et al. proposed in [26] a SFL methodology requiring only one failing test case. Their technique is based on the usage of the AMPLE metric, which needs only one failing test case, but the existence of more positive test cases. However, this should not be a problem since passing test cases are easier to generate than failing ones.

Other popular coefficients are: the *Jaccard* coefficient [82],

$$S_j(i) = \frac{a_{11}(i)}{a_{11}(i) + a_{01}(i)) + a_{10}(i)},$$

and the Tarantula coefficient [57],

$$S_j(i) = \frac{\frac{a_{11}(i)}{a_{11}(i)+a_{01}(i)}}{\frac{a_{11}(i)}{a_{11}(i)+a_{01}(i)} + \frac{a_{10}(i)}{a_{10}(i)+a_{00}(i)}}.$$

In [82] the authors suggest Tarantula as being weaker in predicting the real bug as the Jaccard and Ochiai coefficient.

Due to its reduced computational requirements, SFL was integrated in several projects. Holmes [93] is a tool from Microsoft introduced in the first quarter of 2010 and integrated in the .NET Visual Studio 2010 environment. Holmes performs the debugging/analysis of the programs in two different modes, either non-adaptive debugging or, the faster version, using previously collected debugging information, adaptive debugging. Another SFL-based tool is Ample [26], which was developed at Saarland University. It runs as an Eclipse plug-in and is developed for debugging of Java programs. VIDA [27] is a visual interactive tool developed at Peking University running also as an Eclipse plug-in for debugging Java programs. Based on the analyses of passed and failed test-cases, apart from presenting the fault candidates set, it helps the user in setting breakpoints for further reduction of the bug-search. For C programs there exists the Zoltar tool [109].

SFL is known for decades in the scientific community, but until recently did not caught a lot of attention from real software developers. The recent main-stream tools showed that SFL can be successfully integrated for debugging of complex programs. However the accuracy and the existence of the test suites remain challenges that must be addressed in the future. Combing MBDe techniques with SFL techniques could prove to be an answer to these challenges.

2.6. Other approaches

There are many other approaches for automated debugging. Zeller [131] introduced with delta debugging a technique for test case minimization. The objective in his work is to obtain a smaller test case pool that still reveals the same fault like the original one in an automated way. Based on the given algorithms, Zeller and colleagues [130, 21] also applied the same technique for fault localization. The work is different to slicing-based approaches and model-based approaches because no dependence information given in the source code is used.

Another very promising debugging approach is based on ***genetic programming or program mutations***. Most recently, Weimer and colleagues [112] presented an approach using genetic programming for computing changes necessary for passing a given test suite. The basic underlying idea is to apply mutation and crossover operators to a program until the new program passes all test cases. This approach is in general not feasible. In order to make the approach applicable for larger programs, the authors provide some heuristics. They restrict the search space for possible mutations to expressions already used in a program. Moreover, the authors suggested avoiding computing the mutations for all statements but only for those that are executed when running a test case. The first empirical results provided are very promising. Similar to delta debugging, approaches using mutations do not rely on dependency information. It is interesting to note that there is a relationship between mutation-based approaches and model-based approaches (see [117]). Moreover, Stumptner and Wotawa [102] used expression replacements in their work on model-based debugging of functional programs. These replacements can be seen as mutations of the original expressions.

Console and colleagues [23] introduced model-based debugging of logic programs based on model-based diagnosis [92] improving previous work, i.e., [96]. Bond et al. [16, 15] improves the approach and eliminates some flaws. It is interesting to note that there are other applications to debugging, based on model-based diagnosis, e.g., Liver's work on debugging programs in the telecommunication industry [66, 67]. This work, in contrast to Console et al. and the one presented in this thesis, relies on a model of the program that has to be provided by the user of such a system. Other works in model-based debugging, using different programming languages, include [39, 116, 118] (hardware description language VHDL), [102] (a simple functional language), and [69, 75, 75] (Java). The use of constraints for model-based debugging has been reported in [18, 84, 122, 85].

2.7. Conclusions

Because dynamic slices and program spectra are computed directly from execution traces, there is only a small computational overhead. Therefore, both approaches can be easily adapted to be used for programs of medium to large size. The slicing-based and model-based approaches require at least one failing test case, whereas spectrum-based debugging works only if there are enough positive and negative test cases. The quality of the spectrum-based and the slicing-based approaches is average, meaning that the expected reduction varies between 50 % and 90 % of the code and maybe more in rare situations. The reduction is better when considering model-based debugging because this approach

makes use of both the syntax and the semantics of a program. However, implementing the approach is not trivial because a compiler for model extraction has to be designed and implemented. The mutation based technique, although fairly new ([112]), promises to come up with ways of suggesting repair solutions for a faulty program as well. One weak spot of this approach is the complexity for mutating each statement in the pursuit of passing all test cases. Furthermore the existence of only one test case is in this situation not desirable as we need the existence of multiple positive test case to at least partially confirm the correctness of the proposed repair solution. Usually mutation based debugging is used in combination with spectrum based techniques (See [112]) for focusing only on those statements which are likely to be causing the failure.

A combination of all approaches, e.g., computing spectra using slicing information, and using the obtained diagnoses from the spectrum-based approach for focusing model-based debugging and then use mutation based debugging to further reduce the conflict set resulted from the model based debugger, would help to further improve automated debugging both in terms of reduction capabilities and running time requirements.

In this chapter we presented the most recent developments in automated debugging research. In the last years, the debugging community is focusing mainly on the above described techniques, i.e., slicing-based, spectrum-based, model-based debugging and most recently mutation based debugging. Our main focus is model based debugging (presented in the next chapter).

The following table lists the result of a comparison between the four methods: slicing-based debugging, spectrum-based debugging, model-based debugging and mutation based debugging.

	Program size	#test cases	passing/failing	Quality	Impl.
Slicing	medium/large	≥ 1	failing	avg.	easy
Spectrum	medium/large	$>> 1$	passing/failing	avg.	easy
Model-based	small/medium	≥ 1	failing	opt.	difficult
Mutation debugging	small/medium/large	$>> 1$	failing	avg.	avg./difficult

Chapter 3

Model Based Debugging: A constraint - based approach.

"*Science knows only one commandment - contribute to science.*"
-Bertolt Brecht

In the last three decades, the model based approach was successfully applied in many areas of the artificial intelligence domain. Testing [25], configuration [98] or diagnosis of hardware systems [92, 29] are just some of the AI domains where the model based approach was proven to be not only highly effective and precise, but also highly flexible when it comes to portability between different systems. Due to the advantages of the model based approach, it did not take long until the idea of model based software debugging, or for short model based debugging was introduced. Console et al. [23] were some of the first to introduce this concept in debugging logic programs, being extended later for more complex programing languages like Java [69].

Model-based debugging is an automated debugging method that is based on a formal model of a program. The model together with the test cases is used to check consistency under given assumptions about the correctness of statements. The correctness assumptions are used to invoke or inhibit the corresponding model of a statement. In case of incorrectness, no model is used and, therefore, there are no constraints on the values of variables changed by a statement. Hence, the task of debugging is reduced to finding a set of consistent assumptions. In our work, we implement this idea using

constraints. Our technique relies on encoding the debugging problem as a constraint satisfaction problem (CSP) such that a constraint solver can be used to compute the conflict set for the faulty program. Unlike other model based debugging techniques which rely on some form of program abstraction to decrease the computational complexity, our approach uses the full semantics of the program. Hence the obtained results are more accurate and reliable. However, before being able to perform this conversion, some issues must be tackled. For instance how can you convert a program written in an imperative language, Java in our case, into a declarative one as required by the constraint programming (CP) paradigm? Another issue that must be tackled is the conversion of arrays and function calls. Conditionals and loops are again issues; their dynamic behavior cannot be represented in a CP language. Later on in this chapter we tackle these issues and explain how we can transform a program to its CSP equivalent. This part of my thesis relies on the following published papers:

- *Converting Programs into Constraint Satisfaction Problems* [121];
- *On the Compilation of Programs into their equivalent Constraint Representation* [122];
- *How to debug sequential code by means of constraint representation* [84];
- *Representing Program Debugging as Constraint Satisfaction Problems* [78];
- *Automated debugging based on a constraint model of the program and a test case* [125];

In what follows, we detail our approach. In Section 3.1 we provide the set of definitions required to encode the debugging problem together with the semantics of the analyzed programs. Afterwards, in Section 3.2 we present the encoding process of a program into its static representation. In Section 3.3 we explain how the debugging problem is encoded as constraint satisfaction problem. In Section 3.4 we present the implementation background of our constraint based approach, paying particular interest on the encoding of the constraint solver. In Section 3.5, we analyze the complexity of the proposed algorithm. Last, in Section 3.6 we conclude and present a set of results for our approach.

3.1. Definitions and language semantics

In this section we present the theoretical background of our approach.

First, we give the formal definition for the syntax and grammar of the programming language to be debugged. This is an important step as it states the power and limitation of our debugging algorithm. The language presented here covers all the important constructs that a programming language

supports. However our defined language is not as refined as a state of the art programming language, e.g., for loops it only defines the ***while*** structure - if we want, we can always express the ***do while*** and ***for*** statements as ***while*** statements. At a practical level we debug normal Java programs, but we limit ourselves to the constructs of the presented language. The second part of this section deals with formally providing the set of definitions which help us state the debugging problem for the defined language.

3.1.1. Language Semantics and Grammar

We implemented our constraint based algorithm such that it can operate on Java programs. However, there exists some limitations of the grammar formally describing the language of the debugged programs. These limitations are no impediment in generalizing, if required, our approach for the complete grammar of the Java language, e.g., print statements. In order to be self contained we first formally introduce the syntax and semantics of the programming language used in the rest of this thesis.

The language $\mathcal{L}$ is a simple imperative assignment language defined over numbers, boolean values, and arrays. In this language we ignore variable declarations and type checking. Moreover, pointers are not considered. The restrictions are for keeping the definition overhead as small as possible. We further assume the existence of a semantics function $[\![]\!]$ that maps programs and the current state to a new state. States are represented by environment variables. Before introducing the syntax of $\mathcal{L}$, we assume VAR to be the set of variables used in a $\mathcal{L}$ program and define expressions EXP recursively as follows: EXP represents in the language of $\mathcal{L}$ the set of all basic elements over which the logical and arithmetic operators are defined. Formally, this translates to the following definition.

Definition 1 (Syntax of expressions EXP**)** *The expressions used in $\mathcal{L}$ can be separated into 2 classes: the basic expressions and the combined expressions.*

- Basic expressions*: In EXP we distinguish the following basic expressions:*
  - Boolean values*: true, false are expressions.*
  - Numbers*: n is an expression, if n is a representation of a number, e.g., -1, 0.*
  - Variables*: x is an expression, if $x \in VAR$.*
  - Array access*: $x[E]$, with $x \in VAR$ and $E \in EXP$ is an access to an array element.*

- Combined expressions*: An element of type combined expression from EXP is a boolean or arithmetic relation over basic expression from EXP. The operators allowed in this relation can be either unary: - (minus sign), ! (not), or binary: + (plus), - (minus), * (multiply), / (divide), && (logical and), || (logical or), == (if equal), != (different)≤ (less equal), ≥ (greater equal), ≤ (less), ≥ (greater).*

In the definition of simple expressions, we allow combined expressions to comprise more than one simple expression. Using the above definition of EXP we now define $\mathcal{L}$:

Definition 2 (Syntax $\mathcal{L}$) *The syntax of the programming language $\mathcal{L}$ is given as follows:*

- *A program in $\mathcal{L}$ comprises a sequence of statements $S_1, \ldots, S_n$.*
- *In $\mathcal{L}$ we distinguish the following kind of statements:*
  - Assignment *statements are of the form $x = E$; or x [E'] $= E$; where $x \in VAR$ is a variable, and E, $E' \in EXP$ can be either of type basic expressions or combined expressions.*
  - Conditional *statements are of the form if E { B_1 }, where $E \in EXP$ is a combined expression returning a boolean value, and B_1 is a program written in $\mathcal{L}$, optionally comprising an else block of the form else { B_2 }, where B_2 is again a program.*
  - Loop *statement are of the form while E { B } where $E \in EXP$ is a combined expression and B is a program written in $\mathcal{L}$.*

Note that in the given definitions the programs in $\mathcal{L}$ and all constructs are represented using an underscore. This is for distinguishing the syntactical part of $\mathcal{L}$ from its semantics. However, in the rest of this work we do not use the underscore in our examples. For example, the program given in Chapter 2, Figure 2.1 is obviously a program written in $\mathcal{L}$.

In the following, we define the semantics of $\mathcal{L}$. For this purpose we introduce a function $[\![]\!]$ mapping constructs from $\mathcal{L}$ together with the current state to a new state. By successively applying the semantics function, the final state of a program, given an initial state, is defined. The state in our context comprises the variable environment and a memory. The variable environment $\omega : VAR \mapsto DOM$ is a function mapping variables from VAR to its values in DOM. The set of possible values DOM comprises the boolean values ($\mathbf{T}, \mathbf{F}$), all numbers (-1,0,1,2,...,1.2e5,...), and all possible array identifiers IDX. We assume that array identifiers from IDX are unique and start with a @ followed by a natural

number. The memory itself is a function $\sigma : IDX \times \mathbb{N} \mapsto DOM$ mapping array elements to their value. Moreover, we assume a function $length$ returning the length of an array.

Note that variable environments induce set of pairs, i.e., $E_\omega = \{(x,v)|\omega(x) = v\}$. In the following we use the set representation and the function representation of variable environments interchangeably. For simplicity we introduce a set of environments ENV and a set of memories MEM containing all possible environments and instances of memory respectively.

We start with the definition of the semantics of expressions.

Definition 3 (Semantics of EXP) *Given a state $(\omega,\sigma) \subseteq ENV \times MEM$. The semantics of EXP is defined as follows:*

- Basic expressions
  - Boolean values*:* $[\![\underline{true}]\!](\omega,\sigma) = \mathbf{T}$, $[\![\underline{false}]\!](\omega,\sigma) = \mathbf{F}$
  - Numbers*:* $[\![n]\!](\omega,\sigma) = n$
  - Variables*:* $[\![\underline{x}]\!](\omega,\sigma) = \omega(x)$
  - Array access*:* $[\![x\underline{[}E\underline{]}]\!](\omega,\sigma) = \sigma(\omega(x), [\![E]\!](\omega,\sigma))$
- Combined expressions*: In the following we assume that the operators uop and op are represented by themselves.*
  - $[\![uop\ E]\!](\omega,\sigma) = uop\ ([\![E]\!](\omega,\sigma))$
  - $[\![E\ op\ E']\!](\omega,\sigma) = [\![E]\!](\omega,\sigma)\ op\ [\![E']\!](\omega,\sigma)$

Note that in $\mathcal{L}$, expressions do not change the state. We now use the semantics of EXP to define the semantics of programs.

Definition 4 (Semantics of $\mathcal{L}$) *Given a state $(\omega,\sigma) \subseteq ENV \times MEM$. The semantics of $\mathcal{L}$ is defined as follows:*

- Sequence of statements*:* $[\![S_1 \ldots S_n]\!](\omega,\sigma) = [\![S_2 \ldots S_n]\!]([\![S_1]\!](\omega,\sigma))$ *with* $[\![]\!](\omega,\sigma) = (\omega,\sigma)$ *as the base case of this inductive definition over the number of statements.*
- Assignments*:*
  - Variable at the left side of the assignment*:* $[\![x \underline{=} E\ \underline{;}]\!](\omega,\sigma) = (\omega',\sigma)$ *with* $\forall y \in VAR \wedge y \neq x : \omega'(y) = \omega(y)$ *and* $\omega'(x) = [\![E]\!](\omega,\sigma)$.

  - Array access at the left side: $[\![x\ \underline{[}\ E'\ \underline{]}\ \underline{=}\ E\ \underline{;}]\!](\omega,\sigma) = (\omega,\sigma')$ *where* $\forall i \in IDX \wedge i \neq \omega(x) : \forall j \in \{0,\ldots,length(i)\} : \sigma'(i,j) = \sigma(i,j)$, *and* $\forall j \in \{0,\ldots,length(i)\} \wedge j \neq e : \sigma'(\omega(x),j) = \sigma(\omega(x),j)$ *and* $\sigma'(\omega(x),e) = [\![E]\!](\omega,\sigma)$ *with* $e = [\![E']\!](\sigma,\omega)$.

- Conditionals: *We distinguish 2 cases:*

  - $$[\![\underline{if}\ E\ \underline{\{}\ B_1\ \underline{\}}]\!](\omega,\sigma) = \begin{cases} [\![B_1]\!](\omega,\sigma) & \textbf{if } [\![E]\!](\omega,\sigma) = \mathbf{T} \\ (\omega,\sigma) & \textbf{if } [\![E]\!](\omega,\sigma) = \mathbf{F} \end{cases}$$

  - $$[\![\underline{if}\ E\ \underline{\{}\ B_1\ \underline{\}}\ \underline{else}\ \underline{\{}\ B_2\ \underline{\}}]\!](\omega,\sigma) = \begin{cases} [\![B_1]\!](\omega,\sigma) & \textbf{if } [\![E]\!](\omega,\sigma) = \mathbf{T} \\ [\![B_2]\!](\omega,\sigma) & \textbf{if } [\![E]\!](\omega,\sigma) = \mathbf{F} \end{cases}$$

- Loops:

$$[\![\underline{while}\ E\ \underline{\{}\ B\ \underline{\}}]\!](\omega,\sigma) = \begin{cases} [\![\underline{while}\ E\ \underline{\{}\ B\ \underline{\}}]\!]([\![B]\!](\omega,\sigma)) & \textbf{if } [\![E]\!](\omega,\sigma) = \mathbf{T} \\ (\omega,\sigma) & \textbf{if } [\![E]\!](\omega,\sigma) = \mathbf{F} \end{cases}$$

Note that in the above definition, an assignment can change the value of a particular array element. If two variables point to the same array, such a change leads to a side effect.

Another limitation of the above grammar is the lack of formal definition for function calls. On a practical level, our approach offers support also for this type of functional structures: by method in-line copy. This has less to do with the formal definition of the language, being more of a "syntactic sugar" issue.

Definition 5 (Grammar) *The grammar associated to the syntax of a programing language is a tuple* (V_N, V_T, S, Φ) *such that*

- V_N *represents the non-terminal nodes of the production rules, i.e., placeholders. For example assignment is a placeholder for the actual equation.*
- V_T *represents the terminal nodes, e.g. numbers, variables, operators.*
- S *is the program entry point, i.e., the start node for the production rules.*
- $\Phi \in V_T$ *represents the production rules of the grammar, i.e., the allowed combinations for the syntax of the language.*

Usually the terminal nodes are designated in the production rule by underlining them.

The grammar associated to the syntax of $\mathcal{L}$ is defined as follows:

- $V_N = \{Start, ASSIG, COND, LOOP, Statement, EXP, ELSE\}$

- $V_T = \{\varepsilon, \underline{begin}, \underline{end}, \underline{var}, \underline{num}, \underline{bool}, \underline{if}, \underline{then}, \underline{else}, \underline{while}, \underline{boolOp}, \underline{binOp}, \underline{unOp}, \underline{[}, \underline{]}, \underline{=}\}$, whereas $\underline{var}$, $\underline{num}$, and $\underline{bool}$ are designating the elements of type Definition 3. All allowed binary operators, $\underline{binOp}$, unary operators $\underline{unOp}$ and boolean binary operators $\underline{boolOp}$, are found in Definition 3. The relation between $\underline{boolOp}$ and $\underline{binOp}$ is $boolOp \subseteq binOp$

- $S = \{Start\}$

- $\Phi = \{$

  $Start \rightarrow \underline{begin}\ \ Statement\ \ \underline{end}$
  $Statement \rightarrow ASSIG\ \ \underline{;}\ \ Statement$
  $Statement \rightarrow COND\ \ Statement$
  $Statement \rightarrow LOOP\ \ Statement$
  $Statement \rightarrow \varepsilon$
  $A \rightarrow var\ \ \underline{=}\ \ EXP$
  $A \rightarrow var\underline{[}EXP\underline{]}\ \ \underline{=}\ \ EXP$
  $EXP \rightarrow var$
  $EXP \rightarrow num$
  $EXP \rightarrow bool$
  $EXP \rightarrow var\underline{[}EXP\underline{]}$
  $EXP \rightarrow EXP\ \ \underline{binOp}\ \ EXP$
  $EXP \rightarrow \underline{unOp}\ \ EXP$
  $COND \rightarrow \underline{if}\ \ \underline{(}\ \ E\ \ \underline{boolOp}\ \ E\ \ \underline{)}\ \ \underline{then}\ \ \underline{begin}\ \ Statement\ \ \underline{end}\ \ ELSE$
  $ELSE \rightarrow \underline{else}\ \ \underline{begin}\ \ S\ \ \underline{end}$
  $ELSE \rightarrow \varepsilon$
  $LOOP \rightarrow \underline{while}\ \ \underline{(}\ \ E\ \ \underline{boolOp}\ \ E\ \ \underline{)}\ \ \underline{begin}\ \ Statement\ \ \underline{end}$

  $\}$

According to the Chomsky [20] grammars hierarchy, the above grammar is a context free grammar (also called type 2 grammar). The context free grammars usually represent the theoretical foundation

for the syntax of any programming language. In a context-free grammar, every production rule fulfills the following requirement $|\alpha| \leq |\beta|$, $\alpha \in V_N$, where $\alpha \rightarrow \beta$.

The problem of the above grammar is that it is left recursive, i.e., non-deterministic. The ambiguity, i.e., left recursion, is due to the production rule: $EXP \rightarrow EXP\ \underline{binOp}\ EXP$. However a small patch can be applied to remove this ambiguity. We introduce a new non-terminal node, PATCH, which modifies our grammar as follows:

- The production rule $EXP \rightarrow EXP\ \underline{binOp}\ EXP$ is replaced by $PATCH \rightarrow \underline{binOp}\ EXP\ PATCH|\varepsilon$
- Every production of type $EXP \rightarrow \alpha$ is replaced by $EXP \rightarrow \alpha\ PATCH$,whereas α is a placeholder for any right-side production rule.

Hence, after applying our patch, V_T and S remain unchanged, V_N becomes $V_N = \{Start, ASSIG, COND, LOOP, Statement, EXP, ELSE\} \bigcup \{PATCH\}$ and Φ becomes:

$Start \rightarrow \underline{begin}\ \ Statement\ \ \underline{end}$
$Statement \rightarrow ASSIG\ \ \underline{;}\ \ Statement$
$Statement \rightarrow COND\ \ Statement$
$Statement \rightarrow LOOP\ \ Statement$
$Statement \rightarrow \varepsilon$
$A \rightarrow \underline{var}\ \ \underline{=}\ \ EXP$
$A \rightarrow \underline{var}\underline{[}EXP\underline{]}\ \ \underline{=}\ \ EXP$
$EXP \rightarrow \underline{var}\ \ PATCH$
$EXP \rightarrow \underline{num}\ \ PATCH$
$EXP \rightarrow \underline{bool}\ \ PATCH$
$EXP \rightarrow \underline{var}\underline{[}EXP\underline{]}\ \ PATCH$
$PATCH \rightarrow \underline{binOp}\ \ EXP\ \ PATCH\ \ |\ \ \varepsilon$
$EXP \rightarrow \underline{unOp}\ \ EXP\ \ PATCH$
$COND \rightarrow \underline{if}\ \ \underline{(}\ \ E\ \ \underline{boolOp}\ \ E\ \ \underline{)}\ \ \underline{then}\ \ \underline{begin}\ \ Statement\ \ \underline{end}\ \ ELSE$
$ELSE \rightarrow \underline{else}\ \ \underline{begin}\ \ S\ \ \underline{end}$
$ELSE \rightarrow \varepsilon$
$LOOP \rightarrow \underline{while}\ \ \underline{(}\ \ E\ \ \underline{boolOp}\ \ E\ \ \underline{)}\ \ \underline{begin}\ \ Statement\ \ \underline{end}$

3.1.2. Definitions

Our approach implies the usage of a test case for identifying the faults in the program. Hence, in order to define the debugging problem for programs written in $\mathcal{L}$, we first introduce the formal definition of a test case. Basically, a test case comprises information about the values of input variables and some (but not necessarily all) information regarding the expected output. Theoretically it would also be possible to define expected values for variables at arbitrary locations in the code, but because of simplicity, we do not extend the definition in this respect.

Definition 6 (Test case) *A test case for a given program $\Pi \in \mathcal{L}$ is a tuple (I,O) where $I \in ENV$ is an input environment specifying the values of all variables used as inputs, and $O \in ENV$ is the expected output environment.*

Note that there are no restrictions on the output environment. Hence, an empty set might also be a valid expected output environment. Of course for validation and verification, the expected output has to be defined. A given program $\Pi \in \mathcal{L}$ passes a test case $t = (I,O)$ if and only if $[\![\Pi]\!]I \supseteq O$. Otherwise, we say that the program fails. Because of the use of the $\supseteq$ operator, also partial test-cases are allowed, which do not specify values for all output variables.

Definition 7 (Test suite) *A test suite TS for a program $\Pi \in \mathcal{L}$ is a set of test cases.*

A test suite can be partitioned into two disjoint sets comprising only passing (*PASS*) respectively failing (*FAIL*) test cases, i.e., $TS = PASS \cup FAIL \wedge PASS \cap FAIL = \emptyset$. Formally, we define these two subsets as follows:

$$PASS = \{(I,O) | (I,O) \in TS, [\![\Pi]\!]I \supseteq O\}$$
$$FAIL = TS \setminus POS$$

We are now able to formalize the debugging problem.

Definition 8 (Debugging problem) *A debugging problem is a tuple (Π, TS), where $\Pi \in \mathcal{L}$ is a program, $TS = PASS \cup FAIL$, a test suite with at least one failing test case, i.e., $|FAIL| \geq 1$.*

Definition 9 (Correctness Assumption) *Given the set of all statements S of a program $\Pi \in \mathcal{L}$, then $\forall s_i \in S, i = 1...|S|$, the predicate $AB(s_i)$ denotes the assumption about the correctness of statement s_i. if $AB(s_i)$ is true, then statement s_i is assumed to be incorrect and any output value is presumed to be possible except for the correct one. A statement cannot be both in an incorrect and correct state.*

Definition 10 (Diagnosis) *Let $\Delta \subseteq STMNTS(\Pi)$ be a set of statements such that $\forall j = 1...|\Delta|, AB(s_j) == true$,where $STMNTS$ is a function returning all statements of a program. We call Δ the valid explanation or diagnosis for the debugging problem (Π, TS), if Δ is consistent with all the test cases.*

The motivation behind this definition is that a program passing the whole test suite needs not to be considered for debugging. A solution to the debugging problem would be in general a variant of the original program that passes the whole test suite. In the context of this thesis, a solution, i.e., diagnosis, is a set of statements that, when assumed to behave wrong, explains all failing test cases.

Lemma 1 (Minimum diagnosis) *A diagnosis Δ for a debugging problem (Π, TS) is called minimal, when there exists no other diagnosis Δ' such that $\Delta' \prec \Delta$, $\prec$ denotes the subset operator.*

A diagnosis Δ is of minimal cardinality when no other diagnosis Δ' exists such that $|\Delta'| < |\Delta|$

Definition 11 (Conflict) *Given a failing test case $T \in TS$ of a debugging problem (Π, TS). A conflict is a set of one or more statements $C = \{s, s \in STMNTS(\Pi)\}$ such that $\forall s_i \in C, \neg AB(s_i) \rightarrow s_i$ is not consistent with the behavior resulted from test case T, $STMNTS$ is the function returning all statements of the program Π.*

Lemma 2 (Minimal Conflict) *A conflict is minimal when none of its subset is minimal. The set of all program statements is a supra set of all minimal conflicts.*

Definition 12 (Conflict Set) *The conflict set of a debugging problem (Π, TS) comprises all the minimal conflicts computed with respect to TS and Π.*

In our constraint based-approach, the definition of a conflict set is less important since our modeling allows us to directly extract the minimal diagnosis. However in model based diagnosis and in some model based debugging approaches, after conflict set extraction, an intermediate step is required for computing the minimal diagnosis. One of the most popular techniques implied for computing the minimal diagnosis is the *hitting set algorithm* introduced by Reiter et. all [92]. Slicing based techniques imply also the hitting set for computing the minimal explanation for conflicts computed for different slicing criteria. Another usage of the hitting set is for integration of multiple test cases in debugging VHDL programs [10].

In the next section we depict the steps for replacing the dynamic behavior of the analyzed program with a statical one. Additionally, we model the new resulted program such that the imperative behavior can also be seen as a declarative one, hence in the form specified by the constraint programing paradigm.

3.2. Static Conversion

There are certain challenges that we must undergo before converting a program into its constraint representation. The first challenge is related to answering questions about language dynamic behavior: *what happens with the loop and conditional structures?* or *how are method calls integrated?*.

The second challenge refers to the differences between expressiveness of the imperative languages of $\mathcal{L}$ and expressiveness of the declarative constraint programming language. For example having the following small program fragment in the imperative language of $\mathcal{L}$:

```
{
1. result = 0;
....
2. a = 0;
3. b = 5;
4. result = a + b; // result = 0+5 = 5
.....
}
```

would yield no errors and would compute the correct value of 5 for the variable *result*. Converting the same program in an auxiliary constraint programing language yields:

```
1. equals(result, 0) #result = 0;
....
2. equals(a, 0)      #a = 0;
3. equals(b, 5)      #b = 5;
4. sum(a,b,result)   #result = a + b => result = 5 !!! Inconsistency
.....
```

The above constraint system is however not solvable. In a declarative language, unlike imperative languages, all the statements must be *true* at same time. This property translates in our example to the following inconsistency: $(1.)$ $result = 0$ and $(4.)$ $result = 5$, e.g. $0 = 5 = result$, which of course cannot be satisfied.

The previous example shows just one possible problem which can appear if the conversion between the imperative and declarative language is an *one to one* statement conversion.

We propose to convert programs into constraints in three steps.

* In the first step, the program Π from $\mathcal{L}$ is converted into its loop-free representation Π_{LF}. In this representation all loops are replaced by nested conditional statements where the nesting depth has to be larger or equal to the maximum number of iterations of the loops considering the given test suite.
* From Π_{LF} we obtain a static single assignment version Π_{SSA}. In this representation every variable is defined only once.
* The reason for the SSA representation is the easy conversion into the constraint representation CON_{Π}, performed in the last step. Given the SSA form, the translation comprises replacing the statements with language constructs used by the implied constraint solver, and adding information regarding the correctness assumptions.

In summary, the conversion process looks like:

$$\Pi \in \mathcal{L} \longrightarrow \Pi_{LF} \in \mathcal{L} \longrightarrow \Pi_{SSA} \in \mathcal{L} \longrightarrow CON_{\Pi}$$

In the following subsections we discuss each conversion step and prove the correctness and completeness under certain restrictions. It is worth noting that the whole conversion can be automated and there is no need of user interventions.

3.2.1. Loop elimination

Dynamic structures need further transformation before they can be converted into their constrain representation. Our algorithm implies a static analysis of the program, which means, no direct estimation of the execution is possible. Hence, at the moment of the analysis, we cannot know how an *if* or *while* structure is executed. In the case of an *if* structure we can interpret it in both directions in the constraint system. However loops cannot be directly converted to a constraint representation. Therefore, a step for eliminating loops is necessary.

The elimination of loops for various purposes like verification [22] and test case generation [41] is not new and is based on *unrolling* loops, i.e., to create a loop-free program by replacing the loop of

the original program by a set of nested `if`-statements (e.g., see for example also [18] and [84]). When executing while-statements they behave like a conditional statement in one step. If the condition is fulfilled the statements in the block are executed and the while-statement is executed again afterwards. Otherwise, the while-statement is not executed. Hence, it is semantically correct to represent while-statements using an infinite number of nested if-statements, i.e., `while ( C ) { B }` is equivalent to the representation given in Fig. 3.1, where C represents the condition, and B the statements in the sub-block of the while statement.

- Hence, we can define loop-elimination as a recursive function where n is the number of iterations:

$$LF(\text{while } C\ \{B\}, n) = \begin{cases} \text{if } C\{B\ LF(\text{while } C\ \{B\}, n-1)\} & \text{if } n > 0 \\ \varepsilon & \text{otherwise} \end{cases}$$

This modeling of the loop structures obviously implies that the maximum number of iterations is known in advance. This is a mandatory condition for the intermediate loop-free model $\Pi_{LF} \in \mathcal{L}$ to be equivalent with the original program $\Pi \in \mathcal{L}$. It's worth nothing to us if the loop-free program Π_{LF} is an incomplete estimation of Π. To avoid this situation an overestimation of the loop is required. Overestimating the loop does not have any negative effect on the program output, i.e., the loop condition is tested before entering a new *if*, but can slightly increase the computation time for the CSP solver.

Under this assumption the loop-free program is equivalent to the original program. Note that program debugging is based on one or more test cases, and executing the program for those test cases can also yield an overestimation of the maximum number of iterations in the faulty program for the given test suite. Another possibility is to deduce this number directly from the program's specification. Of course the number of necessary iterations might be large, causing a substantial increase of the size of the loop-free variant. As we will show in a later chapter, this number has a certain influence on the debugging complexity, but becomes neutral after a certain level. However, usually test cases are designed in order to reveal failures using inputs of manageable size. Hence, a small number of iterations is often sufficient for detecting faults (e.g., see [55]). It can be easily proven that the following corollary, stating the correctness of the conversion into loop-free programs, holds.

Corollary 1 *Given a debugging problem* (Π, TS). *Let* Π_{LF} *be the corresponding loop-free program of* Π*, where all loops are replaced by nested-if statements and where the nesting depth is larger than the number of loop iterations for each test case in* TS. Π_{LF} *and* Π *compute the same output values for all test cases in* TS*, i.e.,* $\forall (I,O) \in TS : [\![\Pi]\!]I = [\![\Pi_{LF}]\!]I$.

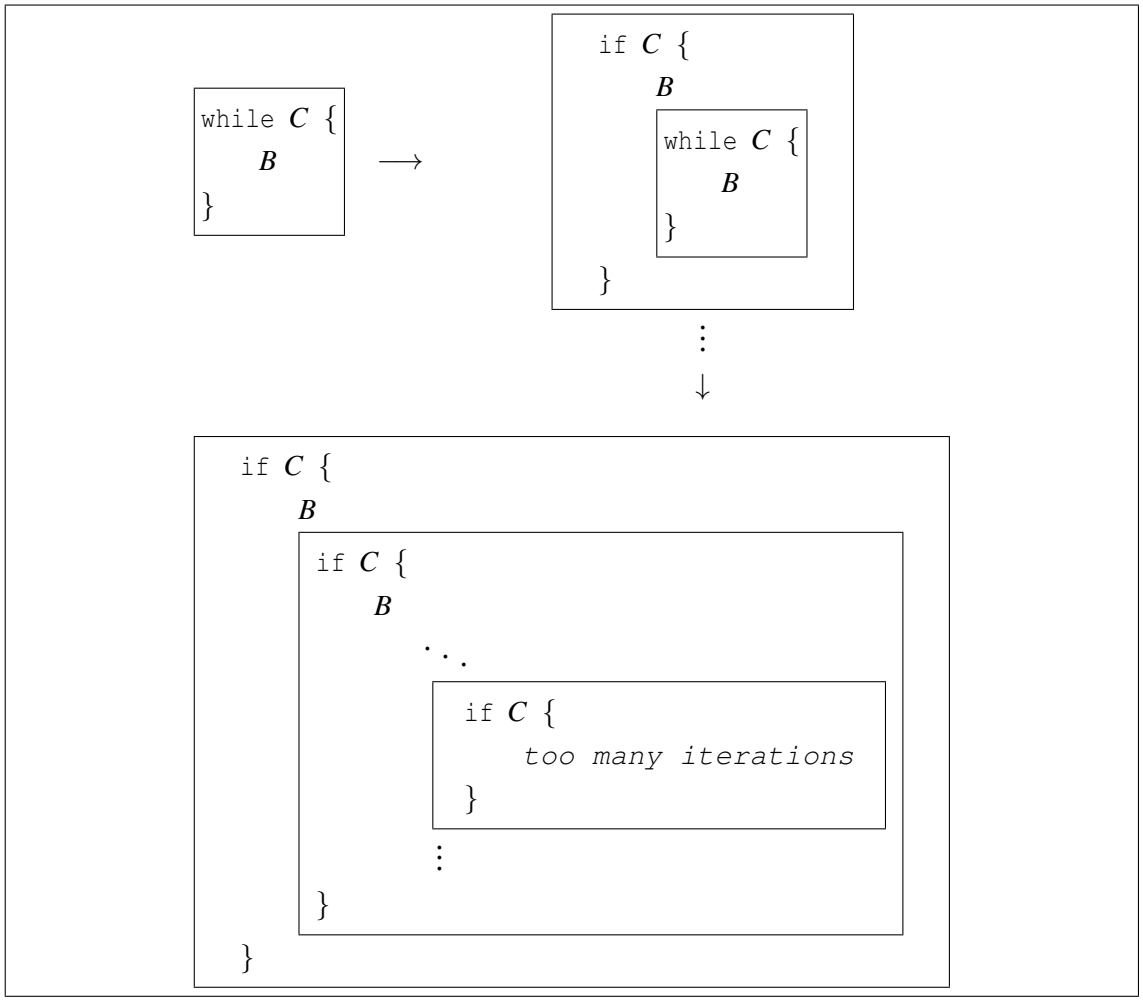

Figure 3.1.: Loop unrolling

```
1.          i = 1;
2.          min = input[0];
3.          max = input[0];
4.          while (i < length) {
5.               if (input[i] < min) {
6.                    min = input[i];
7.               }
8.               if (input[i] > max) {
9.                     max = input[i];
10.              }
11.              i = i + 1;
12.         }
13.         result = min * max;   //error should be result = min + max.
```

Figure 3.2.: A program fragment computing the minimum, maximum, and sum of both for an array of integers

Let's recall now the example program given in the Introduction, depicted in this chapter in Fig. 3.2. In this program we have one *while* structure and two *if* statements.

Let's presume that for the program from Fig.3.2 we assume a number of two iteration for replacing the *while* statement. The unrolled version of the program is given in Fig. 3.3.

Method calls are not directly modeled in the of $\mathcal{L}$ language. Implying an intermediate step for replacing all method calls, transforms a program Π into its specific L representation. Converting *method calls* is somehow similar to the *while* structure conversion, i.e., the method call is replaced with its body overall where there exists an invocation to it. There exist two possibilities to invoke a method:

1. As a stand alone statement, i.e., it returns *void*.

2. In an assignment or conditional statement, i.e., the method has a return statement.

We use Algorithm 1 (*AlgorithmReplace_Method*) for replacing a method call in a program.

```
1.          i = 1;
2.          min = input[0];
3.          max = input[0];
4.          if (i < length) {
5.               if (input[i] < min) {
6.                    min = input[i];
7.               }
8.               if (input[i] > max) {
9.                     max = input[i];
10.              }
11.              i = i + 1;
12.              if (i < length) {
13.                   if (input[i] < min) {
14.                        min = input[i];
15.                   }
16.                   if (input[i] > max) {
17.                        max = input[i];
18.                   }
19.                   i = i + 1;
20.              }
21.         }
22.     result = min *  max;
```

Figure 3.3.: The loop-free version of the program in Fig.2.1 for 2 iterations.

Algorithm 1 Replace_Method $(ST, Meth, Arguments, returnType)$

Require: The program's statement ST where the method is invoked, the method's body without return statement $Meth$, its arguments $Arguments$, and the return value $returnStatement$.

Ensure: In-line replacement of a method call.

1: **if** $isEmpty(Arguments) == false$ **then**
2: **for all** $v_i \in Arguments$, $i = 1 \ldots |Arguments|$ **do**
3: Create before ST the local auxiliary variable, v_{auxi} of type v_i and $v_{auxi} = v_i$.
4: **end for**
5: **end if**
6: **for all** Variables $var_i \in Meth, i = 1...$ **do**
7: Create local corresponding local variable var_i_Meth, before ST,
8: **end for**
9: Copy the method's block, $Meth$ before ST(ignoring its variable declarations).
10: **if** $isEmpty(returnType) == false$ **then**
11: Create before ST the local variable $returnType_{aux}$ of type $returnType$ add $returnType_{aux} = returnType$, and replace in body of ST the invocation of $Meth$ by $returnType_{aux}$.
12: **else**
13: delete ST from program.
14: **end if**

3.2.2. Building the Static Single Assignment (SSA) form

There exists an intensive discussion upon which analysis technique, i.e., static or dynamic, is best suited for MBDe. The dynamic approach, i.e., based on the program's trace, is very scalable for large programs. By using only the program's trace, there exists the advantage of focusing the search space only on those statements involved in the execution of the test case. This translates to a smaller program than the original one, hence, less time required by the algorithm to compute the conflict set.

There is however a drawback of the dynamic approach. There exist situations when the error is not contained in the trace, and therefor never revealed. Let's consider the small program from Fig.3.4. The error is found in statement 3. Due to the fact that this conditional statement evaluates to $false$, it will not be included in the trace. Basically the debugging algorithm can never identify it as a possible explanation for the program failure.

```
.....
1.  div = 2;
2.  result = 4;
3.  if (div == 0) //error should be !=
4.  result = result / 2;
.....
```

Figure 3.4.: A program for computing the division of two natural numbers.

```
.....
1.div = 2;
2.result = 4;
.....
```

Figure 3.5.: The program's trace

The statically analysis of the program implies the whole program as model for the debugging engine. We use the test-case suite only for extracting the number of loop-iterations. And even if after analyzing the test case suite, a loop is marked as *never executed*, we include it in the model as a *one iteration loop*. Hence we eliminate the drawback of the trace-based analysis of the program.

Elimination of the *while* loops and method calls was the first step in obtaining the modeling of program $\Pi \in \mathcal{L}$ into its CSP equivalent. An intermediate representation of all the statements (comprising now only conditional statements and assignments) must be performed. We call this intermediate representation the ***static single assignment (SSA)*** form. In what follows we depict the conversion process of all possible program structures, e.g., conditionals, assignments, arrays.

Before explaining what the SSA form represents, let's start by considering the following program together with its constraint representation:

```
1. result = 0;          1. equals(result, 0) #result = 0;
    ......                   ......
2. a = 0;               2. equals(a, 0)      #a = 0;
3. b = 5;               3. equals(b, 5)      #b = 5;
4. result = a + b;      4. sum(a,b,result)   #result = 0= 5 !!! Inconsistency
    ......                   ......
```

Even if the above program written in $\mathcal{L}$ is correct, its direct constraint language representation is

not (due to fact that every constraint language is declarative, in our example *result* must be both equal to 0 and 5). Let's rewrite the above example as follows.

```
1. result_1 = 0;              1. equals(result_1, 0)     #result_1 = 0;
    ......                         ......
2. a_1 = 0;                    2. equals(a_1, 0)          #a_1 = 0;
3. b_1 = 5;                    3. equals(b_1, 5)          #b_1 = 5;
4. result_2 = a_1 + b_1;       4. sum(a_1,b_1,result_2)   #result_2 =5
    ......                         ......
```

From the semantical behavior point of view, the new program is identical with the original one, but has the property that its corresponding constraint system is no longer inconsistent. We applied a small trick, i.e., we rename all the program's variables such that there exists no assignment with the same left-side. More practically: by the second usage of the *result* variable we rename it to *result_2*. Hence the constraint system no longer finds an inconsistency between the two values of *result*. Due to the renaming process, the statements $result\_1 = 0$ and $result\_2 = 5$ can now be "true" simultaneously.

Our constraint representation requires that all left-side variables in the program have unique names, i.e., each variable should be defined only once. Hence, we use the *static single assignment (SSA)* form, which is an intermediate representation of the program with the property that no two left-side variables have the same name (see [24, 17, 111]). This is achieved by replacing each left-side variable with a new variable whose name is composed of the name of the original variable plus a unique index as suffix, see Fig. 3.6. Since all variables are defined only once, the SSA form allows for a clear representation of the dependencies that are established between different variables inside the corresponding program. The SSA representation of a program is also an intermediate step in the compiling process; basically before compilation, a file is first transformed into its SSA representation. The SSA form is then used as input for the compiler.

In order to obtain the SSA form of a program it is also necessary to convert loops, arrays and conditional statements. As loops are, in our approach, represented by nested `if`-statements, we only need to consider the conversion of conditional statements and array structures.

In the language of $\mathcal{L}$ a valid ***conditional statement*** is of the form:

$$\texttt{if(}\texttt{cond}_{\texttt{expr}}\texttt{) then } \{\texttt{...}\} \texttt{ else } \{\texttt{...}\}$$

Note that the notation $\texttt{x}_{\texttt{expr}}$ denotes a whole expression rather than a single variable. Stepwise the conversion of conditional blocks works as follows:

1. The value of the evaluated condition $\texttt{cond}_{\texttt{expr}}$ is stored in a new boolean variable `cond_i`, where `i` is a unique index.
2. The `if`- and the `else`-branches are converted separately. For both branches, new variables with unique indexes are introduced. The statements of both branches are concatenated; i.e., the program in SSA form will execute the statements of either branch in every run.
3. New variables are added; they have the value of the corresponding variables in the original program *after* the execution of the `if`- or `else`-branch, respectively. The values of these new variables depend on the indexed variables which were introduced for the branches *and* on the boolean condition $\texttt{cond}_{\texttt{expr}}$. For the evaluation of those values we define the Φ-function:

$$\Phi(\texttt{v\_j}, \texttt{v\_k}, \texttt{cond\_i}) \stackrel{\text{def}}{=} \begin{cases} \texttt{v\_j} & \text{if } \texttt{cond\_i} = \mathit{true} \\ \texttt{v\_k} & \text{otherwise} \end{cases}$$

For example, the corresponding SSA form of the program fragment

$$\texttt{if(cond}_{\texttt{expr}}\texttt{)} \ \{\texttt{x = e1}_{\texttt{expr}}\texttt{;}\} \ \texttt{else} \ \{\texttt{x = e2}_{\texttt{expr}}\texttt{;}\}$$

is given as follows:

```
cond_i = cond_expr;
x_j = e1_expr;
x_k = e2_expr;
x_l = Φ(x_j, x_k, cond_i);
```

We handle ***arrays*** similar to how we handle assignment statements. For the purpose of explaining the conversion of arrays, we assume an array `A` of length $n > 0$ with elements $\langle a_1 ... a_i ... a_n \rangle$. The access to elements is assumed to be done using the `[]` operator, which maps from A and a given index i to the array element a_i. We now discuss two cases, i.e., the array is used at the right side of an assignment, and the array occurs at the left side of an assignment.

In a statement of type `z = A[E]`, i.e. the array access is found at the right hand side of an equation with index E, `A` is represented in the SSA form as `A_k`, where k is the currently given unique index k for the array `A`.

The more difficult part is handling statements like `A[m] = x_expr`, where the array is on the left hand side of an assignment. For this purpose we use an ***update array*** function [24]: $\Psi(A, i, exp)$. The function Ψ returns a new array A', which, except for the value at index i, has the same values as the array A. For example, if we encounter statement `A[m] = x_expr` during conversion, then its SSA form is $A_{k+1} = \Psi(A_k, m, \mathtt{x_{expr}})$. We now formally define the function Ψ. Assume a program fragment `A[`i`] = `$f(\vec{x})$, where the i-th element of `A` is set to the outcome of function f, given parameters $\vec{x}$. This statement only changes the i-th element, but not the others.

```
{ A } // A before the statement
A[i] = f(x)
{ A' } // A after the statement
```

The new value after executing the statement is given as follows: `A'[`i`] = `$f(\vec{x})$ and $\forall j \in \{1,\ldots,n\}, i \neq j$: `A'[`$j$`] = A[`$j$`]`. As a consequence, we say the function Ψ (written as `Psi` in the source code) has to implement this semantics in order to allow replacing the original statement with `A = Psi(A,i,`$f(\vec{x})$`)`.

Before stating the correctness of this conversion step, we give another example:

```
1.      min = input[i];
2.      input[i] = 5;
3.      input[2] = input[1] + 5;
```

According to our conversion rules we obtain the following SSA representation:

```
1.     min_0 = input_0[i_0];
2.     input_1 = Psi(input_0,i_0,5);
3.     input_2 = Psi(input_1,2,input_1[1] + 5);
```

Note that Ψ can be implemented as a function in order to ensure the equivalent behavior even in the context of program execution. Assume that **Psi** has the formal arguments A, i, e and that the length of an array can be accessed via a function *length*, then the body of function Ψ is defined by Algorithm 2 (Ψ).

The introduction of the Ψ function for handling arrays does not handle correctly the semantics of arrays stated in the semantics of $\mathcal{L}$, in all cases. The semantic of $\mathcal{L}$ allows for side effects. If we have

Algorithm 2 $\Psi\ (A, i, e)$

Require: Array A, index i, value e.

Ensure: Array B that is a copy of A except for element i. Element i receives the new value e.

```
1: j = 1;
2: while j < length(A) do
3:    if j==i then
4:       B[j] = e;
5:       B[j] = A[j];
6:    end if
7:    j = j + 1;
8: end while
9: return B;
```

two variables pointing to the same arrays, then any change of the array using one of the two variables is visible using the other variable. This is not the case here, where in each array assignment the complete array content is copied. Consequently, the conversion cannot be correct in general. However, if we assume that each array is only accessed via one variable, i.e., there are no two variables pointing to the same array, then the conversion is correct. We again express this equivalence under restrictions in a corollary. In the corollary we use a function ν that maps variables of the original program to their last index used in the SSA form.

Corollary 2 *Given a loop-free program* $\Pi_{LF} \in \mathcal{L}$ *and a test suite* TS*. The SSA representation* $\Pi_{SSA} \in \mathcal{L}$ *of* Π_{LF} *is equivalent to* Π_{LF} *with respect to the test suite* TS *and the corresponding input output variables, if and only if* Π_{LF} *does not contain more than one variable pointing to the same array. I.e.,* $\forall (I,O) \in TS : \forall (y, v_y) \in [\![\Pi_{LF}]\!] I : \exists (y\_\nu(y), v_y) \in [\![\Pi_{SSA}]\!] \{(x\_0, v_x) | (x, v_x) \in I\}$.

Note that for debugging purposes the input output equivalence, which is similar to the input output conformance (IOCO) theory [110] used in testing, is sufficient. The SSA representation of the program from Fig. 3.3 is given in Fig. 3.6.

```
 1.      i_0 = 1;
 2.      min_1=input_0[0];
 3.      max_1=input_0[0];
 4.      cond_0=i_0<size_0;
 5.      cond_1=cond_0 && input_0[i_0]<min_1;
 6.      min_2=input_0[i_0];
 7.      min_3= Phi(min_1,min_2,cond_1);
 8.      cond_2=cond_0 && input_0[i_0]>max_1;
 9.      max_2=input_0[i_0];
10.      max_3= Phi(max_1,max_2,cond_2);
11.      i_1=i_0+1;
12.      cond_3=cond_0 && i_1<size_0;
13.      cond_4=cond_3 && input_0[i_1]<min_3;
14.      min_4=input_0[i_1];
15.      min_5= Phi(min_3,min_4,cond_4);
16.      cond_5=cond_3 && input_0[i_1]>max_3;
17.      max_4=input_0[i_1];
18.      max_5= Phi(max_3,max_4,cond_5);
19.      i_2=i_1+1;
20.      min_6= Phi(min_3,min_5,cond_3);
21.      max_6= Phi(max_3,max_5,cond_3);
22.      i_3= Phi(i_1,i_2,cond_3);
23.      min_7= Phi(min_1,min_6,cond_0);
24.      max_7= Phi(max_1,max_6,cond_0);
25.      i_4= Phi(i_0,i_3,cond_0);
26.      result_1=min_7*max_7;
```

Figure 3.6.: The SSA form corresponding to the program from Fig. 3.3.

3.3. Fault Localization based on a Constraint Representation

Constraints are a powerful mechanism for modeling different classes of problems from the artificial intelligence domain. Throughout the past years there was an extensive research aimed at improving the solving mechanism for constraint systems. This has brought to the market a couple of state-of-the art constraints solver: MINION ([40]), CHOCO ([19]), JaCoP ([56]), ILOG ([54]), Sugar ([106]), Mistral ([51]), ZDC ([129]) which are capable of solving large constraint systems efficiently and fast. Due to this major speed-up gains, plus the powerful modeling language, constraint solving was rapidly adopted and integrated by the industry. Starting from configuration of phone networks [101], configuration of software in an automotive system [83], configuration of services [128], recommender system [89], verification [22] and testing [41], constraints are proven to be efficient and scalable.

By applying constraint modeling for the debugging problem, we make use of the available technological advances which the constraint community has brought in the last years. This translates to the possibility to debug relatively large programs without any abstraction from their initial syntax.

A ***constraint satisfaction problem (CSP)*** is a tuple (V, D, CO), where V is a set of variables, each variable $v \in V$ has a domain $D(v)$, and CO is a set of constraints. Each constraint defines a relation between variables. A solution of a CSP assigns values to all variables s.t. all constraints are satisfied. For more details regarding CSPs we refer to [31].

A solution to a CSP can be computed by implying:

- Constraint propagation. The domains D are updated each time the variables from a constraint $C_i \in CO$ are instantiated, i.e., the variable values inconsistent with C_i are eliminated form the domain C_i.
- Backtracking or local search. For each variable we search for values in the domain that do not contradict any constraint from CO.

A CSP is said to be ***inconsistent***, when no valid instantiation of the variables exists such that all the constraints CO are satisfied. In the CSP standard theory, the constraint system is inflexible (a solution either satisfies all constraints or violates all of them) and hard, i.e., all constraints must be satisfiable. When a CSP is inconsistent there exists the possibility of relaxing the constraint system. Two situations when a CSP can be relaxed are identified:

- *Dynamic CSPs (DCSPs).* The constraint system is still under construction or it suffers from

changes due to new rules. In this situation constraints can be either removed, i.e., *relaxation*, or, inserted, i.e., *restriction*.

- *Flexible CSPs (FCSPs).* This type of CSPs have the property that whenever a complete satisfaction of all constraints is not possible, it relaxes the CSP until a solution is found. The relaxation must however respect a certain optimization criteria, e.g., the offered solution must satisfy the maximum possible number of constraints.

Modeling a problem as a constraint satisfaction problem (CSP) requires first a thorough understanding of what and how constraints are utilized. We can order constraints by the following criteria:

- ***Cardinality:*** with respect to the maximum number of variables possibly involved in one constraint.
  - *Unary Constraints:* the constraint is defined only over one variable, e.g., $equal(x,7)$, $x \in VAR$.
  - *Binary Constraints:* the constraint is defined over two variables, e.g., $lessEqual(x,y)$, $x,y \in VAR$.
  - *High-order Constraints:* the constraint is defined over three or more variables, e.g., $sum(x_0,...,x_i,res)$, $x_0,...,x_i,res \in VAR$, $i \geq 2$
- ***Constraints type:*** with respect to the allowed functionalities.
  - *Logical Constraints:* the constraint contains only logical operators, e.g., "$\vee$", "$\wedge$".
  - *Arithmetic Constraints:* the constraint contains only arithmetic operators, e.g., "$+$", "$-$".
- ***CSP type:*** with respect to its satisfiability.
  - *Standard CSP:* is about finding a solution that satisfies all constraints or prove that none exists.
  - *Max-CSP:* is about finding that instantiation of the variables which satisfies the maximum number of constraints from the CSP. In this situation not all of the constraints have to be satisfied.
  - *Weighted CSP:* is about finding that instantiation of the variables that give the minimum cost. In this situation each constraint has a certain "weight" used in computing the solution. These type of CSPs are also known as optimization problems.

- ***Constraints representation:*** with respect to the constraint tuple-value representation.
  - *Table Constraints:* Also called extensional constraints, they have the property that for each constraint, $C_i \in CO$, the allowed tuples are extensionally specified in a table. In this table the columns represent the constraint's variables (there exists a column for each variable) and the rows represent all the allowed combination of values, which satisfy the given constraint. A database can be seen as nothing more than a table constraint.
  - *Propagated Constraints:* There exists no explicit representation for the values allowed in a constraint $C_i \in CO$. The constraints are formed out of mathematical and logical relations and the variables are defined over discrete or continues domains.

According to the above classification we state that, for the programs written in the language of $\mathcal{L}$, the CSP representation is:

1. *A hard Dynamic CSP:* No relaxation is possible, but changes to the CSP are possible, e.g., adding new constraints encoding new test cases or loop-invariants.
2. *Constraints cardinality:* Unary, binary and high-order constraints are possible.
3. *Constraints type:* both arithmetic and logical.
4. *Representation:* Propagated constraints.

We first introduce some useful formal notations which we need for the subsequent definition of the debugging problem as a constraint representation:

Definition 13 ($\sigma(S)$ - Mapping original program $\leftrightarrow$ SSA)
A program *is a sequence of statements. Let* Π *be the original program in the sequential programming language L, and let* Π_{SSA} *denote the program resulting from the loop-unrolling and SSA-conversion of* Π*. Moreover, let* $\Pi^{\overline{\Phi}}_{SSA} \subseteq \Pi_{SSA}$ *be the loop-free SSA form* without *those statements which contain* Φ*. We define a (total) function* $\overline{\sigma}$ *which maps every statement* $S' \in \Pi^{\overline{\Phi}}_{SSA}$ *backwards to its corresponding statement* $S \in \Pi$*:*

$$\overline{\sigma} : \Pi^{\overline{\Phi}}_{SSA} \mapsto \Pi$$

More precisely, S' *has the general form* `v = E_expr`*, and we distinguish two cases:*

1. *if* `E_expr` *corresponds to the loop condition of a* `while`*-statement* $S_w \in \Pi$*, then* $\overline{\sigma}(S')$ *returns* S_w*.*
2. *otherwise:* `E_expr` *corresponds to the right-hand expression of an assignment statement* $S_a \in \Pi$*, and* $\overline{\sigma}(S')$ *returns* S_a*.*

We also define a (total) function for the forward mapping:

$$\sigma : \Pi \mapsto 2^{\Pi^{\overline{\Phi}}_{SSA}}$$

which returns for every $S \in \Pi$ *the set* $\sigma(S) \stackrel{def}{=} \{S' \mid \overline{\sigma}(S') = S\}$.

Example 1. Let us consider our running example program from Fig. 3.2 again. The program Π comprises 13 statements, i.e., $\langle S_1, \ldots, S_{13} \rangle$ all of them relevant for debugging. The corresponding Π_{SSA}, considering 2 iterations of the while-statement comprises 26 statements $\langle S'_1, \ldots, S'_{26} \rangle$ (see Fig. 3.6). We obtain the mapping:

$\overline{\sigma}(S'_1) = S_1, \quad \overline{\sigma}(S'_{12}) = S_4, \quad \overline{\sigma}(S'_{23}) = \phi,$
$\overline{\sigma}(S'_2) = S_2, \quad \overline{\sigma}(S'_{13}) = S_5, \quad \overline{\sigma}(S'_{24}) = \phi,$
$\overline{\sigma}(S'_3) = S_3, \quad \overline{\sigma}(S'_{14}) = S_6, \quad \overline{\sigma}(S'_{25}) = \phi,$
$\overline{\sigma}(S'_4) = S_4, \quad \overline{\sigma}(S'_{15}) = \phi, \quad \overline{\sigma}(S'_{26}) = \phi.$
$\overline{\sigma}(S'_5) = S_5, \quad \overline{\sigma}(S'_{16}) = S_8,$
$\overline{\sigma}(S'_6) = S_6, \quad \overline{\sigma}(S'_{17}) = S_9,$
$\overline{\sigma}(S'_7) = \phi, \quad \overline{\sigma}(S'_{18}) = \phi,$
$\overline{\sigma}(S'_8) = S_8, \quad \overline{\sigma}(S'_{19}) = S_{11},$
$\overline{\sigma}(S'_9) = S_9, \quad \overline{\sigma}(S'_{20}) = \phi,$
$\overline{\sigma}(S'_{10}) = \phi, \quad \overline{\sigma}(S'_{21}) = \phi,$
$\overline{\sigma}(S'_{11}) = S_{11}, \quad \overline{\sigma}(S'_{22}) = \phi,$

From this example two observations can be derived:

I. As it can be seen from the mapping rules, not all the SSA statements are mapped to a statement in the original program. The SSA-statements corresponding to the ϕ functions are actually an intermediate representation with no direct correspondent to the original program. Even though they are integrated in the debugging reasoning process this type of statement cannot be part of the conflict set.

II. The mapping of the conflict set corresponds to the original program Π and not to the unrolled version Π_{LF}. □

An important point is that the set $\Pi^{\overline{\Phi}}_{SSA}$ comprises exactly those statements in the SSA form which our approach may consider as faulty, whereas the Φ-statements cannot be faulty. As a statement in the original program may correspond to several statements in the SSA form, a single fault in the original

program may lead to multiple faulty statements in the SSA form. Considering the original program, our approach is able to deal (a) with faults in the boolean expressions of loop conditions and (b) with faults in the expressions on the right hand side of assignment statements.

We now have the prerequisites needed for defining the constraint representation of a sequential program.

Definition 14 (CON_Π - Constraint representation of Π) *The constraint representation CON_Π of a sequential program Π is a tuple (V, D, CO) with:*

- $V = VAR\big(\Pi_{SSA}\big) \cup \{ab(S) \mid S \in \Pi\}$ *where*
  - $VAR\big(\Pi_{SSA}\big)$ *is the set of all variables in* Π_{SSA}.
  - $ab(S)$ *denotes a single boolean variable stating whether statement S of the original program is abnormal (i.e., faulty).*
- *the domain $D(v)$ of a variable $v \in VAR\big(\Pi_{SSA}\big)$ is equivalent to the datatype of the variable in the program, and $D\big(ab(S)\big) = \{true, false\}$.*
- *CO comprises exactly one constraint for every statement in the SSA form. CO is created as follows:*
  - *For every $S' \in \Pi_{SSA}$:*
    1. *if S' has the form* `v = Φ(...)`*: add the relation*

$$v = \Phi(\ldots)$$

to CO.

    2. *otherwise, S' has the form* `v = E_expr` *and* `E_expr` *does not contain Φ: add the relation*

$$ab(S) \vee \big(v = \texttt{E}_{\texttt{expr}}\big)$$

to CO, with $S = \overline{\sigma}(S')$.

To be useful for debugging, the mapping from SSA to constraints has to be correct. This is stated in the following corollary.

Corollary 3 *Let Π_{CSP} be the constraint representation of the SSA program Π_{SSA}, and $T = (I, O)$ a test case for Π_{SSA}. Both representations, i.e., Π_{CSP} and Π_{SSA}, allow for computing the same output given the same input with respect to corresponding variables. I.e.: $\forall(y\_\nu(y), v_y) \in [\![\Pi]\!]I$: $y\_\nu(y) = v_y$ is in the solution of the CSP $\Pi_{CSP} \cup I$ when assuming all statements to be correct,.i.e., $\forall S \in STMNTS(\Pi_{SSA}) : ab(S) = false$, where $STMNTS$ is a function returning all statements of a program.*

The following constraint representation is extracted from the SSA form in Fig. 3.6:

- $V = \{min\_0, max\_0, result\_0, \ldots, cond\_0, cond\_1 max\_7, \ldots, i\_4, result\_1\} \cup \{ab(S_1), \ldots, ab(S_{12}), ab(S_8), ab(S_9), ab(S_{11}), ab(S_{13})\}$
- $D(a) = \mathbb{Z}$, $D(cond\_0) = \{true, false\}$, etc.
- constraints:

$$CO = \left\{ \begin{array}{ll}
ab(S_1) \vee (int i_0 = 1), & [S'_1] \\
ab(S_2) \vee (min_1 = input_0[0]), & [S'_2] \\
ab(S_3) \vee (max_1 = input_0[0]), & [S'_3] \\
ab(S_4) \vee (cond_0 = i_0 < size_0), & [S'_4] \\
ab(S_5) \vee \big(cond_1 = cond_0) \wedge input_0[i_0] < min_1\big), & [S'_5] \\
ab(S_6) \vee (min_2 = input_0[i_0]), & [S'_6] \\
min_3 = \Phi(min_1, min_2, cond_1), & [S'_7] \\
ab(S_8) \vee (cond_2 = cond_0 \wedge input_0[i_0] > max_1), & [S'_8] \\
ab(S_9) \vee (max_2 = input_0[i_0]), & [S'_9] \\
max_3 = \Phi(max_1, max_2, cond_2) & [S'_{10}] \\
ab(S_{11}) \vee (i_1 = i_0 + 1), & [S'_{11}] \\
ab(S_4) \vee (cond_3 = cond_0 \wedge i_1 < size_0), & [S'_{12}] \\
ab(S_5) \vee \big(cond_4 = cond_3 \wedge input_0[i_1] < min_3\big), & [S'_{13}] \\
ab(S_6) \vee (min_4 = input_0[i_1]), & [S'_{14}] \\
min_5 = \Phi(min_3, min_4, cond_4), & [S'_{15}] \\
ab(S_8) \vee (cond_5 = cond_3 \wedge input_0[i_1] > max_3), & [S'_{16}] \\
ab(S_9) \vee (max_4 = input_0[i_1]), & [S'_{17}] \\
max_5 = \Phi(max_3, max_4, cond_5) & [S'_{18}] \\
ab(S_{11}) \vee (i_2 = i_1 + 1), & [S'_{19}] \\
min_6 = \Phi(min_3, min_5, cond_3) & [S'_{20}] \\
max_6 = \Phi(max_3, max_5, cond_3) & [S'_{21}] \\
i_3 = \Phi(i_1, i_2, cond_3) & [S'_{22}] \\
min_7 = \Phi(min_1, min_6, cond_0) & [S'_{23}] \\
max_7 = \Phi(max_1, max_6, cond_0) & [S'_{24}] \\
i_4 = \Phi(i_0, i_3, cond_0), & [S'_{25}] \\
ab(S_{13}) \vee (result_1 = min_7 * max_7), & [S'_{26}]
\end{array} \right\}$$

3.3.1. Using the constraint model for debugging

From Corollaries 1 to 3 we are able to conclude that the whole conversion process is correct with respect to its input output behavior and the given restrictions. Therefore, the following theorem must be true.

Theorem 1 (Conversion) *The conversion of a program* Π *into its constraint representation does not change the computed output values for the given input values.*

Theorem 1is similar to the input-output conformance relation from testing [110].

The ***proof*** is actually straight forward: The model for the transformation process is, with respect to program's input and output, correct, i.e., the SSA form delivers for the given set of test cases the same outputs as the original debugged program Π. The statement to statement exit values are, with respect to the $\overline{\sigma}$ function, identical for both the program Π and its SSA representation. Hence, given a limited input-output environment, we can safely state that the SSA form is a correct representation of the program. The CSP is a one to one constraint statement encoding of the SSA representation. Again, for the same set of inputs the exist values of the CSP's variables are equal to the ones in the SSA form, thus the CSP representation is equivalent to the original program Π and a given test case T.

Theorem 1 is important for ensuring the correct computation of fault locations. What is missing for debugging is the connection between the test suite and the constraint representation of a program. In the following we show how test cases are represented as constraints, and, afterwards, define a solution to the debugging problem formally.

Definition 15 (Constraint representation of test cases) *Given a program* Π *and a test case* $T = (I, O)$. *The constraint representation of the test case is given as follows:* $T_{CSP} = \{x\_0 = v_x | (x, v_x) \in I\} \cup \{y\_\nu(y) = v_y | (y, v_y) \in O\}$, *where* $v\_0 \in VAR(\Pi_{SSA}))$ *and* $v\_\nu(y) \in VAR(\Pi_{SSA}))$. *The function* ν *assigns the maximum index used in the SSA form of* Π *to each variable. Basically, a test case* T *is a set of constraints which assign values to those variables in the SSA form which correspond to input and output variables in the original program (i.e, variables which represent values passed to and values returned from the program, respectively).*

From the previous discussion we know that we are interested in assignments to the *ab* variables introduced for each statement. The purpose of these variables is to state correctness (in case the

variable is false) or incorrectness of a statement. In the following, we want to use a constraint solver to compute such assignments ensuring that the system behaves consistent with respect to the given test case. Therefore, we introduce now a correctness assumption $\Gamma(\Delta)$ as follows.

Definition 16 ($\Gamma(\Delta)$) *Let $\Delta \subseteq STMNTS(\Pi)$ be a set of statements from the original program. Then $\Gamma(\Delta)$ denotes the following set of constraints:*

$$\Gamma(\Delta) = \{ab(S) = true \mid S \in \Delta\} \cup \{ab(S) = false \mid S \in STMNTS(\Pi) \setminus \Delta\}$$

Using the notation $\Gamma(\Delta)$ we are now able to formally define a solution to the debugging problem. This definition is similar to Reiter's definition of model-based diagnosis [92], but tailored to the context of program debugging:

Definition 17 (Diagnosis - constraint encoding) *Given a debugging problem (Π, TS) where $\Pi \in \mathcal{L}$ is a program, TS a test suite. Let $T \in TS$ be a failing test case. A set $\Delta \subseteq STMNTS(\Pi)$ is a solution to the model-based debugging problem (Π_{CSP}, T_{CSP}), i.e., a diagnosis, if and only if the constraint problem $\Pi_{CSP} \cup T_{CSP} \cup \Gamma(\Delta)$ is satisfiable. Given a sequential program Π with $CON_\Pi = (V, D, CO)$, a test case T, and the set of statements $\Delta \subseteq \Pi$, then Δ is a* diagnosis *wrt (Π, T) iff the constraint satisfaction problem $DIAG\_CSP(\Pi, T, \Delta)$ is satisfiable, where*

$$DIAG\_CSP(\Pi, T, \Delta) \stackrel{def}{=} (V, D, CO')$$

and

$$CO' \stackrel{def}{=} CO \cup T \cup \Gamma(\Delta)$$

A diagnosis Δ is *(subset-)minimal* iff no proper subset is a diagnosis. Moreover, a diagnosis Δ has a *minimal cardinality* iff there is no diagnosis Δ' with $|\Delta'| < |\Delta|$. In most cases, someone is interested in minimal cardinality diagnoses or even single bugs only.

We observe that, analogous to Reiter's definition of diagnosis, every superset of a diagnosis is also a diagnosis.

Moreover, note that there is always a diagnosis, given that the test case T does not contradict itself. If there is no contradiction within T, then $CO \cup T$ is satisfiable, hence $\Delta = \Pi$ is a diagnosis.

The definition of diagnosis can be immediately applied to create a simple algorithm which computes all single diagnoses for a program Π and a test case T: for every $S \in \Pi$, the algorithm generates

$DIAG\_CSP(\Pi, T, \{S\})$ and employs a constraint solver which checks whether or not this CSP has a solution. If yes, then $\Delta = \{S\}$ is a single diagnosis, otherwise Δ is not a diagnosis.

In the definition of diagnosis, the debugging problem is stated as a CSP and diagnoses are the solutions of the CSP. Because of the conversion, the diagnosis results are correct and make use of the syntax and semantics of a program for computing candidates. The level of diagnosis is the statement level. Hence, only statements can be responsible for misbehavior. Because of the used model, it is not always guaranteed to find the correct solution. The following example discusses such a situation.

Example 2. In almost all situations our approach is able to identify the faulty statements. There exists, however, situations where our algorithm cannot identify the faulty statement. If the error is induced by a missing statement we cannot isolate the root cause. We can perform diagnosis only on a "*what you see is what you get*" principle, i.e., the error must be represented in the model by a statement.

Sometimes, when the error is found at the left side of an assignment statement and the test case is too week, it is also possible to "*omit*" the error from the conflict set. For example let us again consider our running example program (Fig. 3.2). Assume furthermore that Line 13 is correct, i.e. , `result = min + max;`, but Line 6 is not (see Fig. 3.7). Instead of `min = input[i]` this line comprises `max = input[i];`.

An error revealing test case is $input = [1,0,1]$ with the expected output $result = 1$, i.e., $T = \{(I,O)|I = \{input = [1,0,1]\}, O = \{result = 1\}\}$. It can be seen that the faulty version of the program returns $result = 2$, which makes T($input = [1,0,1]$; $result = 1$) an error revealing test case. However, our approach, for this test case, cannot designate statement 6 as faulty.

The reason for this behavior is the following: The only situation in which statement 6 can be a single fault explanation is when the expressions corresponding to statements 4 and 5 evaluate to $true$. For our test case, this happens only when $input[1]$ is evaluated, i.e., $(i < length) \Leftrightarrow (1 < 3)$ (**T**) and $(input[i] < min) \Leftrightarrow (0 < 1)$ (**T**). In this situation statement 6 sets the value of max to 1. If we designate this statement, i.e., 6, as abnormal, the constraint solver tries to find a value for $max \neq 1$, that could lead to $result = min + max = 1$. After statement 2, in the faulty version of the program, min is always 1, i.e., the only natural value which max could take such that $result = 1$, is $max = 0$. If we set max to 0 or less at statement 6, then at the second iteration the value of statement 8, i.e., $if(input[i] > max)$, for $input[2] = 1$, will evaluate to true, i.e., $1 > 0 \Rightarrow true$, and max will be set again to 1, which will, again, contradict the expected value for variable $result$. Obviously, setting max at statement 6 to something

```
1.        i = 1;
2.        min = input[0];
3.        max = input[0];
4.        while (i < length) {
5.             if (input[i] < min) {
6.                  max = input[i]; // New bug, should be: min = input[i];
7.             }
8.             if (input[i] > max) {
9.                   max = input[i];
10.            }
11.            i = i + 1;
12.       }
13.       result = min + max;
```

Figure 3.7.: A program fragment computing the minimum, maximum, and sum of both for an array of integers

greater than 1 will always contradict the expected output value $result = 1$. Therefore, statement 6, for test case T($input = [1,0,1]$; $result = 1$), cannot be designated as single fault candidate. However the same error is successfully identified if we use a different test case, e.g., $T'(input = [1,2,0], result = 2)$. □

The above example shows that there are cases where the real bug cannot be correctly identified. Moreover, it also indicates that the right test case again is able to solve the problem. Hence, a close integration of test case generation into the debugging process and the handling of multiple test cases are recommended. However, even through exhaustive testing, in the case of missing statements, there exists no possibility for our algorithm to identify the error. This remains a serious limitation also for the other approaches. The only mechanism through which this type of bug can be overcome is by mutation [112], i.e., by randomly inserting statements that cause the program to no longer fail on the test suite.

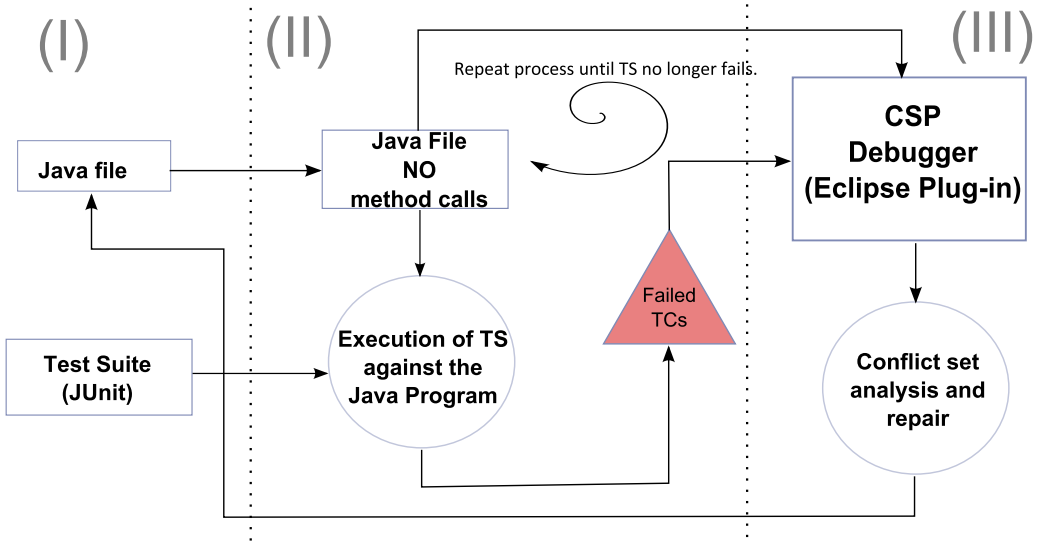

Figure 3.8.: The CSP Debugging Framework

3.4. Implementation

The semantic of $\mathcal{L}$ implies a relative simple programming language but strong enough for allowing every non OO-program to be expressed and modeled into it. In our experiments we tested our approach against a set of Java programs, which complied these language specifications. We implemented our tool both as an Eclipse [36] plug-in as well as a stand-alone unit, which we use in most of our experiments. We avoid going into details about the implementation but explain only the framework of our application and the "'know-how"' of the interaction system.

In Figure 3.8 we depict the framework of the Eclipse plug-in. The process starts with the development of the program and of a test suite, i.e., phase (I). The working environment is assured by the Eclipse-Workbench. For the Java program we use the normal Eclipse Java environment, whereas for the development of the test cases the Eclipse JUnit environment can be used. Phase (II), comprises the method call elimination from the original Java program. This step assures the transformation of the Java program into the $\mathcal{L}$ language specifications. The resulted $\mathcal{L}$-specific program is ran against the available JUnit test suite. If a failing test case is identified, the process stops and the failed test case triggers phase (III). In phase (III) the failing test case together with the $\mathcal{L}$ program is inputted to the CSP based debugger for computing the fault-explanation. Based on the computed explanations,

the user can now compute a repair suggestion. Phase (III) ends with the integration of the repair suggestion in the original program. The whole process is repeated until no other failing test case exists. Only then, can we say that the program is consistent with the given test suite. Again, passing the test suite does not assure that the program is bug-free but only that the program is conform to the provided test suite.

Remark: Every time we repeat the process, it is possible to extend the test suite at the begging of phase (I). The quality-stop condition remains always the same: the program must pass all the test cases.

The CSP-Debugger module is actually a system composed out of three components:

- SSA Converter. Uses the Abstract Syntax Tree associated to the analyzed program to build the SSA form. As input it receives the method-call free program and the number of iterations.
- CSP Converter. Converts the SSA - Program (also written in the syntax of $\mathcal{L}$) into a MINION [40] specific constraint system.
- CSP solver. The MINION constraint system is inputted to the MINION solver which computes the diagnosis and outputs it to the user for repair.

As it can be seen from the description of the debugger module, we use the MINION constraint programing language to express the CSP of the debugging problem. In what follows we briefly explain some of the key features of the MINION constraint solver. We also depict, later on, the MINION constraint representation of the debugging problem associated to the program from Fig. 3.2.

3.4.1. MINION representation

In this section we show how to apply a state-of-the-art constraint solver in program debugging. For this purpose we chose the MINION solver ([40]). We show how to model our constraint representation in MINION, and how to compute the diagnoses. MINION is an open-source constraint solver, which exhibited a superior performance on a number of large problem instances (see [40]), compared with modern constraint toolkits like ILOG or GeCode.

Although it has an expressive input language, its design goal is simplicity: it provides few options and it is free from syntactic sugar. This assures high gains with respect to performances, but an increased difficulty when expressing complex constraints.

One particularity of MINION is that it does not perform an in-between transformation of the input constraint system; i.e., the constraint solving algorithms operate directly on the input, whereas many other constraint toolkits transform the input to an internal representation. This property of MINION aims at increasing the performance, but it also imposes some limitations on the way constraints are modeled. E.g., nested quantifications, multiple operators are not supported. In order to express such constraints we need to subdivide it into two or more simple constraints. This type of representation implies the usage of small data structures for representing the information. This increases the efficiency of memory usage especially for the new computer architectures. Most of the constraint solver toolkits do not require such a formatting of the constraints but perform an hidden transformation of the constraint such that it can be brought to the same simplicity as the MINION constraints.

Example 3. Having the following arithmetic constraint:
$result = a + b - c$ in a normal constraint programming language.
Internally this constraint is represented as follows:
$temp_1 = a + b$
$temp_2 = temp_1 - c$
$result = temp_2$ □

Example 3 shows that in most constraint-solver toolkits the user has no influence on how constraints are represented. By implying the MINION representation we are free from this hidden intermediate-representation and we can optimize our CSP depending on the model specific requirements, hence speed-ups are possible.

The MINION solver offers four types of variables over which constraints can be defined:

- *BOOL* which are actually variables defined over $\{0,1\}$ domain
- *BOUND* where only the begin and end of the domain are maintained.
- *SPARSEBOUND* variables where the domain is composed out of ordered discrete variables, e.g., $\{2, 5, 68, 90\}$, again the maintenance can be done only at the domain borders
- *DISCRETE* where the maintenance is done everywhere in the domain. By domain maintenance we understand elimination of those values which are not consistent with the *CO* set.

The MINION syntax assures a simple but powerful language that among others offers support for arithmetic, logical or even extensional constraints. Additionally, different variables ordering can be imposed to speed up the search-process.

The MINION file structure is dived as follows:

- ****VARIABLE**:*** this section describes the variables of the constraint system.
- ****TUPLELIST**:*** this optional section is used for describing extensional constraints.
- ****SEARCH**:*** describes the search optimization criteria of the CSP, e.g., variable ordering.
- ****CONSTRAINTS**:*** the actual constraint system.
- ****EOF**:*** the end of the actual constraint system, everything after this token is ignored by the solver.

The SSA form of a program to be debugged comprises assignments, conditional statements, arithmetic and boolean expressions. Hence, in order to convert these programs into CSPs, we need arithmetic constraints, conditional constraints and logical constraints. The MINION constraints library provides an implementation of all arithmetic and logical operators, which are needed for our purposes. In addition, the MINION library contains a constraint of the form:

$$\texttt{reify}(Cond_{expr}, Cond_{var})$$

where the boolean variable $Cond_{var}$ is true if and only if the condition $Cond_{expr}$ is satisfied. We need this constraint for converting conditional statements.

In Sec. 3.3 we introduced a boolean variable $ab(S)$ for every statement of the original program Π. In MINION we use an array `AB[..]` containing boolean values which state the abnormality of the corresponding statement. The size of this array is equal to the number of statements involved in the diagnosis process. For example, the MINION syntax corresponding to statement S'_4 of the program given in Fig. 3.6 is:

```
BOOL cond_0
BOOL AB[1]
DISCRETE i_0 {0..250}
DISCRETE size11_0 {0..250}
...
watched-or({element(AB,4,1),
            reify(ineq(i_0,size11_0,-1),cond_0)})
```

The above constraint is satisfied if in the array AB the element found at the index 4 is 1 ($true$) or if $cond\_0 \leftrightarrow (i\_0 < size\_0)$.

The size of the domain depends on the analyzed program, e.g., in our case $\{0..250\}$ is sufficient for performing debugging.

Another challenge in the MINION conversion process is the conversion of the Φ-function. In MINION two constraints are necessary to express this function. One is of the type $cond \leftrightarrow output_{cond}$ and the other one is of the type $\neg cond \leftrightarrow output_{before}$. $output_{cond}$ represents the exit value of variable $output$ if the condition has been executed, and $output_{before}$ represents the value of the variable $output$ if the condition $cond$ was evaluated to $false$. For example, in MINION, statement S'_7 from Fig. 3.6 becomes:

```
watched-or({eq(cond_1,0), eq(min_3,min_2)})
watched-or({eq(cond_1,1), eq(min_3,min_1)})
```

The first constraint holds either if $cond\_1$ is 0 ($false$) or if the condition-exit-value of variable min, min_3 is equal to the last assignment to the variable min within the condition body, min_2. The second constraint holds when either $cond\_1$ is 1($true$) or when the condition-exit-value of variable min, min_3 is equal to the last assignment to the variable min before the condition body, min_1

Minion does not support nested quantifications or assignments with different arithmetic or boolean operands; hence we have to subdivide such a constraint into two or more simpler constraints. For example, for modeling statement S'_8 in Fig. 3.6 in MINION we need to introduce an extra auxiliary boolean variable, condAux which first evaluates $input\_0[i\_0] > max\_1$, and only after we assign to $cond\_2$ the logic-and product between condAux and cond_0. That is, in MINION we have:

```
watched-or({element(AB,8,1),
            reify(ineq(max_1,input_0[i_0],-1),condAux)})
watched-or({element(AB,8,1),
            reify(watchsumgeq([cond_0,condAux], 2),cond_2)})
```

In our approach, the elements equal to 1 of the abnormal array AB[..], designate the possible faulty components. The number of elements designates the cardinality of our diagnosis, i.e., single fault -

per CSP solution, only one component from the array is equal to one, double fault - per CSP solution two components of the array are equal to one, etc. As we want to identify all diagnosis of a certain cardinality, we are interested in all the combinations of the boolean values in the `AB[..]` array.

Every solution to a CSP is a valid instantiation of all variables such that no constraint from $|CO|$ is violated. A CSP with a great number of variables can have hundreds or thousand of solutions, e.g., CSP = {VAR = (a, b), D =((D_a:[0...10],D_b:[0...10]), D_a,$D_b \in \mathbb{N}$),$|CO| = (a < b)$ } has as valid solutions: (a = 0, b = 1), (a = 0, b = 2), ... ,(a = 9, b = 10), which sums to $C_2^{11} = 55$ possible solutions. In debugging we are, of course, not interested in all the solutions that satisfy $|CO|$. Our main focus is identifying those states that could indicate the possible explanations for the program's failure. Hence we have to focus on the abnormal array *AB* and only on those solutions that differ through the values of the *AB* array. For this purpose we use in MINION a solution search ordering based on *AB*, i.e., `VARORDER[AB[_]]` forces the solver to compute all possible solutions of the CSP with the restriction that no two solutions have the same value assignments to the `AB[..]` array; i.e., for two solutions at least one element in this array must have a different value.

Moreover, in practice it is neither possible nor desired to generate all possible diagnoses. A common approach in model-based diagnosis is to compute all (subset-)minimal diagnoses or all minimal-cardinality diagnoses. Although a single call to the MINION solver is not able to deliver all subset-minimal diagnoses, we can achieve that MINION computes all diagnoses Δ with a certain cardinality $|\Delta| = n$ in a single call. For this purpose, we introduce an auxiliary variable `sum` which is equal to the sum of the elements in the `AB[..]` array (each boolean variable has the value 0 or 1). Then we can achieve our goal by adding a constraint which specifies that `sum` must be equal to n. The following MINION code leads to the computation of all single-fault diagnoses:

```
sumleq(AB, sum)
sumgeq(AB, sum)
eq(sum, 1)
```

Because of the syntactical limitations of MINON we have to convert an assignment statement with an expression $\mathtt{E}_{\mathtt{expr}}$ on the right-hand side comprising more than one operator into a sequence of MINON statements. The idea behind the conversion is simple. A constant or variable is represented by itself. For an expression of the form $\mathtt{E}^1_{\mathtt{expr}}$ op $\mathtt{E}^2_{\mathtt{expr}}$ we convert $\mathtt{E}^1_{\mathtt{expr}}$ and $\mathtt{E}^2_{\mathtt{expr}}$ separately, and assign a new intermediate variable for each converted sub-expression. The **ComputeExpression** algorithm (Algorithm 3)implements this conversion.

Algorithm 3 ComputeExpression ($\mathtt{E}_{\mathtt{expr}}$)

Require: An expression $\mathtt{E}_{\mathtt{expr}}$ and an empty set M for storing the MINION constraints..

Ensure: A set of MINION constraints representing the expression stored in M, and a variable or constant where the result of the conversion is finally stored.

1: **if** $\mathtt{E}_{\mathtt{expr}}$ is a variable or constant **then**
2: **return** $\mathtt{E}_{\mathtt{expr}}$.
3: **else**
4: $\mathtt{E}_{\mathtt{expr}}$ is of the form $\mathtt{E}^1_{\mathtt{expr}}$ op $\mathtt{E}^2_{\mathtt{expr}}$
5: **end if**
6: Let aux_1 = **ComputeExpression** ($\mathtt{E}^1_{\mathtt{expr}}$)
7: Let aux_2 = **ComputeExpression** ($\mathtt{E}^2_{\mathtt{expr}}$)
8: Generate a new MINON variable *result* and create MINON constraints accordingly to the given operator op, which defines the relationship between aux_1, aux_2, and *result*, and add them to M.
9: **return** *result*

Example 4. The expression `a_0 + b_0 - c_0` is converted to the following MINION constraints using **ComputeExpression** where `aux1` and `aux2` represent new variables introduced during conversion.

```
sumleq([a_0,b_0],aux1)
sumgeq([a_0,b_0],aux1)
weightedsumleq([1,-1],[aux1,c_0], aux2)
weightedsumgeq([1,-1],[aux1,c_0], aux2)
```

In this example 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 used for representing the minus operator. □

For convenience we assume a function **convert** that implements the conversion of programs into MINION constraints as discussed in this section. Hence, **convert** takes the number of necessary iterations for each while-statement and the program as input and returns a set of MINION constraints as output. We use this function in the next chapter, where we discuss an algorithm for computing distinguishing test cases.

The complete MINION representation of the program from Fig. 3.6 is given in Fig. 3.9

```
          ......................
watched-or({element(ab,4,1), reify(ineq(i_0,size_0,-1 ),cond_0)})
element(input5_0, i_0,auxARRAY1)
watched-or({element(ab,5,1), reify(ineq(auxARRAY1,min_1,-1 ),cond_aux1)})
watched-or({element(ab,5,1), reify(watchsumgeq([cond_0,cond_aux1], 2),cond_1)})
watched-or({element(ab,6,1), element(input5_0,i_0,aux2)})
watched-or({element(ab,6,1), eq(min_2,aux2)})
watched-or({eq(cond_1,0), eq(min_3,min_2)})
watched-or({eq(cond_1,1), eq(min_3,min_1)})
element(input5_0, i_0,auxARRAY3)
watched-or({element(ab,7,1), reify(ineq(max_1,auxARRAY3,-1 ),cond_aux2)})
watched-or({element(ab,7,1), reify(watchsumgeq([cond_0,cond_aux2], 2),cond_2)})
watched-or({element(ab,8,1), element(input5_0,i_0,aux4)})
watched-or({element(ab,8,1), eq(max_2,aux4)})
watched-or({eq(cond_2,0), eq(max_3,max_2)})
watched-or({eq(cond_2,1), eq(max_3,max_1)})
watched-or({element(ab,9,1), sumleq([i_0,1],i_1)})
watched-or({element(ab,9,1), sumgeq([i_0,1],i_1)})
watched-or({element(ab,10,1), reify(ineq(i_1,size_0,-1 ),cond_aux3)})
watched-or({element(ab,10,1), reify(watchsumgeq([cond_0,cond_aux3], 2),cond_3)})
element(input5_0, i_1,auxARRAY5)
watched-or({element(ab,11,1), reify(ineq(auxARRAY5,min_3,-1 ),cond_aux4)})
watched-or({element(ab,11,1), reify(watchsumgeq([cond_3,cond_aux4], 2),cond_4)})
watched-or({element(ab,12,1), element(input5_0,i_1,aux6)})
watched-or({element(ab,12,1), eq(min_4,aux6)})
watched-or({eq(cond_4,0), eq(min_5,min_4)})
watched-or({eq(cond_4,1), eq(min_5,min_3)})
element(input5_0, i_1,auxARRAY7)
watched-or({element(ab,13,1), reify(ineq(max_3,auxARRAY7,-1 ),cond_aux5)})
watched-or({element(ab,13,1), reify(watchsumgeq([cond_3,cond_aux5], 2),cond_5)})
watched-or({element(ab,14,1), element(input5_0,i_1,aux8)})
```

Figure 3.9.

```
watched-or({element(ab,14,1), eq(max_4,aux8)})
watched-or({eq(cond_5,0), eq(max_5,max_4)})
watched-or({eq(cond_5,1), eq(max_5,max_3)})
watched-or({element(ab,15,1), sumleq([i_1,1],i_2)})
watched-or({element(ab,15,1), sumgeq([i_1,1],i_2)})
watched-or({eq(cond_3,0), eq(min_6,min_5)})
watched-or({eq(cond_3,1), eq(min_6,min_3)})
watched-or({eq(cond_3,0), eq(max_6,max_5)})
watched-or({eq(cond_3,1), eq(max_6,max_3)})
watched-or({eq(cond_3,0), eq(i_3,i_2)})
watched-or({eq(cond_3,1), eq(i_3,i_1)})
watched-or({eq(cond_0,0), eq(min_7,min_6)})
watched-or({eq(cond_0,1), eq(min_7,min_1)})
watched-or({eq(cond_0,0), eq(max_7,max_6)})
watched-or({eq(cond_0,1), eq(max_7,max_1)})
watched-or({eq(cond_0,0), eq(i_4,i_3)})
watched-or({eq(cond_0,1), eq(i_4,i_0)})
watched-or({element(ab,16,1), product(min_7,max_7, result_1)})
```

Figure 3.9.: The MINION representation for the program from Fig. 3.6

3.5. Analysis

The conversion process must be able to correctly map the information in both directions. From the program and test case to the CSP model, and backwards from the fault candidates set to the original program statements. This assures that the correspondence between the possible faults and the original debugging problem is valid. Also it worths nothing to the user if the debugger returns the diagnosis mapped on the SSA form of the program. Hence we update the conversion process as follows:

$$\begin{array}{ccccccc} \Pi & \xrightarrow{LR} & \Pi_{LF} & \xrightarrow{SSA} & \Pi_{SSA} & \xrightarrow{CC} & CON_{\Pi} \\ \updownarrow & & \updownarrow & & \updownarrow & & \updownarrow \\ STMNTS & \xrightarrow{\mu_{LR}} & STMNTS_{LF} & \xrightarrow{\mu_{SSA}} & STMNTS_{SSA} & \xrightarrow{\mu_{CC}} & CONSTR_{\Pi} \end{array}$$

Whereas functions $STMNTS$, $STMNTS_{LF}$, $STMNTS_{SSA}$ respectively $CONSTR_{\Pi}$ return the number of statements in Π, Π_{LF}, Π_{SSA} respectively the number of constraints in CON_{Π}.

Before analyzing the complexity of our approach, let's summarize the steps needed for computing the fault candidates associated to a debugging problem. These steps are depicted by the **CSP_Conversion& Debugging** algorithm (Algorithm 4). Hence, the complexity associated to the algorithm is the complexity of the approach.

Estimating the complexity of the *CSP_Conversion&Debugging* algorithm can be a little tricky. Its complexity is given by summing up over the complexities of steps 2 to 8, with Step 8 being the most important in deciding the complexity of the algorithm. Hence, let's start with the analysis of steps 2 to 5.

- Let N denote the size of the input (lines of code), let $\#Loop$ denote the number of loops from the program and let $\#Meth$ denote the number of methods. Each program Π has the property that its control flow graph is connected, and, with the exception of loops, is free from recursions, i.e., no recursive method calls.
- The complexity of the *CSP_Conversion&Debugging* algorithm is stepwise given by:

    1. The algorithm behind Step 2 if summarized in Algorithm 5.

Algorithm 4 CSP_Conversion&Debugging (Π, T)

Require: A program Π and a failing test case T .

Ensure: A set of MINION constraints representing the expression stored in M, and a variable or constant where the result of the conversion is finally stored.

1: Let *it* be the number of loop iterations extracted from T.
2: **Replace method calls** from $\Pi \Rightarrow \Pi \in \mathcal{L}$.
3: **Loop elimination:** each loop is unrolled *it* times in Π resulting $\Pi_{LF} \in \mathcal{L}$.
4: **SSA conversion**, $SSA(\Pi_{LF}) = \Pi_{SSA}$
5: Convert Π_{SSA} and T to their MINION CSP representations $M = \{CON_{\Pi} \cup CON_T\}$
6: Let i be 1.
7: **while** $i \leq$ number of statements in Π **do**
8: Call the **constraint solver** over M to search for solutions regarding the AB variables, where only i statements are allowed simultaneously to be incorrect.
9: **if** constraint solver returns a non-empty set of solutions **then**
10: **return** M and the set of solutions as result.
11: **else**
12: let i be $i+1$.
13: **end if**
14: **end while**
15: **return** M and the empty set as result.

Algorithm 5 Remove_MethodCalls (Π)

Require: A program Π with a cycle-free control flow graph.

Ensure: A program $\Pi \in \mathcal{L}$ free from method calls.

1: Iterate through Π and save the leaf methods, i.e., free from calls to other methods, into array $Leaf_{Meth}$.
2: **for** $i = 0...|Leaf_{Meth}|$ **do**
3: replace every call to $Meth_i$, $Meth_i \in Leaf_{Meth}$, in program Π with body of $Meth_i$.
4: delete $Meth_i$ from Π.
5: **end for**
6: Empty $Leaf_{Meth}$.
7: Iterate through Π and if there $\exists$ method calls go back to Step 1.
8: **return** Π

The class of analyzed programs has a recursions-free CFG wtr. to method calls. Hence for a finite input, the *Remove_MethodCalls* algorithm always ***terminates***. The complexity of the *Remove_MethodCalls* algorithm is $O(N^2)$, whereas N is the input size of program Π.

2. The algorithm behind Step 3 of the *CSP_Conversion&Debugging* algorithm is summarized by Algorithm 6 (is somehow similar to the algorithm from Step 1).

Algorithm 6 Remove_Loops (Π, it)

Require: The program Π resulted after applying the *Remove_MethodCall* algorithm, and a number of iteration it.

Ensure: A loop free program $\Pi_{LF} \in \mathcal{L}$.

1: Iterate through Π and save the leaf loops, i.e., free from loops in body, into array $Leaf_{loops}$.
2: **for** $i = 0...|Leaf_{loops}|$ **do**
3: Replace loop $Loop_i$, $Loop_i \in Leaf_{loops}$, in program Π, with it nested ***if*** $Loop_i$ - blocks.
4: **end for**
5: Empty $Leaf_{loop}$.
6: Iterate through Π and if there $\exists$ loops go back to Step 1.
7: **return** Π

It can be easily seen that the *Remove_Loops* algorithm, for a finite it and for a finite

program written in $\mathcal{L}$, always terminates. Its complexity is again $O(N^2)$, whereas N is the input size of program Π.

3. The conversion processes behind Steps 4 and 5 of the *CSP_Conversion&Debugging* algorithm makes use of the rules defined in Sections 3.2.2 and 3.3. As the input of both steps is finite, their corresponding conversion algorithms always terminate. For both algorithms, the complexity is $O(N)$ (a almost one to one statement conversion from $\Pi_{LF} \rightarrow \Pi_{SSA}$ and from $\Pi_{SSA} \rightarrow CON_{(\Pi,T)}$)

- The complexity of steps 2, 3, 4 and 5 is given by their associated conversion algorithms and is in all cases polynomial. The question that arises now is: are we able to state the same thing about step 8?

Step 8 of the algorithm deals with the actual solving of the debugging problem, encoded as a constraint satisfaction problem. It is a commonly known fact that a CSP can be either NP complete or *PTIME* [31]. Deciding if a CSP is solvable in polynomial time can be done by analyzing the structural properties of the CSP graph, i.e.,by hypergraph. More about CSP hypergraphs and structural properties of hyper graphs can be found in Chapter 5. In this Chapter we show that in our case we work only on the class of constraint systems where the CSP can be solved in polynomial time.

Solving the CSP is the most time consuming step of the algorithm. If the difficulty of debugging is given by the difficulty of solving the associated CSP, can we make any prediction over the debugging difficulty by analyzing different structural properties of the analyzed CSP? We try to answer this question in Chapter 5 and to analyze if this actually can correlate to an indicator for measuring the debugging complexity.

3.6. Results and conclusions

3.6.1. Experimental results

We implemented the described approach using MINION and compare it with the abstract interpretation-based model (AIM) approach, proposed by Mayer and colleagues [73]. For this purpose, we used a variation of the well-known *Traffic Alert and Collision Avoidance System (TCAS)* benchmark, taken from [94]. The TCAS benchmark comprises a set of 41 faulty versions of a correct loop-free program. The obtained results can be found in Table 3.1. The results corresponding to the AIM approach were

taken from [71]. All the experiments were carried out using an Intel Pentium Dual Core 2 GHz with 4 GB of RAM. In our experiments no out-of memory error was encountered.

It can be seen from Table 3.1, that the number of single-fault candidates computed with the MBM approach is sometimes slightly higher than the one computed using the AIM approach. However, these differences can be neglected. From the running time point of view, the MINION approach exhibits superior performances over the AIM approach although a direct comparison is somehow unfair because of different computing equipment used. In all tests the time required by MINION to compute the single-fault candidates was less than half of a second, whereas the best time needed for the AIM approach was 5 seconds and the worst one was 83 seconds, with an average of 16 seconds.

Furthermore we tested our approach on a set of small programs including the one used as running example through this chapter. The programs implement basic arithmetic functions, like division, multiplication, greatest common divisor, power, and others. The diagnosis time was always less than 1/10 seconds. The approach allowed for reducing the number of statements to be considered during debugging. It is worth noting that in almost all cases a slicing-based approach would not allow to reduce the statements to be considered even by one.

3.6.2. Conclusions

This chapter presented a detailed overview of the insides of our constraints model based approach. For the purpose of explaining model-based debugging, we introduced the syntax and semantics of a small, but Turing-complete sequential and imperative language. The language together with a test suite, which comprises test cases stating inputs and expected outputs, forms a debugging problem. We provided the background model used in model-based debugging to compute a solution to a given debugging problem. We showed that the background model can be automatically obtained from the source code. Furthermore, we proved the correctness of this conversion process. The model itself is represented as a set of constraints. When using this model together with the constraint representation of a test case, we formulate the debugging problem as a constraint satisfaction problem. Solutions to the constraint satisfaction problem can be easily obtained using a modern constraint solver. Our empirical results indicate that the approach is feasible for smaller programs up to several 1000 lines of code, from which follows that an application for automated debugging methods or functions is in reach. More results for our approach can be found in Chapters 4 and 5.

		MINION		AIM			
Variant	**#LOC$_\Pi$**	**#D**	**Time(s)**	**#D^{AIM}**		**Time(s)^{AIM}**	
				min	max	worst	best
tcas_v01	78	28	0,28	21	23	16	83
tcas_v02	78	26	0,28	12	22	11	33
tcas_v03	78	29	0,32	2	23	15	18
tcas_v04	78	25	0,26	20	23	13	16
tcas_v05	78	25	0,33	18	21	12	25
tcas_v06	78	25	0,28	19	22	15	18
tcas_v07	78	9	0,26	10	22	12	19
tcas_v08	78	27	0,36	22	22	26	26
tcas_v09	78	11	0,26	11	12	11	22
tcas_v10	78	29	0,23	21	26	16	31
tcas_v11	78	23	0,31	17	24	12	29
tcas_v12	78	23	0,21	17	23	12	37
tcas_v13	78	27	0,26	21	22	24	28
tcas_v14	78	6	0,15	6	6	5	35
tcas_v15	78	24	0,25	18	21	13	19
tcas_v16	78	26	0,29	20	22	17	47
tcas_v17	78	9	0,21	10	22	16	44
tcas_v18	78	9	0,24	10	22	13	51
tcas_v19	78	9	0,26	10	22	14	24
tcas_v20	78	27	0,28	22	23	15	29
tcas_v21	78	27	0,24	22	22	15	29
tcas_v22	78	8	0,28	9	9	10	13
tcas_v23	78	9	0,29	11	11	11	15
tcas_v24	78	24	0,24	19	21	14	18
tcas_v25	78	9	0,26	10	10	14	16
tcas_v26	78	25	0,24	18	21	16	22
tcas_v27	78	25	0,23	18	21	14	21
tcas_v28	78	14	0,23	10	22	10	65
tcas_v29	78	10	0,23	10	22	9	37
tcas_v30	78	13	0,28	12	22	11	33
tcas_v31	78	24	0,21	18	21	12	14
tcas_v32	78	23	0,28	16	19	13	18
tcas_v33	78	9	0,26	10	23	14	30
tcas_v34	78	22	0,28	18	20	13	26
tcas_v35	78	14	0,26	10	22	9	68
tcas_v36	78	2	0,24	3	3	10	13
tcas_v37	78	9	0,26	10	23	13	17
tcas_v38	78	1	0,001	3	11	13	30
tcas_v39	78	9	0,26	10	10	12	12
tcas_v40	78	8	0,23	12	12	14	16
tcas_v41	78	27	0,21	21	24	13	15

Table 3.1: Each program **Variant** is characterized by the number of statements **#LOC$_\Pi$**,the number of single fault candidates computed with the MINION approach **#D**, the time needed to compute the single fault candidate with the MINION approach **Time(s)**, the minimal and maximal number of single fault candidates obtained using the AIM approach **#D^{AIM}**, and the time needed for AIM approach to compute the worst / best diagnosis **Time(s)^{AIM}**.

Name	LOC_{Π}	#It	LOC_{ssa}	#D	T_s	|CO|	#*Varco*
Division_V0	21	1	26	4	0,01	24	22
Division_V1	21	1	26	3	0,01	24	22
Division_V2	21	1	26	2	0,01	24	22
Division_V3	21	2	32	5	0,01	33	28
Division_V4	21	2	32	5	0,01	33	28
Division_V5	21	2	32	2	0,01	33	28
Mult_V0	12	1	20	4	0,01	13	12
Mult_V1	12	1	20	4	0,01	13	12
Mult_V2	12	1	20	2	0,01	13	12
Mult_V3	12	2	25	5	0,01	21	17
Mult_V4	12	2	25	5	0,01	21	17
Mult_V5	12	2	25	2	0,01	21	17
MultV2_V0	18	1	27	6	0,01	24	20
MultV2_V1	18	1	27	6	0,01	24	20
MultV2_V2	18	1	27	6	0,01	24	20
MultV2_V3	18	2	48	6	0,01	65	49
MultV2_V4	18	2	48	5	0,01	65	49
MultV2_V5	18	2	48	8	0,01	65	49
Sum_V0	13	1	21	4	0,01	13	10
Sum_V1	13	1	21	2	0,01	13	10
Sum_V2	13	1	21	2	0,01	13	10
Sum_V3	13	2	26	5	0,01	22	16
Sum_V4	13	2	26	2	0,01	22	16
Sum_V5	13	2	26	5	0,01	22	16
gCD_V0	24	2	37	3	0,01	31	34
gCD_V1	24	2	37	4	0,01	31	34
gCD_V2	24	2	37	5	0,01	31	34
Power_V0	5	1	6	2	0,01	12	14
Power_V1	5	1	6	3	0,01	12	14
Power_V2	5	1	6	2	0,01	12	14
Power_V3	5	2	11	2	0,01	21	24
Power_V4	5	2	11	5	0,01	21	24
Power_V5	5	2	11	2	0,01	21	24
sumPower_V0	10	1	13	3	0,01	23	22
sumPower_V1	10	1	13	3	0,01	23	22
sumPower_V2	10	1	13	2	0,01	23	22
sumPower_V3	10	2	21	3	0,01	34	43
sumPower_V4	10	2	21	5	0,01	34	43
sumPower_V5	10	2	21	8	0,01	34	43
Data_V1	21	2	34	7	0,02	59	47
Data_V2	21	2	34	4	0,06	55	45
Data_V3	21	2	34	5	0,01	61	49
Data_V4	21	2	34	5	0,01	61	49
Data_V5	21	2	34	2	0,01	58	47
Data_V6	21	2	34	5	0,01	59	47
Data_V7	21	2	34	4	0,01	59	47

Table 3.2: Each program **Name** has associated its number of statements, $\mathbf{LOC_{prog}}$, the number of iteration for the loop unrolling, **#It**, the number of SSA-statements, $\mathbf{LOC_{ssa}}$, the number of diagnosis, **#D**, the MINION computation time, $\mathbf{T_s}$ and the total number of constraints and constraints variable from the MINION file, $|\mathbf{CO}|$ and $\#\mathbf{Var_{CO}}$ respectively.

Chapter 4

Extensions

"*Research is to see what everybody else has seen, and to think what nobody else has thought.*" –
Albert Szent-Gyoergi.

Applying a constraint based approach to the debugging problem assures great speed-ups when computing the fault candidates set. However, for large programs the number of fault candidates can sometimes be too large for the user to employ debugging. Hence further filtering techniques must be applied such that the fault-candidates set becomes easier to address. In this chapter we propose further improvements to our constraint based debugging approach and study their impact on the fault candidates set. Basically we discuss two major extensions to the debugging problem, both approaches being general enough to be also integrated in other MBDe engines:

- Based on integration of ***specification knowledge*** discussed in Section 4.1.
- Based on ***mutation and distinguishing test cases*** which is discussed in Section 4.2.

Both techniques are interesting and, as we will show in this chapter, successful in reducing the size of the conflict set. However each method has its drawbacks when applying it. Of course a combination of both is always preferable but not always realizable. For instance when we want to integrate specification knowledge the user must be able to correctly compute loop invariants or the pre- and post-conditions of each software block. Additionally, the computed invariant must be strong enough to be able to discriminate between the diagnosis candidates. In our experiments we tried

to compute the program invariants by implying both an out-of-the-box tool and manual computation. When we rely on the tool to compute the program invariants, for all programs, the computed invariants were too poor to reduce the conflict set. However the manual computed invariant was strong enough to substantially reduce the conflict set. The major drawback of the manual approach is that it is (obviously) not automatic and that is a very time consuming task.

This part of my thesis is based on the following papers:

- *From constraint representations of sequential code and program annotations to their use in debugging* [86]
- *On the use of Specification Knowledge in Program Debugging* [85]
- *Generating Distinguishing Tests using the MINION Constraint Solver* [123]
- *Does testing help to reduce the number of potentially faulty statement in debugging?* [68]

4.1. Integrating Specification Knowledge

Throughout the software development process the use of specifications to formally describe the software requirements is a commonly-met practice. This is especially useful when dealing with large software modules which need extensive maintenance over time. For example one important step of the software development process, where specifications are often required, is the testing phase. Many state-of-the-art testing approaches imply specifications to generate test cases for the developed software. Hence the question which now arises is: ”can we use specification knowledge also in the process of debugging?”, and if so, ”what effect will that have?”. In what follows we tackle these questions and explain how the current constraint approach can be extended such that it can integrate invariants.

In this section we focus on improving the precision of the debugging results based on integration of program specification knowledge in the form of loop invariants. Debugging which is based on a single test case often exhibits a bad discrimination among diagnosis candidates; i.e., a large number of program statements cannot be eliminated as potential fault candidates. Two important approaches for tackling this problem are the integration of multiple test cases (see [10]) and the usage of additional specification knowledge. Specifications can be provided by, e.g., assertion statements in the program, pre- and postconditions, or class and loop invariants. Previous papers suggested that the use of specification knowledge may improve the results of model-based debugging by reducing the

number of diagnoses or the number of statements considered as possibly faulty ([18], [84]). However, those works did not provide any experimental results which confirmed this claim.

In order to integrate specification knowledge into model-based debugging we rely further on the constraint representation of programs presented in Section 3.3. The constraint representation is able to express the full semantics of the specification knowledge, and it allows us to use a state-of-the-art constraint solver for computing the diagnoses. In this section we also extend the formalization of the constraint representation presented in Section 3.3 to the extent of integrating program specification knowledge in the form of loop invariants.

We also present first empirical results which compare the precision of the diagnostic results with and without specification knowledge. For this purpose, we used a set of programs which implement simple arithmetic operations, and we defined assertions like invariants, pre- and postconditions for those programs. We manually injected bugs into the programs, and we found test cases which are not satisfied by the faulty programs.

The obtained empirical results are very promising. Using specification knowledge we were able to further reduce the number of statements to be considered as possibly faulty by about 20% to 60% in the single-fault case. The average reduction was about 30%. The results are even better in the double-fault case.

4.1.1. Specifying the Knowledge

Although programmers often regard assertions as programming-language constructs which are used for fault detection at runtime, the concept of assertions has originally been introduced for the purpose of formal program specification and verification. In *Hoare logic* assertions are utilized to prove the (partial) correctness of programs, see [52]. Common types of assertions are *preconditions*, *postconditions*, and *loop invariants*. In what follows we present the mechanism behind computing the loop invariants, pre- and post-condition relying on Hoare logic [52] to formally express it.

The Hoare logic is usually used to reason about correctness of certain software. It can be seen as a contract between two clients, the one invoking the software, and the software-block itself, e.g., functions, loops. That is, we have on one side the *preconditions*, which describe the *"obligations"* of the invoker and on the other side the *postconditions* which are the *"obligations"* of the invoked software-block. If the preconditions are fulfilled then, after the execution of the software block (if it

terminates), the postconditions are true. These specifications, i.e., pre- and postconditions, are actually checking the correctness of the software block.

When the preconditions are true, ***if*** the software-block terminates and the postconditions are also true the software is said to be ***partially correct***. If additionally, the software block is guaranteed to ***always*** terminate such that for true preconditions the postconditions are always true, the software block is said to be ***totally correct***. If the software-block is invoked but the preconditions do not hold, no prediction about the behavior of the postconditions can be made, i.e., the postconditions could no longer hold. In this situation the software block is said to always be correct.

Formally a Hoare specification is a triplet of form: $\{P\}\ S\ \{Q\}$, whereas P are the preconditions, S the actual software-block and Q represents the postconditions.

Example 5. Let's presume we have the following Hoare triplet:

$\{x = 3\}$ *result* $= x * 2\ \{x > 0\}$, it can be easily seen that for the precondition $P : x = 3$ and $S = x * 2 = 3 * 2 = 6$; the postcondition always holds, i.e., $x > 0 \Longleftrightarrow 6 > 0$ which is always true. □

It can be seen that in Example 5 for the given P and S, Q can be further improved. In our case the best approximation of Q would be $Q : x == 6$. In this case Q is called the *strongest postcondition* and is formally defined as:

Definition 18 (Strongest postcondition) *A postcondition Q, $\{P\}\ S\ \{Q\}$, is called the strongest postcondition, $iff\ for \forall\ Q'$, $\{P\}\ S\ \{Q'\}$, $Q \rightarrow Q'$.*

Similarly, presume that the precondition $P : x = 3$ is replaced by $P : x > 0$. Let's presume that S is executed only for positive values of x. Then $P : x > 0$ would be the most unrestrictive possible precondition. Hence,

Definition 19 (Weakest Precondition) *A precondition P, $\{P\}\ S\ \{Q\}$ is called the weakest precondition $iff\ for \forall\ P'$, $\{P'\}\ S\ \{Q\}$, $P' \rightarrow P$.*

It can be clearly seen that in Example 5 the initial precondition, $P' : x = 3$ also implies $P : x > 0$.

Formally the two definitions from above can be also written as:

$$\frac{P' \rightarrow P,\ \{P\}\ S\ \{Q\},\ Q \rightarrow Q'}{\{P'\}\ S\ \{Q'\}}$$

After introducing some basic concepts about pre- and post-conditions, in what follows we introduce some notions about loop invariants and present how they can be formalized within the Hoare logic.

A loop invariant *INV* states a condition which must hold before and after each iteration of the loop. More precisely, the body of a `while`-loop preserves the invariant *INV*, given that the loop condition *C* holds before the execution of the body. The following rule of inference, taken from [52], formalizes this concept, and it allows us to conclude which conditions hold after the execution of the loop is finished:

$$\frac{\{C \wedge INV\}\ \texttt{body}\ \{INV\}}{\{INV\}\ \texttt{while(C) do body}\ \{\neg C \wedge INV\}}$$

This rule formalizes the following deductive inference: if the *Hoare triple* $\{C \wedge INV\}$ `body` $\{INV\}$ holds, then it can be concluded that the successful execution of `while(C) do body` leads from a state in which *INV* holds to a state in which $\neg C \wedge INV$ holds (provided that the loop terminates). A Hoare triple has the general form $\{P\}$ `stmt` $\{Q\}$: if the condition *P* holds before the execution of `stmt`, then the execution of `stmt` establishes the condition *Q* (provided that `stmt` terminates).

4.1.2. Integrating annotations

In Chapter 3 we presented a constraint based framework for debugging sequential programs with syntax and semantics similar to well-known languages like Java, but without object-oriented constructs. In what follows we extend the presented framework by integrating additional debugging information, i.e., specification knowledge.

At the moment we rely on specification knowledge given in the form of loop invariants, pre- and post-conditions, for further discriminating between the diagnosis candidates. In order to be fully automated, our framework presumes the prior existence of the loop invariants (manually computed by the user), integrated in the analyzed source code. This limitation is imposed by the fact that we cannot rely on a tool to automatically compute the loop invariants for a certain piece of code (sadly most tested tools performed poor when extracting the loop invariant).

To better illustrate our approach we make use of a small program for computing a^{exp} given in Fig. 4.1.

Before integrating annotation we proceed to converting the program into its constraint representation. Again we follow the steps presented in Chapter 3: remove loops (Fig. 4.2), conversion to the

int power(int a, int exp)

```
1.   int e = exp;
2.   int res = 1;
3.   while (e > 0) {
4.     res = res * a;
5.     e = e - 1;
     }
```

Figure 4.1.: A program for computing a^{exp}, where a and exp are integers. The variable *res* denotes the result.

SSA representation (Fig. 4.3) and last encoding the debugging problem as a constraint satisfaction problem (Fig. 4.4).

We summarize the conversion of the SSA statements to MINION constraints in Table 4.1 using some of the statements from the example given in Fig. 4.3.

SSA Statement	MINION Constraint
int e_0 = exp;	auxVar = **ComputeExpression**(exp), *eq*(e_0, auxVar)
bool cond_0 = (e_0 $>$ 0);	*reify*(*ineq*(0,e_0,-1),cond_0)
bool cond_1 = cond_0 & (e_1 $>$ 0);	*reify*(*ineq*(0,e_1,-1),cond_aux) *reify*(*watchsumgeq*([cond_0,cond_aux], 2),cond_1)
int res_4 = Φ(res_3, res_0, cond_0);	*watched-or*(*eq*(cond_0,0), eq(res_4,res_3)) *watched-or*(*eq*(cond_0,1), eq(res_4,res_0))

Table 4.1.: MINION constraints conversion

Example 6. We inject a fault in statement S_1 of the program in Fig. 4.1 by replacing `e = exp` with `e = 0`. The faulty program is depicted in Fig. 4.5. Accordingly, statement S_1' of the SSA form is changed to `int e_0 = 0`, and the corresponding constraint in CON_Π is $ab(S_1) \vee (e\_0 = 0)$.

We consider the test case $T = \{a = 2, exp = 2, res\_4 = 4\}$, which would lead to 2 loop iterations in the correct program. As the faulty program, which never executes the loop body, returns $res = 1$

int power_loopfree(int a, int exp)

```
1.   int e = exp;
2.   int res = 1;
3.   if (e > 0) {
4.     res = res * a;
5.     e = e - 1;
6.     if (e > 0) {
7.       res = res * a;
8.       e = e - 1;
       }
     }
```

Figure 4.2.: The loop-free version of the program in Fig.4.1 for 2 iterations.

int power_SSA(int a, int exp)

```
1.   int e_0 = exp;
2.   int res_0 = 1;
3.   bool cond_0 = (e_0 > 0);
4.   int res_1 = res_0 * a;
5.   int e_1 = e_0 - 1;
6.   bool cond_1 = cond_0 ∧ (e_1 > 0);
7.   int res_2 = res_1 * a;
8.   int e_2 = e_1 - 1;
9.   int res_3 = Φ(res_2, res_1, cond_1);
10.  int e_3 = Φ(e_2, e_1, cond_1);
11.  int res_4 = Φ(res_3, res_0, cond_0);
12.  int e_4 = Φ(e_3, e_0, cond_0);
```

Figure 4.3.: The loop-free SSA form of the program in Fig. 4.1 for 2 iterations. The variable `res_4` represents the output of the program (i.e., the final result).

- $V = \{a, exp, e\_0, \ldots, e\_4, res\_0, \ldots, res\_4, cond\_0, cond\_1\} \cup \{ab(S_1), \ldots, ab(S_5)\}$
- $D(a) = \mathbb{Z}$, $D(cond\_0) = \{true, false\}$, etc.
- constraints:

$$CO = \left\{ \begin{array}{ll} ab(S_1) \vee (e\_0 = exp), & [S'_1] \\ ab(S_2) \vee (res\_0 = 1), & [S'_2] \\ ab(S_3) \vee \big(cond\_0 = (e\_0 > 0)\big), & [S'_3] \\ ab(S_4) \vee (res\_1 = res\_0 * a), & [S'_4] \\ ab(S_5) \vee (e\_1 = e\_0 - 1), & [S'_5] \\ ab(S_3) \vee & \\ \big(cond\_1 = (cond\_0 \wedge (e\_1 > 0))\big), & [S'_6] \\ ab(S_4) \vee (res\_2 = res\_1 * a), & [S'_7] \\ ab(S_5) \vee (e\_2 = e\_1 - 1), & [S'_8] \\ res\_3 = \Phi(res\_2, res\_1, cond\_1), & [S'_9] \\ \ldots & \end{array} \right\}$$

Figure 4.4.: The CSP representation of the program in Fig. 4.3.

($res\_4 = 1$ in the SSA form, respectively), this test case fails, and so the empty set $\{\}$ is not a diagnosis. The algorithm for finding all single-fault diagnoses, which is described above, yields 3 diagnoses: $\Delta_1 = \{S_1\}, \Delta_2 = \{S_2\}, \Delta_3 = \{S_3\}$.

E.g., when the algorithm checks whether the candidate $\Delta_2 = \{S_2\}$ is a diagnosis, then we have $\Gamma(\{S_2\}) = \{ab(S_1) = false, ab(S_2) = true, ab(S_3) = false, \ldots\}$, and the constraint solver can determine that the constraint system $CO \cup T \cup \Gamma(\{S_2\})$ has a solution which assigns the value $res\_0 = 4$. Intuitively, S_2 is a single diagnosis because if we replaced S_2 in the faulty program (Fig. 4.5) by `int res = 4`, then the resulting program would satisfy the test case T. Moreover, S_3 is also a diagnosis, because the constraint system has a solution with the assignment $cond\_0 = true$ and $cond\_1 = true$. In other words, S_3 is a diagnosis because the loop condition could be changed s.t. the resulting program has 2 loop iterations for the given test case.

□

int power_faulty (int a, int exp)

```
1.   int e = 0;
2.   int res = 1;
3.   while (e > 0) {
4.     res = res * a;
5.     e = e - 1;
     }
```

Figure 4.5.: A faulty program for computing a^{exp}. It is almost equal to the (correct) program in Fig. 4.1, but statement S_1 was changed from `e = exp` to `e = 0`.

4.1.3. Improving the Diagnostic Precision by Integrating Specification Knowledge

In the diagnosis example above we obtained 3 single-fault diagnoses, i.e., less than half of the statements in Fig. 4.5 could be eliminated as potential single-faults. It is obvious that automated debugging should strive for a higher diagnostic precision in order to be useful in practice. The low precision in the example above results from the fact that the correct program behavior is only specified by a single test case, and that a test case is a *black-box* specification, meaning that it defines only the input-output behavior of a program, but does not specify expected behavior inside the program.

In this section we tackle this problem by integrating *assertions* and empirically investigating their benefit. With assertions we denote specifications, which state conditions that must hold at specific locations within the program. Many programming languages, like C or Java, have `assert(cond)`-statements, which can be placed at arbitrary locations in the source code and which are checked at runtime. In recent years the use of assertions has gained wide acceptance among software developers, which has also been fostered by growing tool support. Assertions are also closely related to Design by Contract, see [77].

Our approach presumes that the loop invariant is correct, i.e., it specifies the desired behavior of a loop, and a correct implementation of the loop must conform to this specification. For a given program, it can be proven by induction over loop iterations that the program conforms to the loop invariant; i.e., it has to be shown that the loop body preserves the invariant, provided that the loop condition holds. However, if the program contains a bug, then the (correct) loop invariant may contradict the faulty program. Hence, the intention is that a loop invariant is defined *before* the loop's body is implemented, because the written code may be faulty, and deriving a loop invariant from a faulty

int power_faulty_annot(int a, int exp)

$\| \mathit{PRE} : \texttt{exp} \geq 0 \|$

```
1.   int e = 0;
2.   int res = 1;
3.   while (e > 0) {
```

$\quad \| \mathit{INV} : (\texttt{res} == \texttt{a}^{\texttt{exp}-\texttt{e}}) \wedge (\texttt{e} \geq 0) \|$

```
4.     res = res * a;
5.     e = e - 1;
     }
```

$\| \mathit{POST} : \texttt{res} == \texttt{a}^{\texttt{exp}} \|$

Figure 4.6.: The faulty program from Fig. 4.5 annotated with three assertions: the program's pre- and postcondition and a loop invariant.

program can lead to an invariant which does not express the desired behavior and which is useless for debugging.

Also note that the invariant of a given loop is ambiguous because a loop invariant does not need to state *all* conditions which are preserved by the loop body. If INV_1 and INV_2 are invariants of a given loop, we say that INV_1 is *stronger* than INV_2 iff $\mathit{INV}_1 \models \mathit{INV}_2$ and $\mathit{INV}_2 \not\models \mathit{INV}_1$. For debugging purposes we desire invariants which are as strong as possible in order to improve the diagnostic precision. However, in practice it is often not possible to find the strongest invariant, but our experience has shown that even a "weak" invariant can often be useful to eliminate diagnosis candidates.

Example 7. Figure 4.6 depicts the faulty example program annotated with assertions. It can be seen that the chosen invariant is strong enough to prove that the postcondition *POST* holds after the loop, provided that the loop is correctly implemented and that it terminates. More precisely, $\neg C \wedge \mathit{INV} \models \mathit{POST}$ holds, where C is the loop condition $(e > 0)$, because $\neg C \wedge \mathit{INV} \models (e == 0)$, and so $\neg C \wedge \mathit{INV} \models (res == a^{exp})$. □

Fig. 4.7 sketches a schematic depiction of a loop invariant: it can be seen that the invariant must hold before the first iteration of the loop and after each executed iteration. The formulation of a debugging problem as a CSP allows for the usage of the full semantics of assertions. The integration of assertions in the loop-free SSA form is done as follows. Every pre- or postcondition P of a program

loop with invariant:

```
assert(INV);        [A1]
while(C) {
  ...  [ body ]
  assert(INV);      [A2]
}
```

Figure 4.7.: General schema showing how a loop invariant can be represented by two `assert(cond)`-statements.

is directly mapped to a single SSA-statement of the form

$$\texttt{assert}(\texttt{P}_{\texttt{expr}});$$

Moreover, for a loop invariant *INV* the loop-free SSA form contains $n+1$ additional `assert`-statements, where n is the number of loop iterations considered in the SSA form: regarding the general schema in Fig. 4.7, the assertion $\mathcal{A}_1$ is mapped to the SSA form

$$\texttt{assert}(\texttt{INV\_0}_{\texttt{expr}});$$

where $\texttt{INV\_0}_{\texttt{expr}}$ represents the invariant *INV* referring to the variable values *before* the first loop iteration. $\mathcal{A}_2$ is mapped to

$$\texttt{assert}\big((\texttt{cond\_i} = \texttt{false}) \vee \texttt{INV}_{\texttt{i+1,expr}}\big);$$

where variable `cond_i` corresponds to the loop condition after i iterations (i.e., if `cond_i` is *true*, then the $(i+1)^{th}$ loop iteration is executed), and $\texttt{INV}_{\texttt{i+1,expr}}$ refers to the variable values *after* the $(i+1)^{th}$ loop iteration. The SSA form for the example program enhanced by assertions is shown in Fig. 4.8.

The following definition deals with the integration of assertions into the constraint representation:

Definition 20 (Integration of assertions into CON_Π)
The constraint representation CON_Π, which was defined in Def. 14, is extended as follows. The set of constraints CO comprises exactly one constraint for every statement in the SSA form, including `assert`*-statements. If a statement $S' \in \Pi_{SSA}$ of the SSA form has the form* `assert(`$\texttt{cond}_{\texttt{expr}}$`)`*, then add the relation* $\texttt{cond}_{\texttt{expr}}$ *to CO. Otherwise, translate S' as defined in Def. 14.*

int power_SSA_annot(int a, int exp)

```
...  ...
13.  assert(exp ≥ 0);
14.  assert((res_0 == a^(exp−e_0)) ∧ (e_0 ≥ 0))
15.  assert((cond_0 == false)
             ∨ (res_1 == a^(exp−e_1)) ∧ (e_1 ≥ 0))
16.  assert((cond_1 == false)
             ∨ (res_2 == a^(exp−e_2)) ∧ (e_2 ≥ 0))
17.  assert(res_4 == a^exp);
```

Figure 4.8.: The loop-free SSA form, from Fig. 3.6 enhanced with the assertions from Fig. 4.6.

Example 8. Integrating the assertions from Fig. 4.8 adds the following constraints to CO:

$$CO = \left\{ \begin{array}{ll} \ldots & \ldots \\ exp \geq 0, & [S'_{13}] \\ (res\_0 = a^{exp-e\_0}) \wedge (e\_0 \geq 0), & [S'_{14}] \\ \ldots & \ldots \end{array} \right\}$$

□

In the following example we demonstrate the benefit of assertions for improving the diagnostic precision:

Example 9. As in the diagnosis example from Sec. 3.3, we use the test case $T = \{a = 2, exp = 2, res\_4 = 4\}$, which led without the use of assertions to the single-fault diagnoses $\Delta_1 = \{S_1\}, \Delta_2 = \{S_2\}, \Delta_3 = \{S_3\}$. Remember that S_3 was a diagnosis because the constraint system $CO \cup T \cup \Gamma(\{S_3\})$ has a solution with the assignment $cond\_0 = true$ and $cond\_1 = true$. However, if we integrate the loop invariant as given in Fig. 4.6 into CO, then this constraint system has no solution: if $cond\_0 = true$, then the invariant enforces the condition $e\_1 \geq 0$ (see statement S'_{15} in Fig. 4.8), which contradicts the constraints $e\_0 = 0$ and $e\_1 = e\_0 - 1$ (which correspond to statements S_1 and S_5 in the faulty program in Fig. 4.6). Hence, the candidate $\{S_3\}$ is no longer a diagnosis, and only two single-fault diagnoses remain (Δ_1, Δ_2).

Also note that the invariant $INV_w : (\mathtt{e} \geq 0)$, which is weaker than the complete invariant as defined in Fig. 4.6, was here sufficient to eliminate $\{S_3\}$ as diagnosis candidate. □

4.1.4. Experimental Results

We evaluated our debugging approach on a set of small Java programs which have the property that they contain at least one `while`-loop. We annotated these programs with pre- and postconditions and with loop invariants. We investigated the benefit of assertions by performing the diagnosis for the annotation-free programs as well as for the annotated programs and by comparing the diagnostic results afterwards.

For this purpose we performed the following steps:

1. In every program we injected different single- and double-faults (see below).
2. For each faulty program we created two loop-free SSA representations: one with and one without assertions.
3. We converted the resulting programs into a constraint system in MINION syntax.
4. For each program we defined a test case which leads to either 1 or 2 loop iterations. Based on this test case, we computed the diagnoses using the MINION implementation provided by [80].

In the first set of experiments we injected 3 different single faults in every program; i.e., there were three different faulty versions for each program. The experimental results are given in Tbl. 4.2. Each fault changes either the right-hand side of an assignment statement or the boolean expression of a loop condition. In Tbl. 4.2 we use identifiers to denote each fault (column **Er**). For example, in the program `power(a, exp)` (see Fig. 4.1) we injected the faults P1-P3, where P1 changes line 2 to `int res = 0`, P2 changes line 1 to `int e = exp`, and P3 changes line 4 to `res = res + a`.

When comparing the columns **#D** and $\mathbf{\#D_{inv}}$ in Tbl. 4.2, it can be clearly seen that the integration of loop invariants significantly increases the diagnostic precision: on average, only 1.8 single-fault diagnoses remain after integrating the assertions. Moreover, we defined the metrics $E_{\%}$ and $E_{\%,inv}$ which denote the percentage of statements which are eliminated as potential faults by our debugging approach. In case of single-faults they can be simply computed as follows:

$$E_{\%} \stackrel{\text{def}}{=} 100 * \frac{L_{\Pi} - \#D}{L_{\Pi}}; E_{\%,inv} \stackrel{\text{def}}{=} 100 * \frac{L_{\Pi} - \#D_{inv}}{L_{\Pi}}$$

As average value of all experiments we obtained $avg(E_{\%,inv} - E_{\%}) = 30.6\%$; i.e., the integration of assertions significantly improved the diagnostic precision by further eliminating 30.6% of the statements as fault candidates.

In all experiments it took less than $0.1\,sec$ to compute all solutions on a PC with a Pentium 4 2.0 GHz CPU.

In the second set of experiments we injected a double fault in each program (one fault before the loop and another one in the loop condition). The results are given in Tbl. 4.3. A remarkable finding is that without using assertions we obtained several single-fault diagnoses for every program, although the programs contained two faulty statements. The integration of loop invariants greatly improved the diagnostic results: in all of those experiments, the invariant was able to eliminate the single-fault diagnoses, and the number of double-fault diagnoses was significantly smaller. The integration of assertions improved the diagnostic precision by further eliminating $avg(E^2_{\%,inv} - E^2_{\%}) = 31.9\%$ of the statements as fault candidates.

4.2. Mutation Based Debugging

Computing assertions can sometimes prove to be a difficult task, e.g., in software maintenance, when the software maintainer is other than the code's owner. This limits the approach only to programs (methods) were the user has enough information over the source code to the extent that it can extract the specification knowledge. Another drawback of the approach is that the computed invariant cannot guarantee to further discriminate over the conflict set.

In what follows, we propose an algorithm for further reducing the size of the conflict set corresponding to a given debugging problem. In order to discriminate between the possible faults, the proposed algorithm relies on the concept of distinguishing test cases and program mutation. An advantage of this approach is the fact that it can be combined with any existent conflict-based debugger.

Additionally to the existent program and test suite, our proposed algorithm requires the existence of a prior computed conflict set. Contrary to this, the specification based approach, discriminates between components during the debugging reasoning process and is restricted only to the framework proposed in Chapter 3. The distinguishing test cases approach reasons at the end of the debugging process, i.e., over the conflict set, and by implying mutation-based techniques tries to reduce the conflict set or, in some situations, even repair the existing bug. Further details about mutation will be given later on in this chapter. We present first the distinguishing test case approach which is at the foundation of this approach.

No	Name	$\mathbf{L}_\Pi$	#It	$\mathbf{L}_{\Pi_{SSA}}$	Er	#D	$\mathbf{E}_\%$	$\#\mathbf{D}_{\mathbf{inv}}$	$\mathbf{E}_{\%,\mathbf{inv}}$	$\mathbf{E}_{\%,\mathbf{inv}} - \mathbf{E}_\%$
1.	Division	8	1	12	D1	4	50.0 %	2	75.0 %	25.0 %
2.	Division	8	1	12	D2	3	62.5 %	1	87.5 %	25.0 %
3.	Division	8	1	12	D3	2	75.0 %	2	75.0 %	0.0 %
4.	Division	8	2	18	D1	5	37.5 %	2	75.0 %	37.5 %
5.	Division	8	2	18	D2	5	37.5 %	2	75.0 %	37.5 %
6.	Division	8	2	18	D3	2	75.0 %	2	75.0 %	0.0 %
7.	Mult	5	1	9	M1	4	20.0 %	1	80.0 %	60.0 %
8.	Mult	5	1	9	M2	4	20.0 %	2	60.0 %	40.0 %
9.	Mult	5	1	9	M3	2	60.0 %	1	80.0 %	20.0 %
10.	Mult	5	2	15	M1	5	0.0 %	1	80.0 %	80.0 %
11.	Mult	5	2	15	M2	5	0.0 %	2	60.0 %	60.0 %
12.	Mult	5	2	15	M3	2	60.0 %	1	80.0 %	20.0 %
13.	MultV2	8	1	16	M21	6	25.0 %	2	75.0 %	50.0 %
14.	MultV2	8	1	16	M22	6	25.0 %	4	50.0 %	25.0 %
15.	MultV2	8	1	16	M23	6	25.0 %	3	62.5 %	37.5 %
16.	MultV2	8	2	41	M24	6	25.0 %	1	87.5 %	52.5 %
17.	MultV2	8	2	41	M25	5	37.5 %	1	87.5 %	50.0 %
18.	MultV2	8	2	41	M23	8	0.0 %	3	62.5 %	62.5 %
19.	Sum	5	1	8	S1	4	20.0 %	1	80.0 %	60.0 %
20.	Sum	5	1	8	S2	2	60.0 %	1	80.0 %	20.0 %
21.	Sum	5	1	8	S3	2	60.0 %	1	80.0 %	20.0 %
22.	Sum	5	2	14	S2	5	0.0 %	1	80.0 %	80.0 %
23.	Sum	5	2	14	S3	2	60.0 %	1	80.0 %	20.0 %
24.	Sum	5	2	14	S4	5	0.0 %	2	60.0 %	60.0 %
25.	gCD	9	2	22	G1	3	77.7 %	1	88.8 %	11.1 %
26.	gCD	9	2	22	G2	4	55.5 %	4	55.5 %	0.0 %
27.	gCD	9	2	22	G3	5	44.4 %	4	55.5 %	11.1 %
28.	Power	5	1	6	P1	2	60.0 %	1	80.0 %	20.0 %
29.	Power	5	1	6	P2	3	40.0 %	2	60.0 %	20.0 %
30.	Power	5	1	6	P3	2	60.0 %	1	80.0 %	20.0 %
31.	Power	5	2	11	P1	2	60.0 %	1	80.0 %	20.0 %
32.	Power	5	2	11	P4	5	0.0 %	2	60.0 %	60.0 %
33.	Power	5	2	11	P3	2	60.0 %	1	80.0 %	20.0 %
34.	sumPower	10	1	13	SP1	3	70.0 %	1	90.0 %	20.0 %
35.	sumPower	10	1	13	SP2	3	70.0 %	2	80.0 %	10.0 %
36.	sumPower	10	1	13	SP3	2	80.0 %	2	80.0 %	0.0 %
37.	sumPower	10	2	21	SP1	3	70.0 %	1	90.0 %	20.0 %
38.	sumPower	10	2	21	SP2	5	50.0 %	2	80.0 %	30.0 %
39.	sumPower	10	2	21	SP3	8	20.0 %	6	40.0 %	20.0 %
					average:	**3.9**	**42.4 %**	**1.8**	**73.0 %**	**30.6 %**

Table 4.2.: Experimental results for the first set of experiments. For each program **Name**, $\mathbf{L}_\Pi$ is the number of statements in the original program, **#It** denotes the number of considered loop iterations, $\mathbf{L}_{\Pi_{SSA}}$ denotes the number of statements in the loop-free SSA form, **Er** identifies the injected fault, **#D** is the number of single-fault diagnoses obtained without loop invariants and $\#\mathbf{D}_{\mathbf{inv}}$ is the number of single-fault diagnoses when using loop invariants. Moreover, $\mathbf{E}_\%$ and $\mathbf{E}_{\%,\mathbf{inv}}$ denote the percentage of statements of the original program which do not occur in any single-fault diagnosis without assertions ($E_\%$) or when using assertions ($E_{\%,inv}$); see the text for details. Hence, the last column $\mathbf{E}_{\%,\mathbf{inv}} - \mathbf{E}_\%$ states the percentage of statements which can be eliminated as single-fault candidates when integrating assertions.

Name	L_Π	#It	$L_{\Pi_{SSA}}$	Er	$\#D^2$	$E^2_\%$	$\#D^1$	$\#D^2_{inv}$	$E^2_{\%inv}$	$\#D^1_{inv}$	$E^2_{\%inv} - E^2_\%$
Division	8	2	18	D1 + D2	19	0.0 %	4	4	50.0 %	0	50.0 %
Mult	5	2	15	M1 + M2	7	0.0 %	2	3	40.0 %	0	40.0 %
MultV2	8	2	41	M21 + M22	25	0.0 %	5	7	0.0 %	0	0.0 %
Sum	5	2	14	S1 + S2	7	0.0 %	2	2	40.0 %	0	40.0 %
gCD	9	2	22	G1 + G2	21	0.0 %	3	6	33.0 %	0	33.8 %
Power	5	2	11	P1 + P4	7	0.0 %	2	3	20.0 %	0	20.0 %
sumPower	10	2	21	SP1 + SP2	27	0.0 %	3	5	40.0 %	0	40.0 %
				average:	**15.7**	**0.0 %**	**2.9**	**4.3**	**31.9 %**	**0**	**31.9 %**

Table 4.3.: Experimental results for the double diagnosis and specification knowledge. $\#\mathbf{D^1}$ and $\#\mathbf{D^2}$ denote the number of diagnoses with cardinality 1 and 2, respectively, when no loop invariants are used, whereas $\#\mathbf{D^1_{inv}}$ and $\#\mathbf{D^2_{inv}}$ state the number of diagnoses when using loop invariants. $\mathbf{E^2_\%}$ and $\mathbf{E^2_{\%inv}}$ denote the percentage of statements of the original program which do not occur in any double-fault diagnosis without assertions ($\mathbf{E^2_\%}$) or when using assertions ($\mathbf{E^2_{\%inv}}$). The column $\mathbf{E^2_{\%inv} - E^2_\%}$ states the percentage of statements which can be eliminated as candidates when integrating assertions.

4.2.1. Generating Distinguishing Test Cases

Constraints have been used for various purposes like verification [22], debugging [18, 122], program understanding [115] as well as testing [32, 41, 42]. Some of the proposed techniques use constraints to state specification knowledge like pre- and post-conditions. Others use constraints for modeling purposes. In this section we rely on the latter and use constraints obtained from the program directly. In contrast to previous research we focus on generating test cases that can be used for distinguishing between different implementations. A test case distinguishes between two implementations if it reveals a different output behavior using the same inputs for both implementations. Of course such a distinguishing test case might not always exist. Moreover, we assume that the implementations behave deterministically. Otherwise, it is not guaranteed that a given input always generates the same outputs.

There are many potential applications of distinguishing test cases. The first application scenario is debugging. In debugging we might obtain too many diagnosis candidates, i.e., parts of the program that explain a detected misbehavior. In order to reduce the size of the diagnosis candidates set, for

each fault candidate we compute a set of mutants as possible explanation for the faulty statements. The distinguishing test case generator together with an oracle helps to discriminate between two possible repairs. The other scenario is test case generation based on program mutation. In such an application mutants for a given program are generated. The distinguishing test case generator is used to compute test cases for each mutant and the original program.

The idea behind our approach is to convert two implementations into constraints and to represent the problem of generating distinguishing test cases as a constraint satisfaction problem. In order to convert a program into a constraint system we make use the approach described in Chapter 3, i.e., we first remove the loops, replacing them with a bounded sequence of conditionals; we then compile the resulting program into its static single assignment form, from which we directly compute the constraints. For the constraint representation and the solving, we rely on the MINION constraint solver [40, 80].

Before discussing our technique for generating distinguishing test cases in detail, we outline the underlying ideas on a small example program. More details are given in the rest of this section.

```
1.    begin
2.         i = 2 * x;
3.         j = 2 * y;
4.         o1 = i + j;
5.         o2 = i * i;
6.    end;
```

This program can be easily converted into a constraint representation using the constraint language from MINION. We only need to convert the program statement by statement. For representing constraints we use a relational notation. For example, the multiplication $x * y = z$ is represented by `product(x,y,z)` and for the sum $x + y = z$ we use the two relations `sumleq([x,y],z)` and `sumgeq([x,y],z)` stating $x + y \leq z$ and $x + y \geq z$ respectively. Hence, the constraint representation of our program is the following:

```
product(2,x,i)
product(2,y,j)
sumleq([i,j],o1)
```

```
sumgeq([i,j],o1)
product(i,i,o2)
```

Now consider a variant of the program where Line 3 is changed to `j = 3*y` and let us again convert it into its constraint representation. Note that in this case we added a post-fix string "_v" to each variable to distinguish the variables of the original program from the variables of the variant.

```
product(2,x_v,i_v)
product(3,y_v,j_v)
sumleq([i_v,j_v],o1_v)
sumgeq([i_v,j_v],o1_v)
product(i_v,i_v,o2_v)
```

Informally speaking, a distinguishing test case is a test case for separating the behavior of two programs where the input values for each program is the same but the computed output is not. Hence, we have to state that the inputs are the same and that there exists at least one output where the computed values are not equivalent. Using the MINION constraint `eq` for stating equivalence, `diseq` for stating that two variables have different values, and the logical constraint `watched-or` for formalizing a disjunction, we give the constraints necessary to obtain a distinguishing test case in our example:

```
eq(x,x_v)
eq(y,y_v)
watched-or({diseq(o1,o1_v),diseq(o2,o2_v)})
```

A solution for the given constraints is also a distinguishing test case. Using MINION as constraint solver we are able to compute more than one distinguishing test cases for this example, e.g., one solution is `x`=2, `y`=2 and there are many others. All solutions have in common that `y` is not equal to 0.

The rest of this section is organized as follows. Next we give all necessary definitions and discuss the preliminaries of our approach. Afterwards, we outline the algorithm for computing distinguishing test cases and present first empirical results. Finally, we discuss related research and conclude.

Computing distinguishing test cases

We assume that the program $\Pi \in \mathcal{L}$ to be compiled into a constraint representation is deterministic and written in an imperative assignment language with the usual kinds of statements, e.g., variable assignments, conditional statements, and loops. The underlying type system of the language comprises basic datatypes like Booleans, integers, floating point numbers and arrays. Each variable stores a value of the corresponding datatype. The values of variables are stored in a variable environment. A variable environment (or environment for short) is a function mapping variables to their values. We further assume that each program Π has some input variables and output variables. We use $[\![\Pi]\!](I)$ to denote execution of Π on a specific input environment (or input for short) I. The result of the execution is always an environment, i.e., the output environment. In the following we also represent environments as set of tuples (x, v) where x is a variable and v is a value.

According to Definition 6, a test case is a tuple (I, O) where I is the input environment denoting the given values of input variables, and O is the output environment where the expected values of the output variables are specified. Note that, regarding the definition, it is also possible that O is empty. A program Π is passing a test case (I, O) if and only if the execution of Π on I returns the expected output values specified in O. Formally, we define passing and failing test cases as follows:

$$[\![\Pi]\!](I) \supseteq O \Leftrightarrow \Pi \text{ passes test case}(I, O)$$
$$\neg(\Pi \text{ passes test case}(I, O)) \Leftrightarrow \Pi \text{ fails test case}(I, O)$$

Note that not all values have to be specified. However, it is necessary that all given values are returned as expected. A variable where no value is specified in O can have an arbitrary value after program execution.

Definition 21 (Distinguishing test case) *Given programs* Π *and* Π'. *A test case* $(I, \emptyset)$ *is a distinguishing test case if and only if there is at least one output variable where the value computed when executing* Π *is different from the value computed when executing* Π' *on the same input* I.

$$(I, \emptyset) \textit{ is distinguishing } \Pi \textit{ from } \Pi' \Leftrightarrow$$
$$\exists x : (x, v) \in [\![\Pi]\!](I) \;\;\&\;\; (x, v') \in [\![\Pi']\!](I) \;\;\&\;\; v \neq v'$$

We call the problem of finding an input environment that distinguishes two programs Π and Π' the *distinguishing test case problem*. It is worth noting that a distinguishing test case is according to

our definitions always a passing test case, because the output environment is not specified. From the distinguishing test case we are always able to derive a test case with a specified expected output. This can be done manually or in some cases automatically. The latter is used to compute test cases from the mutations of a given program. In this case, the program is assumed to be correct and the output of the execution of the program is therefore the expected output of the test case. When searching for distinguishing test cases for all mutants we finally receive a test suite that can be used to separate the original program from all its mutations.

In order to compute distinguishing test cases for two programs Π_1 and Π_2 we have to ensure that the inputs for both programs are the same whereas the computed outputs are different. This idea can be easily represented in MINION. We only have to add the corresponding constraints to the converted programs. Moreover, we have to ensure that the converted programs use different names for the variables. Hence, we have to rename the variables in the constraint representation before putting them together. Algorithm **computeDistinguishingTC** (Algorithm 7) depicts the process of computing the distinguishing test case of two programs.

Algorithm 7 computeDistinguishingTC(Π_1, Π_2, #*It*)

Require: Two programs Π_1 and Π_2 having the same input variables (*IN*) and output variables (*OUT*), and a maximum number of iterations #*It*.

Ensure: A distinguishing test case.

1: Call **convert**(Π_1,#*It*) and store the result in M_1.
2: Call **convert**(Π_2,#*It*) and store the result in M_2.
3: Rename all variables x used in constraints M_1 to x_P1.
4: Rename all variables x used in constraints M_2 to x_P2.
5: Let M be $M_1 \cup M_2$.
6: **for all** input variables $x \in IN$ **do**
7: Add the constraint `eq(`x`_P1, `x`_P2)` to M.
8: **end for**
9: **for all** output variables $x \in OUT$ **do**
10: Add the constraint `diseq(`x`_P1, `x`_P2)` to M.
11: **end for**
12: **return** the values of the input variables obtained when calling the MINION constraint solver on M as result.

The **computeDistinguishingTC** algorithm obviously terminates. The given programs and sets are all finite and the conversion terminates. Moreover, the constraint solver also terminates after checking all possible solutions when considering only finite domains. The computational complexity is mainly determined by the constraint solver. The conversion itself is polynomial in the size of the programs. Finding a solution for a finite domain is exponential in the number of used variables.

Note that, the whole approach is not necessary restricted to the MINION constraint solver. All discussed steps can be adapted to other constraint solvers. In the following section, we present first empirical results of the approach using MINION.

Experimental results

We implemented the discussed approach in Java and applied it on some small Java programs ignoring object-oriented features. For each program we have the original bug-free version, and a set of four mutants obtained by manually injecting different single-faults into the original program. Each program comprises at least one loop structure. We generate the discriminating test cases, i.e., kill the mutants, considering 2, 4 and 7 iterations of each loop statement. We present the obtained results in Table 4.4. All the experiments were performed using an Intel Pentium Dual Core 2 GHz computer with 4 GB RAM. We imposed a limit of two hours in which the mutant should be killed and a distinguishing test case has to be computed. In our experiments no out-of memory error was encountered. All variables from the tested programs are either of type boolean or of type integer. All integer variables are defined over the finite discrete domain $[-250 \ldots 250]$.

In some cases the inserted fault leads to an infinite execution of the loop structure, e.g., replacing a minus with a plus in a while-structure. Due to the fact that our analysis is based on a static representation of the programs using a fixed number of iterations, the constraint solver is still able to compute an output that satisfies the requirements. But when executing the program and its mutant, the mutant will never stop. Hence, our approach does not require checking program termination for computing test cases.

Another limitation of this approach is that there is no guarantee for computing a solution, i.e., a test case that kills the mutant. In order to identify a faulty statement both the original and its mutant must execute that faulty statement. However there are situations when the faulty statement does not have an influence over the output. In this case a distinguishing test case cannot be determined.

One problem we faced in our experiments was the time needed for computing a solution for some

examples. See for example the results of program `GcdATC` in Table 4.4 where MINION was not able to compute a solution for the versions V3 and V4 within 2 hours. The reason was the huge search space and the fact that no variable ordering was imposed. However, after applying a variable ordering where variables are ordered with respect to their first definition in the program, the situation changed. When using the variable ordering MINION had no problem in killing them in less than a second. Due to this particularity, in our approach we always impose an ordering over the input and output variables. Note that for programs of reduced complexity the variable ordering leads to no gains with respect to time performances.

The obtained results are very promising but further studies have to be performed. In particular, generating test cases for larger programs comprising several thousands lines of code and the ability to handle object-oriented constructs are of interest.

Related research

Distinguishing test cases can play an essential role in improving the debugging process but also in mutation testing [123].

The idea of distinguishing two programs via a test case computed with the help of their constraint representation is, as far as we know, new. However there exists a number of approaches which are based on same principle. The authors of [7] concentrate on discriminating between two non-deterministic Mealy machines via the input and output behavior. They introduce the concept of *distinguishing strategies*. For a Mealy state machine, without an initial state specification, they try to find all the possible initial states differentiated via the input output behavior. The authors of [76] use the concept of distinguishing test cases for the purpose of discriminating between competing hypothesis for a fix propositional language. They define the concept of *discriminating tests* as a test case which for a hypothesis space, e.g, diagnosis, is able to prove that at least on hypothesis is false. In [99] the authors propose a testing mechanism for hardware system based on faulty behavior consistency check. If faulty behaviors exists which are not cover by the existent test suite, new test cases are generated. Another problem which is tackled here is computing the smallest set of tests which can cover all possible faulty behaviors.

In classical mutation testing, the distinguishing test cases are not generated, but a given test suite is assessed with respect to its ability of distinguishing all mutants, i.e. to find all injected faults [50, 90]. It would not be useful to inject faults and then generate a distinguishing test case to find these known

Name	LOC	#I/O	#It	V1	V2	V3	V4	#CO	#Var$_{CO}$
MultATC	12	2/1	2	Killed (0,07s)	Killed(0,06s)	Killed(0,04s)	Killed(0,03s)	47	32
			4	Killed (0,04s)	Killed(0,08s)	Killed(0,07s)	Killed(0,07s)	87	56
			7	Killed (0,01s)	Killed(0,10s)	Killed(0,11s)	Killed(0,11s)	151	92
SumATC	13	2/1	2	Killed (0,4s)	Killed(0,03s)	Killed(0,4s)	Killed(0,4s)	49	34
			4	Killed (0,4s)	Killed(0,07s)	Killed(0,49s)	Killed(0,47s)	89	58
			7	Killed (0,67s)	Killed(0,11s)	Killed(0,62s)	Killed(0,09s)	149	94
MulV2ATC	18	2/1	2	Killed (0,2s)	Killed(0,12s)	Killed(0,21s)	Killed(0,18s)	132	86
			4	Killed (0,34s)	Killed(0,23s)	Killed(0,31s)	Killed(0,31s)	418	258
			7	Killed (2,09s)	Killed(2,09s)	Killed(2,15s)	Killed(2,15s)	1144	696
DivATC	22	2/1	2	Killed (0,06s)	Killed(0,06s)	Killed(0,06s)	Killed(0,06s)	65	52
			4	Killed (0,08s)	Killed(0,08s)	Killed(0,6s)	Killed(0,08s)	105	76
			7	Killed (0,10s)	Killed(0,10s)	Killed(0,09s)	Killed(0,12s)	165	112
GcdATC	24	2/1	2	Killed (0,07s)	Killed(0,35s)	Killed(46s/0,6s)	X/Killed(0,15s)	126	90
			4	Killed (0,08s)	Killed(0,08s)	X/Killed(0,12s)	X/Killed(0,5s)	206	138
			7	Killed (0,10s)	Killed(0,10s)	X/Killed(0,4s)	X/Killed(0,65s)	333	220
RandomATC	52	3/1	2	Killed (0,25s)	Killed(0,25s)	Killed(0,24s)	Killed(0,24s)	303	213
			4	Killed (0,8s)	Killed(0,8s)	Killed(0,8s)	Killed(0,8s)	667	433
			7	Killed (3,5s)	Killed(3,47s)	Killed(3,6s)	Killed(3,59s)	1513	943

Table 4.4: For each program **Name**, **LOC** is the number of statements in the original program, **#I/O** represents the number of inputs and outputs involved in the generated test case, **#It** represents the number of iterations for the loop-unrolling, **V1,V2,V3** and **V4** designate four different mutants of the program, **#CO** designates the number of MINION constraints whereas #**Var$_{CO}$** designates the number of variables associated to the MINION constraint system. The table indicates the time necessary to identify a suitable test case, able to ”kill” the mutant. X stands for not being able to kill the mutant within less than 2 hours

faults. However, the idea found application in model-based mutation testing, where distinguishing test cases are generated from mutated models and then executed on an implementation. Very early, Tai and Su [105] proposed algorithms for generating test cases that guarantee the detection of Boolean operator errors in electronic circuits.

The closest work to ours is [41, 42]. In these papers the authors described the use of constraint solving for test case generation. They also make use of similar conversion techniques. In contrast to the previous work we are focusing on computing distinguishing test cases to be used for debugging and also for test case generation based on program mutations.

4.2.2. Mutation Based Debugging

In the previous section we introduced a constraint based algorithm for generating distinguishing test cases for two different programs written in $\mathcal{L}$. This can be useful both in testing and debugging. In this section we focus on the later and explain how the debugging process can be improved using distinguishing test cases together with repair suggestions generated by program mutation. In particular we show how test cases can be generated to distinguish potential diagnosis candidates. A potential diagnosis candidate, or diagnosis candidate for short, is a statement that can explain why the test cases fail. A diagnosis candidate needs not to be the real bug. But the real bug should be included in the list of diagnosis candidates delivered by an automated debugger.

The main idea behind this approach is to first compute the conflict set by applying the algorithm introduced in Chapter 3 and then for every element of the conflict set try to generate all possible corrections. We do this by implying mutation, i.e., for each statement which is part of the conflict set we generate the set of all possible mutants. We then analyze which mutated versions of the program passes all the test cases (positive and negative) of the test suite. Further discrimination is done by computing distinguishing test cases between the passing mutants.

We now consider the following code snippet to illustrate our combined debugging and testing approach. We use this small program to avoid introducing too much technical overhead and to focus on the underlying idea.

```
        ...
1.      i = 2 * x;
2.      j = 2 * y;
3.      o1 = i + j;
4.      o2 = i * i;
        ...
```

We cannot say anything about the correctness of such a code fragment without any additional specification knowledge. Let us assume that we also have the following test case specifying expected outputs for the given inputs: `x = 1, y = 2, o1 = 8, o2 = 4`. Obviously, the program computes the outputs `o1 = 6` and `o2 = 4`, which contradicts the given test case. Therefore, we know that there is a bug in the program and we have to localize and correct it. At this stage we might use different approaches for computing potential fault locations. If using the data and control dependencies of the program, we might traverse the dependencies from the faulty outputs to the inputs backward. In our example, we are able to identify statements 1, 2, and 3 as potential candidates.

A different way to locate bugs is to consider statements as equations and to introduce correctness assumptions (as proposed in Chapter 3). If the test case together with the assumptions and the equations are consistent, the assumptions stating incorrectness of statements can be used as potential diagnosis candidates. Consider we are looking only for single faults. Let's assume Statement 1 to be faulty, i.e., $AB(S1) = true$, and all other statements to be correct. As a consequence, Statement 1 does not determine a value for variable `i`, hence i can have any other value except the one implied by the behavior of $S1$. However, from *Statement 4* and the test case ($o2 = 2, y = 2$) we can conclude that `i` has to be 2 (if assuming only positive integers). This results in $o1 = i + j = 4$. But the value of $o1$ from the test case is $o1 = 8$ hence a contradiction of the given test case appears. Therefore, the assumption that Statement 1 is a diagnosis candidate cannot be correct.

By implying the same reasoning for making and checking correctness assumptions for the other statements of the program we finally obtain statements 2 and 3 as diagnosis candidates. That is two of the four program statements must be investigated. If in the case of a small program we may investigate a small number of statements, for larger programs we must consider a larger number of potential diagnosis candidates. In this situation, further discrimination is required. One possible solution is the integration of specification knowledge. Another solution is to ask the user about the expected value of intermediate variables (sort of breakpoint) like `i` or `j` for specific test cases. This approach requires more or less executing the program stepwise. Moreover, especially in the case of software

maintenance where a programmer is not very familiar with the program answering questions about values of intermediate variables, both approaches can hardly be implemented. Therefore, we suggest computing test cases that allow distinguishing between diagnosis candidates. More specifically, we are searching for inputs that reveal a different behavior of diagnoses candidates. In case no such distinguishing test case can be computed, the diagnosis candidates are, from the perspective of their input output behavior, equally good.

What prevents us from applying the approach of distinguishing test cases to distinguish diagnosis candidates is the fact that the fault localization approaches only give us information about the incorrectness and correctness of some statements but not about the correct behavior of potentially faulty statements. Hence, computing test cases is hardly possible. In order to solve this problem we borrow the idea of mutation or genetic-based debugging [112, 30]. Mutants, i.e., variants of the original program, are computed and tested against a test suite. The mutants that pass all test cases are potential diagnosis candidates. Computing mutants for all statements and testing them against the test suite is very time consuming and some techniques for focusing on relevant parts of the program have been suggested. In our case, we are able to use the diagnosis candidates for focusing on relevant parts of the program. Hence, when finding a mutant for a diagnosis candidate that passes all test cases, we do not only localize the bug, but also state a potential correction.

According to our proposed approach, for the example program, we must compute mutants for statements 2 and 3, i.e., for the fault candidates. For the sake of simplicity let's presume that we compute only one mutation per conflict. Let m_1 and m_2 be the mutants for statements 2 and 3, e.g.:
$S2: j = 2 * y; /m_1$:`2. j=3*y` and
$S3: o1 = i + j; /m_2$:`3. o1=i+j+2.`

It can be easily seen that both proposed mutants pass the original failing test case (`x = 1, y = 2, o1 = 8, o2 = 4`), that is they are both eligible repair suggestions for the existent bug. Obviously there are more mutations available but for illustrating the distinguishing test cases we only use these two now. A distinguishing test case for these mutants is `x = 1, y = 1`. Mutant m_1 computes the value 5 and mutant m_2 the value 6 for the output variable `o1`. If we know the correct value of `o1` (via an oracle, e.g., the user or the program's specifications), we are able to distinguish the two mutants and eliminate the candidate which explanation (mutant) failed to pass the generated distinguishing test case. From this example we conclude that we are able to discriminate between the diagnosis candidates by means of distinguishing test cases. What remains an open research issue is to provide empirical evidence that the approach is feasible and provides a reduction of diagnosis candidates when

applied to general programs.

In this section, we introduce and discuss the approach and tackle the research question regarding the approach's practical applicability, with some exceptions. The programs used for the empirical evaluation are small programs and they mainly implement algebraic computations. Moreover, we do not handle object-oriented constructs. However, we do not claim to answer the research question completely. We claim that the approach can be used for typical programs comprising language constructs like conditionals, assignments, and loops. The structures of the used programs are similar to those of larger programs or at least we do not see why there should be any big differences.

Next we present the theoretical background of our approach. We introduce the basic definitions and a new program serving as running example. Since the debugging approach is based on a model of the program, we briefly introduce the constraint representation of the example program that serves our purpose in the next section. This model can be used for debugging as well as for computing distinguishing test cases. We then introduce the diagnosis algorithm using constraints and mutations. Last we present and discuss the obtained empirical results and the related research.

Discriminating between the Bug candidates

The proposed approach uses the models of programs to compute the mutants and to perform debugging. The class of debugged programs adopts the syntax of the language $\mathcal{L}$. We further restrict the data domain of the language to integers and booleans. In Figure 4.9 we state an example program, which serves as running example. The program implements the division of two natural numbers where a bug is introduced in Line 1.

For computing the conflict set we rely on the framework proposed in Chapter 3. The definitions of the debugging problem, of a test case and of distinguishing test cases, are given in Section 3.1, Definition 4.2.1. Implying the prior definitions and algorithms, in the context of our work, we still need to formally define program mutation.

Definition 22 (Mutant) *Given a program Π and a statement $S_\Pi \in \Pi$. Further let S'_Π be a statement that results from S_Π when applying changes like modifying the operator or a variable. We call the program Π', which we obtain when replacing S_Π with S'_Π , the mutant of program Π with respect to statement S_Π.*

```
1.     tmp = (a + 1); // ERROR
2.     if (b == 0) {
3.         result = -1;
        } else {
4.         result = 0;
5.         while (tmp > 0) {
6.             result = result + 1;
7.             tmp = tmp - b;
            }}
```

Figure 4.9.: A program for dividing two natural numbers

Example 10. Let's presume that in the program $\Pi \in \mathcal{L}$ from Fig. 4.9 we change statement $1.tmp = (a+1);$ to $tmp = a;$. Then the resulted program $\Pi' \in \mathcal{L}$,

```
1.     tmp = a;
2.     if (b == 0) {
3.         result = -1;
        } else {
4.         result = 0;
5.         while (tmp > 0) {
6.             result = result + 1;
7.             tmp = tmp - b;
            }}
```

is called the mutant of Π with respect to statement 1. In this special situation the chosen mutant is also the correct repair suggestion for the bug introduced in the program from Fig. 4.9. □

Program mutation together with generation of distinguishing test cases, can be used for improving the debugging results in combination with any debugging engine as long as access to the source code is provided. In our case we use our constraint based approach to compute the conflict set. For the resulted conflict set we compute all mutants. The stepwise process for integrating the mutation-based add-on is:

1. ***The first step*** comprises the computation of bug candidates, i.e., program statement that might cause the revealed misbehavior, from the constraint representation of a program $\Pi \in \mathcal{L}$.
2. ***In the second step***, for each candidate, a set of mutants is computed that would lead to a new program passing all previously failing test cases. If no such mutant can be found, the bug candidate is removed from the list of potential candidates.
3. ***In the third step***, distinguishing test cases are computed. These allow choosing between two randomly selected bug candidates. The third step can be executed several times to further reduce the number of bug candidates.

The first step has already been detailed in the previous chapters. What we still have to explain are the last two steps of the mutation based debugging process. We do that in what follows.

Let CON_Π be the constraint representation of a program Π and CON_T the constraint representation of a failing test case T. The debugging problem formulated as a CSP comprises CON_Π together with CON_T. Note that in CON_Π assumptions about correctness or incorrectness of statements are given, which are represented by a variable AB assigned to each statement. The algorithm for computing bug candidates calls the CSP solver using the constraints and asks for a return value of AB as a solution. The size of the solution corresponds to the size of the bug, i.e., the number of statements that must be changed together in order to explain the misbehavior. We assume that single statement bugs are more likely than bugs comprising more statements. 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 and iterates calling the constraint solver. This is done until either a solution is found or the maximum size of a bug, which is equivalent to the number of statements in Π, is reached.

For the sake of clarity we present a simplified version of the *CSP_Conversion&Debugging* (Algorithm 4, Chapter 3) algorithm, which is given in Algorithm 8.

For example, for the constraint system corresponding to the program from Fig. 4.9 the constraint solver MINION finds 5 possible explanations for the failing test case $I : (a\_0 = 0, b\_0 = -250), O : (result\_7 = 0)$ in less the 0.1s. This result is very satisfactory, especially with respect to computation time. However, further steps might be performed in order to reduce the size of the bug candidates. For this purpose we suggest to use mutations.

Assume a faulty program Π and a failing test case (I, O). Let D_{AB} be the set of bug candidates obtained when calling Algorithm 8 on the constraint representation of Π and (I, O). The following algorithm makes use of program mutations for further restricting D_{AB}.

Algorithm 8 CSP_Debugging (Π, T)

Require: A constraint representation CON_Π of a program Π, and a constraint representation CON_T of a failing test case T.

Ensure: A set of minimal bug candidates D_{AB}.

1: Let i be 1.
2: **while** $i \leq$ number of statements in Π **do**
3: Call the **constraint solver** over $CON_{(\Pi,T)}$ to search for solutions regarding the AB variables, where only i statements are allowed to be simultaneously incorrect.
4: **if** constraint solver returns a non-empty set of solution **then**
5: $D_{AB} =$ solution set.
6: **return** D_{AB}.
7: **end if**
8: Otherwise, let i be $i+1$.
9: **end while**
10: **return** the empty set as result.

Algorithm 9 Generate_Valid_Mutants (D_{AB}, Π, T)

Require: A set of bug candidates D_{AB}, the faulty program Π, and the failing test case T.

Ensure: A set of mutants Mut_Π of program Π.

1: Let Mut_Π be the empty set.
2: **for all** elements $d \in D_{AB}$ **do**
3: Generate all mutants of program Π with respect to the statements from d and store them in V_{Mut}.
4: Add every program $\Pi' \in V_{Mut}$ passing test case T to Mut_Π.
5: **end for**
6: **return** Mut_Π.

Algorithm 9 (**Generate_Valid_Mutants**) returns for the faulty program Π a set of repair possibilities Mut_Π. Due to the usage of the debugging algorithm **CSP_Debugging** (Algorithm 8), we compute the repair only for the resulting bug candidates set D_{AB}. A mutant is part of Mut_Π, i.e., a repair, if and only if it is able to pass the failing test case T. If more test cases are available, a mutant has to pass all of them in order to be considered correct. Hence, we expect that the number of bug candidates can be

reduced. Moreover, since mutation is only applied for bug candidates we do not need to compute all possible mutations even in the case when they cannot explain the revealed misbehavior.

Definition 23 (Fault Explanations Set) *Let Mut_Π be the set of mutants resulted after applying the Generate_Valid_Mutants algorithm (Algorithm 9) for the tuple (D_{AB}, Π, T). Let $M_i \subseteq Mut_\Pi$ be the set of all mutants of program Π computed with respect to a diagnosis $D_i \subseteq D_{AB}$, such that $\forall M_i, M_j \subset Mut_\Pi, i \neq j$, $M_i \cap M_j = \{\phi\}$ then $\bigcup_{i=1..|D_{AB}|} M_i = Mut_\Pi$. We call Mut_Π the **explanations set** for the failure of program Π on a test case T. This set can also be used for computation of a valid repair suggestion.*

The number of repair possibilities for a statement of the D_{AB} set is strongly tied to the capabilities of the used mutation operators and the used mutation tool. Because of this fact this part of the approach is as good as the available capability of the used mutation tool. Note that after applying the **Generate_Valid_Mutants** algorithm, in our experiments we were able to eliminate between 20% and 60% of the bug candidates. That happened because of the inability of the suggested repair to pass the test case. Hence, filtering based on mutations was very successful.

The last step of our algorithm comprises the integration of distinguishing test cases to further reduce the bug candidates set. Let Mut_Π be the set of mutants for a program Π obtained after applying the **Generate_Valid_Mutants** algorithm. And let CON_{Mut_Π} be the constraint representation of the programs from Mut_Π.

Algorithm 10 (**DistingTC_Generator**) searches, i.e., *generates*, a test case that can distinguish between two mutants. The algorithm in the current form is restricted to finding only one pair of such mutants, but can be easily changed in order to compute several different pairs where a distinguishing test case is available. The algorithm generates a valid test case which has the property that is both correct with respect to the program specification and is able to eliminate at least one mutant. We use the resulted test case against the remaining mutants, i.e., step 23, and eliminate the mutants failing on it. The only disadvantage of this algorithm is that Step 10 requires an interaction with an oracle. If no automated oracle is available, user interactions are required and prevent the approach from being completely automated.

To solve the constraint system resulted at step 9 we use the MINION constraint solver. Another particularity of this approach is that, for the CSP to be solvable, the name of the variables of the two mutants should differ. This is however an encoding problem which can be easily overcome by encapsulating in the name of each variable the name of the mutant file.

Algorithm 10 DistingTC_Generator (D_{AB}, Π, T)

Require: A set of valid repair possibilities, Mut_Π, for a faulty program Π and their constraint representation CON_{Mut_Π}.

Ensure: A subset of Mut_Π.

1: Let $Tested$ be empty.
2: **if** $\exists$ mutants $\Pi', \Pi'' \in Mut_\Pi$ with $(\Pi', \Pi'') \notin Tested$ **then**
3: Add (Π', Π'') to $Tested$ and proceed with the algorithm.
4: **else**
5: **return** Mut_Π.
6: **end if**
7: Let $CON_{\Pi'}$ and $CON_{\Pi''} \in CON_{Mut_\Pi}$ be the constraint representation of programs Π' and Π'' respectively.
8: Let CON_{TC} be the constraints encoding $Input_{\Pi'} = Input_{\Pi''} = I \wedge Output_{\Pi'} \neq Output_{\Pi''}$.
9: Solve the CSP: $CON_{\Pi'} \cup CON_{\Pi''} \cup CON_{TC}$ using a constraint solver.
10: Let O be the correct output for the original program Π on input I (derived from user interaction or specifications).
11: **if** $Output_{\Pi'} = O \wedge Output_{\Pi''} \neq O$ **then**
12: Delete Π'' from Mut_Π.
13: **end if**
14: **if** $Output_{\Pi''} = O \wedge Output_{\Pi'} \neq O$ **then**
15: Delete Π' from Mut_Π.
16: **end if**
17: **if** $Output_{\Pi'} \neq O \wedge Output_{\Pi''} \neq O$ **then**
18: Delete Π' and Π'' from Mut_Π.
19: **end if**
20: **if** (CSP has no solution) **then**
21: go to step 2.
22: **end if**
23: **for all** $\Pi' \in Mut_\Pi$ **do**
24: **if** Π' fails on generated test case (I, O) **then**
25: Delete Π' from Mut_Π.
26: **end if**
27: **end for**
28: **return** Mut_Π.

When using the above approach for the example from Fig. 4.9 we are able to reduce the conflict set to one element, which was also the correct one. For more information regarding distinguishing test cases and their computation using MINION, we refer the interested reader to [124].

Regarding step 2 of the *DistingTC_Generator* algorithm (Algorithm 10): if more than one conflict exists, it is preferred to first choose the pair of mutants such that, they are not belonging to the set of explanations of the same fault candidate. This increases the chances of finding a valid distinguishing test case, but also of eliminating a fault candidate without the necessity of generating new distinguishing test cases. This property is important when the main focus is debugging. If we already reduced the conflict set to one element, then we can use the distinguishing test case approach in order to identify the best repair suggestion to the user.

To obtain the program's set of mutants relative to the set of fault candidates we rely on the JAVA mutation tool MuJava [127]. MuJava is a Java based mutation tool, which was originally developed by Offut, Ma, and Kwon. Its main three characteristics are:

1. Generation of mutants for a given program.
2. Analysis of the generated mutants.
3. Running of provided test cases.

Due to the new implemented add-ons, the tool supports a command line version for the mutation analysis framework, which offers an easy integration of the tool in the testing or debugging process.

Offutt proved that the computational cost for generating and executing a large number of mutants can be expensive, and thus he proposed a selective mutation operator set that is used by the MuJava tool. It works with both types of mutation operators:

- Method level mutation operators (also called traditional), which modify the statements inside the body of a method;
- Class level mutation operators, which try to simulate faults specific to the object oriented paradigm (for example faults regarding the inheritance or polymorphism).

For our experiments we take into account only the traditional mutation operators, i.e., at method level. As we do not support object oriented programs, the Class level operators are ignored. Moreover, we further restrict the mutation operators to mutations on expressions comprising deletion, replacement, and insertion of primitive operators (arithmetic operators, relational operators, conditional operators, etc.). Mutation by deletion of operands or statements was proved to be inefficient [2]. Because

of the selected tools there are currently some limitations of our implementation. If the bug is on the left hand side of an assignment we cannot correct it. Another limitation is with respect to constants. If the bug is due to an initialization, MuJava is not able to generate any mutants. Missing statements are another limitation of the approach. We currently do not consider bugs because of missing statements. Finally, there is a limitation regarding multiple bugs in one statement. In this case the MuJava tool is not able to mutate more than one variable or operator per statement and mutant, i.e., each mutant contains only one change when compared with the original program (this limitation is however easy to overcome). The last problem regarding mutation is that sometimes equivalent mutants are generated. For example if we have statement $a = b + c$ and *MuJava* generates for this statement the mutants:

```
m1: a = b+c+1;
m2: a = b+c+(2-1);
```

then, $m1$ and $m2$ are told to be equivalent mutants, i.e., they are both describing the same semantically functionality. There exist mechanisms through which equivalent mutants are detected and removed, however this is not always feasible.

4.2.3. Empirical Results

We tested our approach against a set of faulty programs. In each program we injected a single fault. All the faults are found at the right-hand side of the assignment operator and with the exception of the *tcas03* program all faults are functional faults. We used as test oracle the original bug free version of each faulty program. Using the output values of the original bug-free program we were able to decide which of the mutants are to be eliminated after computing the distinguishing test cases. In a real life situation we cannot benefit from the existence of such a program. Therefore, we must rely on the user or a given formal specification to determine the correct output for a given input.

The process of mutant generation, program to CSP conversion, and the computation of the conflict set are fully automated. However the generation of the distinguishing test cases require user intervention to decide which test case is valid with respect to the correct behavior of the program.

In order to obtain the empirical results, we applied the following process. For each program we first performed the conversion into its constraint representation. Then, we computed the fault candidates. For each fault candidate, i.e., faulty statement, we computed all its possible mutants. We eliminated from the generated set of mutants all mutants which were not able to pass the error revealing test case.

In addition, we tested the number of oracle-interactions required to obtain the minimal set of faulty components. By an oracle-interaction we understand repeating the **DistingTC_Generator** (Algorithm 10) algorithm until no other distinguishing test case can be generated, i.e., each time we applied the algorithm we asked the oracle, i.e., the original fault free program in our case, to provide the correct output for the generated test case.

The results of the empirical study are given in Table 4.5. In most cases we were able to eliminate more than half of the initial fault candidates set. Reducing the diagnosis candidates by eliminating those candidates where no mutant that passes the original test suite can be found, is very effective. The use of distinguishing test cases further reduces the number of fault candidates. Thus finally, only one diagnosis candidate remains, which was always the correct one. When using larger programs like *tcas* a reduction to one diagnosis candidate was not possible. However, even in this case the approach leads to a reduction of more than 60% regarding the computed diagnosis candidates.

Another factor, which influences the quality of the obtained results, is the way of choosing the mutant pairs for computing distinguishing test cases. There is no way to predict if a certain pair of mutants will produce the best or worst distinguishing test case. Therefore, we randomly selected the pair of mutants when carrying out the empirical evaluation. For example, we observed that after trying out all mutant pairs for the *DivATC_V4* program the best distinguishing test case would lead to 1 element in the conflict set contrary to 3 as given in Table 4.5.

Another particularity of Table 4.5 is that, columns **#UI** and $|\mathbf{Diag_{TC}}|$, for some entries, have more than one value. This values indicate after how many user iterations, **#UI**, the minimal number of fault candidates $|\mathbf{Diag_{TC}}|$ was obtained, and how this number evolved at each user iteration, e.g., for *GcdATC_V2* after 5 user iterations only one fault candidate remains, whereas for *SumPowers_V2* only after two iterations the conflict was isolated.

It is also worth noting that computing the diagnosis candidates and the distinguishing test cases using the CSP solver MINION was very fast. For all examples, the necessary time never exceeded 0.3 seconds using a Pentium 4 Dual core 2 GHz with 4 GB of RAM computer. Hence, for smaller programs or program parts that can be separately analyzed like methods, the proposed approach is feasible.

Name	It	$\mathbf{Var}_{\Pi}$	$\mathbf{LOC}_{\Pi}$	Inputs	Outputs	$\mathbf{LOC}_{\mathbf{SSA}}$	|CO|	$\mathbf{Var}_{\mathbf{CO}}$	|Diag|	$\mathbf{\|Diag_{filt}\|}$	#UI	$\mathbf{\|Diag_{TC}\|}$
DivATC_V1	2	5	21	2	1	32	33	29	3	2	1	2
DivATC_V2	2	5	21	2	1	32	33	29	5	3	1	1
DivATC_V3	2	5	21	2	1	32	33	29	3	2	1	2
DivATC_V4	2	5	21	2	1	32	33	29	4	4	1/2	3(1)/1
GcdATC_V1	2	6	35	2	1	49	61	46	2	2	1	1
GcdATC_V2	2	6	35	2	1	49	61	46	10	3	1/2/3/4/5	3/3/2/2/1
GcdATC_V3	2	6	35	2	1	49	61	46	2	2	1	1
MultATC_V1	2	5	16	2	1	26	24	19	2	2	1	1
MultATC_V2	2	5	16	2	1	26	24	19	2	2	1	1
MultATC_V3	2	5	16	2	1	26	24	19	2	2	1	1
MultATC_V4	2	5	16	2	1	26	24	19	5	2	1	1
MultV2ATC_V1	2	6	20	2	1	49	67	46	6	2	1	1
MultV2ATC_V2	2	6	20	2	1	49	67	46	2	1	1	1
MultV2ATC_V3	2	6	20	2	1	49	67	46	6	1	1	1
SumATC_V1	2	5	18	2	1	27	24	20	2	2	1	1
SumATC_V2	2	5	18	2	1	27	24	20	3	2	1	1
SumATC_V3	2	5	18	2	1	27	24	20	5	2	1	1
SumPowers_V1	2	11	36	3	1	72	87	70	16	6	1/2/3/4	4/4/2/2
SumPowers_V2	2	11	36	3	1	72	87	70	11	6	1/2	2/1
SumPowers_V3	2	11	36	3	1	72	87	70	11	1	1	1
tcas08	1	48	125	12	1	125	98	132	27	13	1/2/3/4	11/11/11/10
tcas03	1	48	125	12	1	125	98	132	27	13	1/2/3/4	13/12/9/9

Table 4.5: Each program **Name**, has associated a number of iterations **It**, the number of variables $\mathbf{Var}_{\pi}$, its size given in lines of code $\mathbf{LOC}_{\Pi}$, the number of inputs **Inputs**, number of outputs **Outputs**, the size of its SSA representation given as lines of code$\mathbf{LOC}_{\mathbf{SSA}}$, the number of MINION constraints $|\mathbf{CO}|$, the number of MINION variables over which the constraint system is defined $\mathbf{Var}_{\mathbf{CO}}$, the number of fault candidates $|\mathbf{Diag}|$, the size of the conflict set resulted after applying *Generate_Valid_Mutants* algorithm, $|\mathbf{Diag}_{\mathbf{filt}}|$, the number of calls to the *DistingTC_Generator* algorithm, **#UI** to obtain the number of fault candidates $|\mathbf{Diag}_{\mathbf{TC}}|$.

4.2.4. Related research

In [72] the authors present a method for combining model based debugging with spectrum based techniques for further reducing the fault candidates set. They rely also on a debugging engine for computing the conflict set, and based on spectrum-based techniques they rank the fault candidates according to their probabilities to be the cause for program's failure. They rely both on positive and negative test cases for computing this ranking. However this technique does not eliminate any of the fault candidates and can sometimes mislead the user, as the highest ranked component must not always be the actual fault. Another limitation of the approach is the fact that, in the case that the used coefficient (Ohciai) ranks the same for all faulty components, no discrimination is possible. Contrary to this, our work combines model based debugging with distinguishing test case generation and mutation to reduce the fault candidates set. Hence in our case the conflict set is reduced to a smaller one but no ranking is possible between the components. A combination of both methods would be interesting and would most probably lead to an improvement of the debugging results.

In [112] and more recently [30], the authors describe the application of mutations and genetics programming to software debugging. In order to avoid computing too many mutants, the authors use focusing techniques based on dependencies and spectrum-based methods respectively. The use of mutations is similar to our work. The difference is that we are using constraint-based debugging for focusing and integration of testing for reducing the size of the conflict set, which, to the best of our knowledge, has not been introduced before.

4.3. Conclusion

In order to improve the diagnostic precision, we first extended our approach by integrating specification knowledge into the Framework proposed in Chapter 3. Such knowledge is provided by program annotations like pre- and postconditions or loop invariants.By integrating specification knowledge in the debugging process we managed to obtain considerable gains with respect to the size of the conflict set. However computing the right annotation is not always easy. This restricts somehow the approach only to programs well specified. Another advantage of this method is the possibility to use pre- and post-condition for modularizing the debugging process at method level, i.e., isolates the method where the error may be and focus the debugger on it. Finally we provide the first empirical results for model-based debugging based on constraint representation and specification knowledge.

In particular, we compare the diagnostic results obtained with and without using loop invariants. Our results clearly demonstrate the advantage of integrating specification knowledge wrt. the number of statements which are eliminated as possible fault candidates.

The second extension which we tackle in this chapter is the integration of mutation and distinguishing test cases for reducing the conflict.

A distinguishing test case for two programs is a test case that reveals different values for the output variables of the programs when using the same input. The application areas are automated test case generation based on program mutations and fault localization. With respect to mutation testing, the distinguishing test cases serve the following purposes: The tests are defined to kill the set of known mutants, which would not be useful in itself. However, these test cases (1) can be used as a basis for further regression testing, (2) they guarantee a certain coverage in the code (it is well-known that mutation coverage subsumes classical coverage criteria, like e.g. branch coverage), (3) via the mutation testing assumption of the coupling effect, these test cases will detect more subtitle faults in the program. With respect to fault localization distinguishing test cases are of interest for reducing the number of fault candidates.

Besides test case generation, we make use of distinguishing test case approach to extend and improve the results of a debugger. If we could compute repair suggestions over a conflict set (several per conflict) and try to distinguish them via a test case and an oracle (which decides upon the correctness of the output), we could further eliminate conflict explanation (together with the conflict).

Hence we proposed an approach for restricting the number of potential diagnosis candidates by providing distinguishing test cases. A distinguishing test case for two diagnosis candidates is characterized by a set of inputs that reveal different executions for both diagnosis candidates such that they can be distinguished with respect to their output behavior. Just using the distinguishing test case alone, we are not able to decide which diagnosis candidate to remove or if we should eliminate both from the list of candidates. This can only be done after consulting a test oracle, e.g., the user or a formal specification, for the expected output of the distinguishing test case. Candidates where the computed output is not equivalent to the expected one can be eliminated. The advantage of this approach is that only the input-output behavior of a program is used for distinguishing diagnosis candidates. Moreover, the approach computes additional test cases based on their discriminating power for distinguishing diagnosis candidates. Usually, test cases are generated for fulfilling coverage criteria like statement coverage or branch coverage.

Apart from the theoretical contribution, we present first empirical results of the mutation and dis-

tinguishing test cases approach. The results indicate that the approach allows a substantial reduction of the diagnosis candidates. For smaller programs we were able to reduce the diagnosis candidates to the real bug. Obviously, this was not always the case. For larger programs more diagnosis candidates remain. This has been somehow expected because programs cannot be usually corrected only by replacing one statement with another. Instead, the right repair actions might comprise changes at different positions in the program.

Chapter 5

Complexity

In Chapter 3 we started a discussion about the complexity of our approach, there we show that the algorithm for converting a program into its constraint representation, has a complexity of $O(N^2)$. The last open issue is with respect to the actual complexity of debugging. In our proposed framework (see Chapters 3 and 4) we make use of constraints to encode the debugging problem. Hence the complexity of debugging is, in our case, given by the complexity of solving the resulted constraint system.

A CSP can be solvable in either PTIME or can be NP complete. The question is: can we guarantee that for every debugging problem (Π, TC) the resulted CSP is always solvable in polynomial time? In this chapter we answer this question and show that for every program $\Pi \in \mathcal{L}$, the constraint system associated to its debugging problem (Π, TC) can be solved in polynomial time. Moreover we identify a metric capable of indicating the complexity of debugging a certain program.

The objective of this chapter is not to provide a new model for debugging but instead to focus on the computational requirements in terms of running time needed for finding all minimal faults in the source code given a failing test case. A worst case estimation considers the computational complexity of the program to be debugged and the computational complexity of the debugging itself. In the case of model-based diagnosis all subsets of the set of components, i.e., program statements, have to be considered. This estimation is very coarse and needs to be improved in order to be of practical interest. In particular, we aim to provide a complexity measure for debugging, given the debugging problem, i.e., the source code of a program and a test suite.

Thorup [107] presented a similar approach on stating complexity of certain tasks on given programs.

In his paper the focus is on program analysis methods used in compiler construction. The main contribution is that there is a close relationship between the complexity and the use of programming language constructs like goto-statements. Hence, the complexity, given in terms of tree width [14], depends on the programming language. It is worth noting that Thorup's work is not directly of use in the case of debugging because of its dynamic nature.

This chapter relies on the following prior published papers:

- *Converting Programs into Constraint Satisfaction Problems [121]*;
- *On the Compilation of Programs into their equivalent Constraint Representation [122]*;
- *On the complexity of program debugging using constraints for modeling the programs syntax and semantics [126]*;

We extend Thorup's work on the dynamic case. In particular, we consider a constraint representation of programs to be used for debugging [18, 84, 85, 122]. Moreover, instead of tree width we use hypertree width [43, 44, 45, 46] as a measure of the structural properties of programs. Using these two ingredients of our approach, we are able to identify the reasons for the debugging complexity.

The structure of this chapter is as follows. In Section 5.1 we start by introducing the underlying theory behind the structural tractability of a constraint system. Here we also reflect on previous theorems stating the relationship between the debugging problem and the hypertree width and give a summary. Section 5.2 introduces the theory behind computing complexity from the structural representation of the constraint system associated to the debugging problem. In Section 5.3 we present the results of our complexity study, correlating the debugging time with different metrics and analyzing the relationship between them. In Section 5.4 we draw our conclusions.

5.1. Structural properties of a CSP

Our model based debugger implies the use of a state-of-the-art constraint solver for computing the debugging results. That is, both the program Π and the error revealing test case TC, which describe the debugging problem (Π, TC), are converted into a constraint system. Furthermore, each constraint system, based on the constrains scopes, has associated a hypergraph describing the dependencies that are established between different constraints. We propose to use the structural properties of

$$TS_{\texttt{simple}} = \left\{ \begin{array}{c} (\{\texttt{red}=1,\texttt{blue}=5,\texttt{green}=8,\texttt{yellow}=2\}, \\ \{\texttt{sweet}=40,\texttt{sour}=40,\texttt{salty}=7,\texttt{bitter}=49\}) \end{array} \right\}$$

When executing simple using the test case in $TS_{\texttt{simple}}$ the program delivers wrong values for sweet, sour, and bitter as outputs.

```
 1.    red = 2 * red;  // BUG: red=5*red;
 2.    sweet = red * green;
 3.    sour = 0;
 4.    i = 0;
 5.    while (i < red) {
 6.        sour = sour + green;
 7.        i = i + 1; }
 8.    salty = blue + yellow;
 9.    yellow = sour + 1;
10.    bitter = yellow + green;
```

Figure 5.1.: The program `simple` adapted from [48]

the constraint system, i.e., its hypergraph, to decide the complexity of a given debugging problem (Π, TC).

We start by first explaining the basic notions about the structural properties of a constraint system. This, among others, includes: the definition of hypergraph, constraint graph, hypertree decomposition, tree decomposition, hypertree width and the tree width.

Throughout this chapter we use the program from Fig. 5.1 as running example. Its SSA representation (for two iterations) is given in Fig. 5.2.

Theorem 2 (Solution Equivalence) *Given a program Π, its SSA representation Π' and its corresponding CSP C_Π, then the value assignments of the variables in Π', which are caused by executing Π' on an input I are a solution to the corresponding CSP C_Π and vice versa.*

This holds now directly for debugging and we are interested in classifying programs with regard to their debugging complexity. We define debugging complexity as a measure that corresponds to the

```
1.  red_1 = 2 * red_0; //scope (1):  (red_1, red_0)
2.  sweet_0 = red_1 * green_0; //scope (2):  (sweet_0, red_1, green_0)
3.  sour_0 = 0;  //scope (3):  (sour_0)
4.  i_0 = 0; //scope (4):(i_0)
5.  cond_0 = (i_0 < red_1); //scope (5):(cond_0, i_0, red_1)
6.  sour_1 = sour_0 + green_0;//scope (6):(sour_1, sour_0, green_0)
7.  i_1 = i_0 + 1; //scope (7):(i_1, i_0)
8.  cond_1 = cond_0 && (i_1 < red_1); // scope (8):(cond_1, cond_0, i_1, red_1)
9.  sour_2 = sour_1 + green_0; //scope (9):  (sour_2, sour_1, green_0)
10. i_2 = i_1 + 1; //scope (10):(i_2, i_1)
11. i_3 = ϕ(i_2, i_1, cond_1); //scope (11):(i_3, i_2, i_1, cond_1)
12. sour_3 = ϕ(sour_2, sour_1, cond_1);//scope (12):(sour_3, sour_2, sour_1, cond_1)
13. i_4 = ϕ(i_3, i_0, cond_0); // scope (13):  (i_4, i_3, i_0, cond_0)
14. sour_4 = ϕ(sour_3, sour_0, cond_0); //scope (14):  (sour_4, sour_3, sour_0, cond_0)
15. salty_0 = blue_0 + yellow_0;// scope (15):(salty_0, blue_0, yellow_0)
16. yellow_1 = sour_4 + 1; // scope (16):  (yellow_1, sour_4)
17. bitter_0 = yellow_1 + green_0; // scope (17):  (bitter_0, yellow_1, green_0)
```

Figure 5.2.: The SSA form corresponding to the program from Figure 5.1 for two iteration unrolling. Additionally each statement has associated the scope of the derived constraint.

complexity of computing a solution using CSP algorithms. In the following, we discuss structural properties of CSPs, which can be used for classification and which are based on the hypergraph representation of programs.

Definition 24 (Constraint scope) *Let CO be a set of constraints over a set of variable V, defined on the set of domains D. For all constraints $C_i \in CO$, $i = 1..|CO|$, the set of all variables $V_i \subseteq V$ involved in the relation C_i is called the **scope of the constraint** C_i. The set of all constraints scopes is called the **scheme** of the constraint system.*

Example 11. Let's presume we have the following three-constraints system (one boolean and two arithmetical) : $C1 : a * b = c$
$C2 : a + d < 10$
$C2 : a1 + a2 = result$
$D = \{D_a, D_b, D_c, D_d, D_{a1}, D_{a2}, D_{result}\}$
$V = a, b, c, d, a1, a2, result$

The scope of constraint $C1$ is (a, b, c), for $C2$ we have (a, d) and for $C2$, $(a1, a2, result)$. □

In Fig. 5.2 for each SSA statement we depict the resulted constraint scope. We ignore the abnormal and the auxiliary variables required by the specific constraint modeling language.

Based on the scopes involved in the constraints of CO we build the graphical representations of the structural dependencies of the constraint system. The graphical representation of a CSP is used to compute the complexity of the debugging process.

Definition 25 (Graph and Hypergraph) *A graph is a pair $\{V, E\}$, where $V = v_1 ... v_n$ is a set of vertices, and $E = \{(v_i, v_j) | v_i \neq v_j \wedge v_i, v_j \in V\}$ a set of edges. A **hypergraph** is a pair (HV, HE), where HV is a set of vertices and HE a set of hyperedges. Each hyperedge is a non-empty subset of HV, $HE = \{HE_1 ... HE_t\} \subseteq HV$; i.e., it may connect more than two vertices.*

Definition 26 (Constraints Graph) *The graph of a constraint system is an undirected graph (V, E) where the vertices V are the constraint's variables, and $\forall v_i, v_j \in V$, $v_i \neq v_j$, if v_i, v_j are involved in at least one common constraint scope, there exists one and only one edge $(v_i, v_j) \in E$, whereas the set E represents the edges of the constraints graph.*

The graph corresponding to the program from Fig. 5.2 is given in Fig. 5.3.

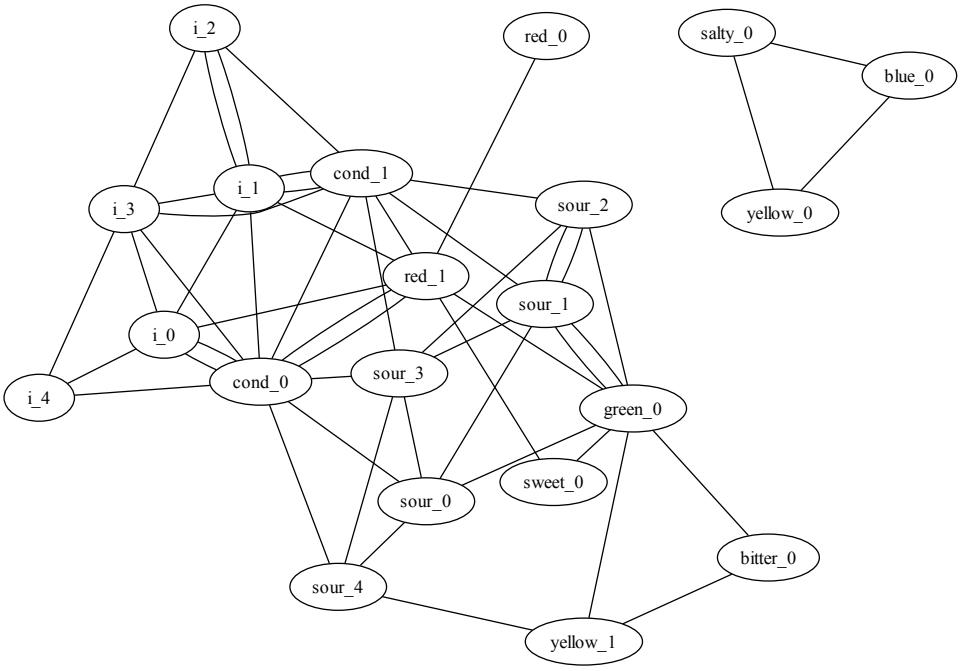

Figure 5.3.: The constraint graph corresponding to the program from Fig.5.2

Definition 27 (Tree Decomposition[81]) *A tree decomposition of a hypergraph (VE, HE) is a pair $< T, \chi >$ where $T = (V, E)$ is a rooted tree, and χ is a labeling function which associates to each vertex $v \in V$ the set $\chi(v) \subseteq VE$ such that:*

1. *For every hyperedge $h \in HE$ there exists $v \in V$ such that $scope(h) \subseteq \chi(v)$*
2. *For each variable of the hypergraph, $v_i \in VE$ the set $\{vt \in V | v_i \in \chi(vt)\}$ induces a subtree of T*

Every hypergraph that contains cycles can be converted into a hypertree by implying hypertree decomposition techniques. The following definition is adapted from [46].

Definition 28 (Constraints hypergraph) *The hypergraph of a constraint system is a tuple (V, HE), where V is the variables set of the constraint system, and the hyperedges set is defined by $HE = \{\{scope(C_i)\} | \forall C_i \in CON\}$, where function $scope(C_i)$ returns a subset of V designating the scope of constraint C_i. Hence the hypergraph for a CSP represents variables as vertices and the constraint scopes as edges.*

It is well known that solving a CSP is NP-complete. However, there are CSPs that can be solved in polynomial time. CSPs whose corresponding hypergraph is acyclic can be solved in polynomial time [31]. From Definition 28 we know that the induced hypergraph of a CSP can easily be obtained by creating a vertex for each variable, and an edge for each constraint. Because solutions to CSPs with acyclic hypergraphs can be computed fast, the question remains whether all CSPs can be converted to an equivalent CSP with an acyclic hypergraph. Four our class of programs, the answer to this question is yes. By joining constraints we are able to finally compute such a CSP but at the cost of increasing the number of constraint tuples. Therefore, it is of interest to find an equivalent acyclic CSP where only a small number of constraints have to be joined. This problem is referred to as composition problem and hypertree decomposition [45, 44].

Definition 29 (Hyper Tree) *A hypertree (HE, E) is an acyclic rooted hypergraph (VE, HE), where the hyperedges $\{\forall HE_i \in HE, HE_i \subseteq VE\}$ become the tree vertices and the tree edges connect every two hyperedges that share at least one common variable $v_i \in VE$.*

Every hypergraph that contains cycles can be converted into a hypertree by implying hypertree decomposition techniques. The following definition is adapted from [46].

Definition 30 (Hypertree Decomposition) *[46] A hyper tree decomposition of a hypergraph (VE, HE) is a triple $< T, \chi, \psi >$ where $T = (V, E)$ is a rooted tree, and χ and ψ are labeling functions which associate with each vertex $v \in V$ two sets $\chi(v) \subseteq VE$ and $\psi(v) \subseteq HE$ satisfying the following:*

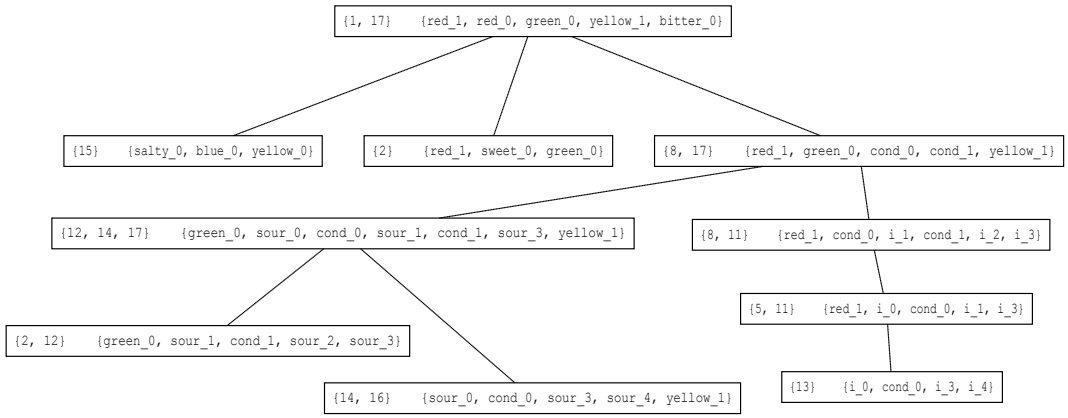

Figure 5.4.: The hypertree corresponding to the constraint system resulted from the program given in Fig. 5.2

1. *For every hyperedge $h \in HE$ there exists $v \in V$ such that $h \in \psi(v)$ and $scope(h) \subseteq \chi(v)$*
2. *For each variable of the hypergraph, $v_i \in VE$ the set $\{vt \in V | v_i \in \chi(vt)\}$ induces a subtree of T*
3. *For each $vt \in V$, $\chi(vt) \subseteq (\psi(vt))$*
4. *For each vt in$V$, scopes $(\psi(vt)) \cap \chi(T_{vt}) \subseteq \chi(vt)$, where $T_{vt} = (V_{vt}, E_{vt})$ is the subtree of T rooted at vt and $\chi(T_{vt}) = \bigcup_{u \in V_{vt}} \chi(u)$*

Definition 31 (Hypertee width) *The hypertree width of a hypertree decomposition is given by $hw = max_v(\psi(v))$, i.e., the maximum of constraints joined in one vertex of the hyper tree.*

From now on we presume that by hypertree width we always refer to the optimal hypertree decomposition.

The hypertree decomposition of the program from Fig. 5.2 is given in Fig. 5.4. The hypertree width of the resulted hypertree is $hw = 3, i.e., \{12, 14, 17\}$. That is the maximum number of constraints that have to be joined in order to remove all the cycles of the program is 3.

5.2. Estimating complexity

What are the consequences when joining constraints for the computational complexity? In [43] the authors state that worst case scenario for computing a solution to a CSP is limited by $O(|I|^{hw} * log|I|)$,

where *hw* is the hypertree width and $|I|$ designates the input size. The smaller *hw* is, the faster we can compute a solution to a CSP, i.e., a small *hw* indicates a reduced solving complexity and vice versa. Hence finding the best decomposition is crucial for solving the constraint system.

Lemma 3 (Optimal hypertree decomposition) *A hypertree decomposition is **optimal** if its hypertree width is the smallest possible from all possible hypertree decompositions.*

Computing the best hypertree decomposition is NP hard. Most of the proposed algorithms compute an approximation of the best decomposition. For computing the hypertree decomposition and the hypertree width we relied on an implementation provided by [53] which employs the ***Bucket Elimination*** algorithm [81] found in Algorithm 11. Note that this algorithm is an approximation algorithm, i.e., it does not always generate the optimal hypertree decomposition with a minimal width. However, as reported in [8], the algorithm which performs the optimal decomposition is very time and space demanding and is therefore not suitable for practical use, and the Bucket Elimination algorithm in most cases provides better approximations than other known approximation algorithms.

The tree decomposition resulted from Algorithm 11 is a hypertree decomposition $\langle T, \chi, \psi \rangle$ if it satisfies the extra constraint:
*Let $\langle T : (B, E), \chi \rangle$ be the tree decomposition returned by the **Bucket_Elimination** algorithm, then for each $p \in B$, $\chi(p) \subseteq \psi(p)$.*

If a node $B_{v_i} \in B$ of the tree decomposition resulted after applying the **Bucket_Elimination** algorithm does not fulfill the above condition, the authors of [81] suggest adding hyperedges from the original hypergraph to the node until the additional property is satisfied.

For more information on hypertree decomposition and a comparison of different hypertree decomposition we refer the interested reader to [46]. The hypertree width is the number of constraints to be joined in order to obtain an acyclic CSP. A small hypertree width is an indicator for the tractability of a problem. The hypertree width is a number between 1 and the number of constraints. Hence, knowing the hypertree width of programs should also be an indicator for the computational complexity of the corresponding debugging problem, which is of practical interest.

This work is however limited to extensional or table constraints, i.e., for each constraint all allowed tuples are fully specified in a table. A solution to such a CSP is found by applying the join operator over constraints that share common variables. If after applying the join operator, there exist tables with 0 tuples, we say that the CSP has no solution.

Algorithm 11 Bucket_Elimination $(H = (V, HE), \sigma)$, given in [81]

Require: A hypergraph $H = (V, HE)$ and a variable ordering $\sigma = (v_1 ... v_n)$, $\bigcup_{i=1..n} v_i = V$.
Ensure: A tree decomposition $\langle T, \chi \rangle$.

1: let $B =$ and $E =$.
2: **for all** $v_i \in V$ **do**
3: Introduce an empty bucket B_{v_i} such that $(\chi(B_{v_i}) = \{\phi\})$
4: **end for**
5: Fill the buckets $B_{v_1} ... B_{v_n}$ such that:
6: **for all** $h \in HE$ **do**
7: Let $v \in h$ be the maximum vertex of h according to the ordering σ
8: $\chi(B_v) = \chi(B_v) \cup h$
9: **end for**
10: **for** $i = n...2$ **do**
11: Let $A = \chi B_{v_i} \{v_i\}$
12: Let $v_j \in A$ be the highest vertex smaller than v_i according to ordering σ
13: $\chi(B_{v_j}) = \chi(B_{v_j}) \cup A$
14: $E = E \cup (B_{v_i}, B_{v_j})$
15: **end for**
16: **return** $\langle (B, E), \chi \rangle$ where $B = B_{v_1} ... B_{v_n}$.

Hence, the use of hypertree width as a complexity measure was originally relevant for table or extensional constraints, i.e., for every relation of the constraint system, all allowed combination of values are explicitly specified in a table which has a column for each variable from the constraint's scope. For cycle-free hypergraphs this implies only semi join operation over the tables. Additionally, for a hypertree decomposition $< T, \chi, \psi >$, $T = (V, E)$, for all $v \in V$, with $\psi(v) \geq 2$ a natural join between the constraints is required.

Example 12. Given the constrains:
$C1 : a + b = c$
$C2 : d - c = e$
$0 < e < 2$
$D = \{0..2\}$
The extensional representation is:

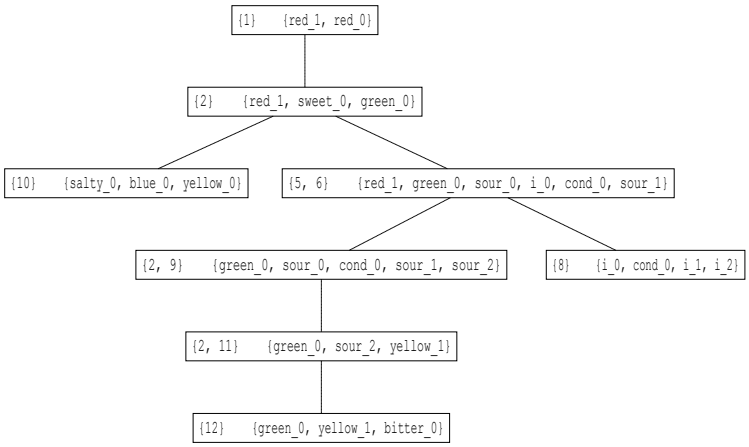

Figure 5.5.: The hypertree corresponding to the constraint system resulted from the program given in Fig. 5.1 for one loop iteration.

C1	a	b	c
1.	0	0	0
2.	1	0	1
3.	0	1	1
4.	1	1	2
5.	2	0	2
6.	0	2	2

C2	c	d	e
1.	0	0	0
2.	0	1	1
3.	1	1	0
4.	1	2	1
5.	0	2	2
6.	2	2	0

C1 $\wedge$ **C2** $\wedge$ **C3**	a	b	c	d	e
1.	0	0	0	1	1
2.	1	0	1	2	1
3.	0	1	1	2	1

□

There exists a tight relationship between the number of iterations used for unrolling a program's loops and the hypertree width. For example the hypertree decomposition of the hypergraph corresponding to the one loop iteration version of the program from Fig 5.1, results in an hypertree width of 2, see Fig. 5.5. Whereas the hypertree decomposition of the hypergraph corresponding to the two iterations version has a hypertree width of 3, see Fig. 5.4.

In the following we discuss the consequences of the debugging model in terms of complexity. In particular we are interested whether the hypertree width is bounded for such a model or not. In [107] the author proved that structured programs have a hypertree width of 6 in the worst case. Unfortunately, the result is based on considering only the control flow graph and not the data flow, which is sufficient for some tasks to be solved in compiler construction. The following theorem shows that the

result of [107] cannot be applied in the context of debugging where the control and data flow is of importance.

Theorem 3 (Hypertree width upper bound) *There is no constant upper-bound for the hypertree width of arbitrary programs.*

Proof: We prove this theorem indirectly. We assume that there is a constant value which serves as upper-bound for all programs and show that there is a class of programs where this assumption does not hold. Consider the class of programs that is obtained running the following algorithm for $n > 0$:

1. Let Π_n be a program comprising an empty block statement.
2. For $i = 1$ to n do:
   a) For $j = i + 1$ to n do:
      i. Add the statement
      $x_{j,i} = x_{i,i-1} + x_{j,i-1}$
      at the end of the block statement of program Π_n.
3. Return Π_n.

In this class, programs have n inputs and 1 output. Every variable depends on any other directly or indirectly via another variable. Hence, the hypertree width depends on the number of statements and there is no constant value, which contradicts our initial assumption. □

Note that there is always an upper-bound of the hypertree width of a particular program, which is given by the number of statements. However, the above theorem states that there is no constant which serves as upper-bound for all programs. What is missing is a clarification whether the number of nested if-statements after loop-unrolling has an influence on the hypertree width. If the hypertree width of a program comprising a *while*-statement depended on the number of considered iterations, then the complexity of debugging would heavily depend on the used test cases. Fortunately this is not the case as it is stated in the following theorem.

Theorem 4 (Maximal upper bound of a program) *Given an arbitrary program* Π *comprising at least one* while *statement, there always exists an upper bound on the hypertree width when the number of iterations increases.*

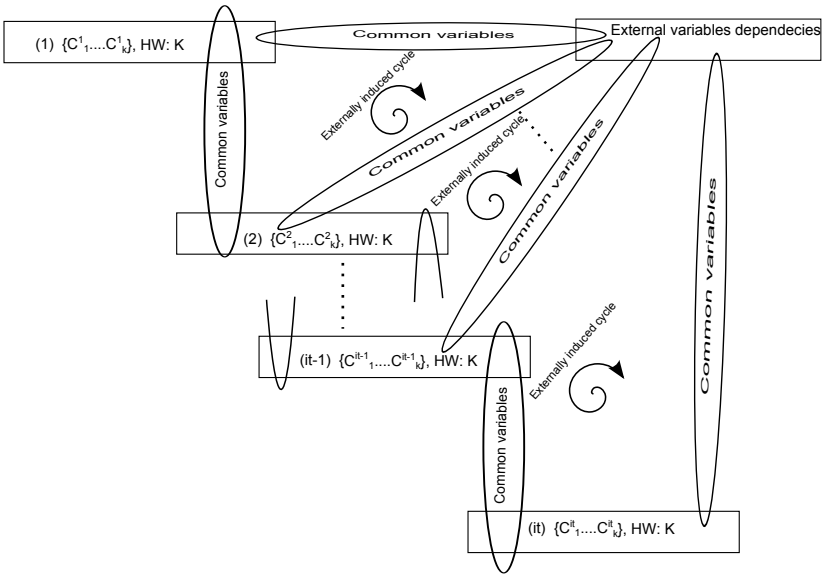

Figure 5.6.: The worst case-scenario of a partial hypertree-decomposition for an *it*-iterations loop unrolling (without the external dependencies)

Proof:

For this proof we make use of Fig. 5.6. We start by analyzing how an *while* structure affects the overall hyper tree width when unfold. Let's presume that $\{C_1...C_k\}$ are the constraints of the while block. Presume that in a worst case scenario, all constraints of the block have to be joined to remove all cycles from the hypergraph induced by $\{C_1...C_k\}$, i.e., the hypertree width of the resulted hypertree decomposition is k. Naturally the same hypertree width corresponds to the other iterations of the while loop. Basically if we have *it* iteration the hypertree induced by the unrolling of a loop has a depth of *it* and a worst case scenario hypertree width of k. Hence increasing the number of iterations always induces a ***constant hypertree width*** which is always $\leq$ than the number of statements, i.e., constraints, from the while block. In the worst case scenario, when all k constraints are joined, the unrolling of the while-structure is actually an acyclic hypergraph (as depicted in Fig. 5.6). Each iteration shares common variables only with its prior (except the first iteration) and the next iteration (except the last iteration). In this situation there exist no cycles induced by the internal structure of the hypergraph resulted from unrolling the *while* loop and, hence, no need to join any other constraints (see Fig. 5.6).

The next point to be tackled is analyzing the dependencies created between the external variables and the statements of the *while* block. Clearly these dependencies are the same for all iterations and

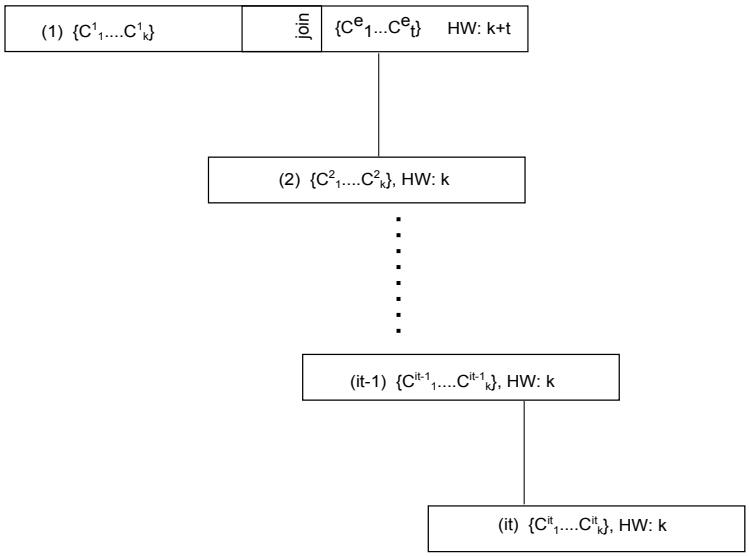

Figure 5.7.: The worst case-scenario of the hypertree-decomposition of the structure given in Fig. 5.6

induce external cycles in the hypergraph (see Fig. 5.6).

From the definition of the hypertree decomposition (Definition 30),we see that in order to remove the cycles created by the external variables, it is enough (worst-case scenario) to join the constraints of the first loop iteration with the external constrains. The resulted hypertree decomposition is given in Fig. 5.7 and has the hypertree width equal with the hypertree width of the first loop iteration. In the worst case scenario the resulted hypertree width is $(k+t)$, where k is the number of the original loop body statements and t is the total number of external constraints which influence the loop's body. The other iterations do not have any influence on the hypertree width, hence it is bounded by the behavior of the first iteration and the external dependencies.

□

In our original approach we used an extensional representation of our constraint system and used the $TREE*$ [103] solving algorithm for computing the fault candidates set for the faulty programs. The problem is that for larger programs the extensional representation of constraints is no longer feasible with respect to computation time and memory requirements.

As described in Chapter 3, we changed the modeling of our constraint system into the language of MINION. The MINION solver uses Partial Assignment Membership (PAM) propagators [47] together

with backtrack search for computing the solutions to a CSP. The question which we now answer is: *Can the hypertree width further be an indicator of the debugging complexity?*

Deciding if the resulted constraint system is tractable means in our case deciding if the PAM representation of the constraint system is tractable. From [47] we know that a PAM representation is tractable only if the following theorem holds:

Theorem 5 (PAM tractability [47]) *Given any list of structures H, generate the list of structures H' by removing from members of H all isolated vertices. Then the PAM representation of all multi-hypergraphs of H is tractable if and only if H' is of bounded arity and has bounded hypertree width.*

For our class of programs H is equal to the list of constraints scopes. We know that all constraints have bounded arity, i.e., the hyperedges and that the hypertree decomposition of the CSP's hypergraph always has a bounded hypertree width (Theorem 4). Hence the PAM representation of our constraint system, remains tractable.

We now try to identify the relationship between the complexity of solving the constraint system and the hypertree width. We know that in the case of the extensional representation this relationship is tight, e.g., a hypertree width of 6 is an indicator of a complex problem.

The relationship between the hypertree width and the running time for diagnosis has not been explored before. We discuss an experimental study using a number of programs in the next section.

5.3. Experimental Results

In order to answer the question whether the hypertree width is a good indicator of the debugging complexity given a constraint representation we conducted a set of experiments. We implemented the conversion process and the debugging algorithm. In our implementation we make use of the MINION constraint solver [40] to find solutions for the debugging problem. It is interesting to note that MINION does not rely on the join operator but on PAM propagators. Therefore, it is even more interesting to give an answer to the question whether the hypertree width, as a representative of the structural properties of the constraint system, still remains a good complexity indicator even for PAM propagators.

To empirically answer this question, we tested our approach over a set of Java programs ranging from 10 LOC up to 1,360 LOC respectively between 21 and 2,000 LOC in the SSA form. The

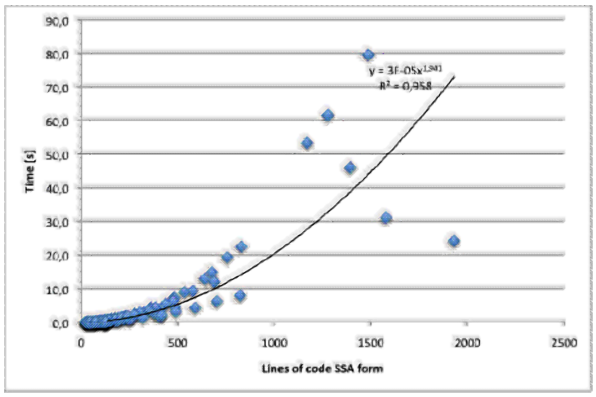

Figure 5.8.: Running time vs. the LOC for the SSA form

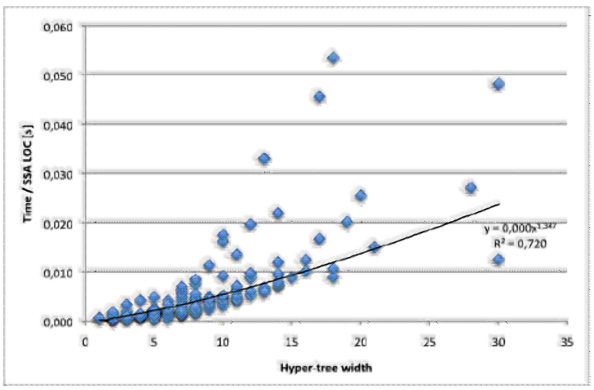

Figure 5.9.: Running time vs. HT width

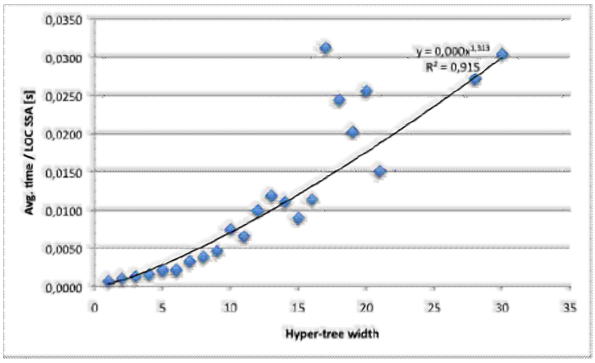

Figure 5.10.: Average running time vs. HT width

associated hypertree widths range from 2 up to 30. We implemented the conversion as described in Chapter 3 in the language of Java. For each program containing loops we compute different versions of loop elimination, ranging from 2 up to 21 iterations. Further, we compute the hypertree width using the decomposition tool from [53], which implies bucket elimination [81].

We performed all the experiments on a Intel Pentium Dual Core 2 GHz PC with 4 GB of RAM machine. Table 5.1 summarizes the obtained results. The given time is for reading the MINON files and computing all minimal cardinality diagnoses. In the experiments only single faults were considered.

From the data in Table 5.1 we obtained the Figures 5.8 and 5.9 that show the relation between the running time for computing all solutions and the lines of code (LOC) of the SSA form, and the hypertree width, respectively. It can be seen that in both cases there is a relationship but there is also a huge variance especially when the LOC or hypertree width increases. Note that the data indicates the LOC to be a better estimator of the running time than the hypertree width. However, in Figure 5.10 is depicted the average running time divided by the LOC of the SSA form depending on the hypertree width. In this case there is a very strong correspondence, showing that the hypertree width is a good indicator of the running time on average. Moreover, we also computed the correlation coefficient for the three cases:

Parameter 1	Parameter 2	Correlation coeff.
Time	LOC SSA	0.8401
Time/LOC SSA	Hypertree width	0.7426
Avg. Time/LOC SSA	Hypertree width	0.9016

We see that the best correlation is between the average time divided by the LOC of the SSA form and the hypertree width.

Regarding the applicability of the approach for debugging the obtained numbers showed that even programs of up to 2,000 LOC can be debugged in less than 1 1/2 minutes when using constraint solving. Hence, the approach can be effectively used for debugging methods in the context of object-oriented systems.

5.4. Conclusions

In this section we reflect on the objective to state whether there is a relationship between the hypertree width of the constraint representation of a program, and the running time of debugging. For this purpose, we described how a program can be compiled into a constraint representation and discussed the obtained empirical results. From the obtained results we are able to say that programs up to 2,000 LOC can be debugged in less than 1 1/2 minute. Moreover, with a correlation coefficient of 0.9 we can state that there is a high correlation between the hypertree width and the average debugging running time divided by the LOC. In addition to this result, we obtained a strong correlation between the LOC and the running time. Hence, both measures are good for estimating the running time and the average running time of debugging. We are now able to summarize all findings regarding the influence of hypertree width to debugging: (1) there is an upper bound of the hypertree width with respect to the number of iterations considered to obtain the constraint model, (2) the hypertree width correlates with the average running time divided by LOC, and (3) the latter correlation is higher than the correlation between the running time and LOC.

Name	It	$\mathbf{Var_{\Pi}}$	$\mathbf{LOC_{\Pi}}$	Inputs	Outputs	NoWhiles	$\mathbf{LOC_{SSA}}$	|CO|	$\mathbf{Var_{CO}}$	Time	|Diag|	HW
Binomial	1	37	189	6	3	16	233	329	244	1.544	5	13
	⋯						⋯	⋯	⋯	⋯	⋯	⋯
	4						1277	2261	1480	61.589	5	30
BinSearch	1	8	37	3	1	1	44	54	48	0.031	3	3
	⋯						⋯	⋯	⋯	⋯	⋯	⋯
	21						484	1014	748	4.633	213	14
ComplexHypertree	1	9	39	4	1	1	41	44	31	0.031	5	3
	⋯						⋯	⋯	⋯	⋯	⋯	⋯
	21						421	804	431	4.492	32	18
Data	1	5	40	1	1	1	34	33	32	0.046	11	3
	⋯						⋯	⋯	⋯	⋯	⋯	⋯
	21						254	513	392	1.263	0	9
DivATC	1	5	21	2	1	1	27	23	23	0.015	3	2
	⋯						⋯	⋯	⋯	⋯	⋯	⋯
	21						127	223	143	0.546	68	8
GcdATC	1	6	35	2	1	1	38	40	33	0.015	4	3
	⋯						⋯	⋯	⋯	⋯	⋯	⋯
	21						258	460	293	1.139	8	9
Hamming	1	15	77	2	1	5	85	97	79	0.171	7	3
	⋯						⋯	⋯	⋯	⋯	⋯	⋯
	5						1169	2117	1331	53.384	71	17
MultATC	1	5	16	2	1	1	21	14	13	0.015	3	1
	⋯						⋯	⋯	⋯	⋯	⋯	⋯
	21						121	214	133	0.281	4	8
MultV2ATC	1	6	20	2	1	2	28	26	21	0.015	10	2
	⋯						⋯	⋯	⋯	⋯	⋯	⋯
	16						1393	2741	1656	46.051	14	13
RandomATC	1	8	53	3	1	4	63	90	73	0.031	3	5
	⋯						⋯	⋯	⋯	⋯	⋯	⋯
	16						1578	3105	1888	31.106	8	12
SumATC	1	5	18	3	1	2	22	14	14	0.015	3	1
	⋯						⋯	⋯	⋯	⋯	⋯	⋯
	21						122	214	134	0.421	7	6
SumPowers	1	11	36	3	1	2	46	41	40	0.046	6	2
	⋯						⋯	⋯	⋯	⋯	⋯	⋯
	16						1486	2621	1750	79.576	872	18
whileTest	1	16	94	4	1	3	77	81	67	0.141	12	2
	⋯						⋯	⋯	⋯	⋯	⋯	⋯
	7						275	477	283	2.324	28	8
tcas01	1	48	125	12	1	0	125	98	132	0.265	28	5
tcas22	1	48	125	12	1	0	125	98	132	0.250	8	5
tcas41	1	48	125	12	1	0	125	98	132	0.265	27	5
IscasC432V1	1	199	412	1	1	0	415	633	744	2.121	5	10
IscasC432V2	1	199	412	1	1	0	415	633	744	2.091	3	10
IscasC432V3	1	199	412	1	1	0	415	633	744	2.153	7	10
replaced	1	192	1256	3	1	11	1929	4236	2420	24.192	144	30

Table 5.1.: Each program **Name**, has associated a number of iterations **It**, the number of variables $\mathbf{Var_{\Pi}}$its size given in lines of code $\mathbf{LOC_{\Pi}}$, the number of inputs **Inputs**, number of outputs **Outputs**, the number of contained while structures **NoWhiles** the size of its SSA representation given as lines of code$\mathbf{LOC_{SSA}}$, the number of MINION constraints |**CO**|, the number of MINION variables over which the constraint system is defined $\mathbf{Var_{CO}}$, the time in which MINION found all solutions **Time**, the number of diagnosis candidates |**Diag**| and the hypertree width associated to the CSP's hypergraph **HW**.

Chapter 6

My Conclusions and Future Work

"*Nothing in the world that's worth having comes easy*" - Dr. Kelso , Scrubs TV Series.

In this last chapter of my thesis I took the liberty of being a little more informal about my work. Each chapter of my thesis already contains a section comprising the conclusions of the work presented in it. So, please refer to this if you would like to know more about a specific topic presented in this thesis.

What I intend to do in this part of my thesis is to present an overview of my work together with my view about the future of debugging.

In the spirit of how I began, I will end up my thesis, by telling a short story.

In one of my lectures I once ask my students (third year bachelors) how many of them used an automated debugger. The answer was, as expected, none. Nobody tried to use one of the automated debugging tools. Hmmm... no need of debugging perhaps? I asked them another question regarding debugging, but this time I wanted to know how many of them used a symbolic debugger so far. The answer was: less than 50%. The last question which I asked was regarding how many of them used (at least once) the "insert print statements" -method to identify the fault. Almost all of them admitted to do so when confronted with a bug.

The truth is that, there exists no coding without the need to debug. So, why don't we use more advanced techniques but rely instead on the "old fashion way" to isolate the bug? I think this question

should puzzle all of us, from the debugging community. We have a lot of model based debugging algorithms, spectrum based algorithms and lately mutation based debugging algorithms but no real industrial tool.

The problem is that no unification between the different approaches exists. We have the algorithms, the scientific-implementations (with quite a lot of restrictions on the input program), but there is no real desire to implement state of the art tools which can be used in the more general case for debugging high level programing languages, like Java or C++. A combination of the available techniques would be more than appropriate and would definitely lead to an improvement of the results. Even more, it was proven that by combining different debugging approaches the scalability problem can be solved.

For example, in this thesis we showed that by combining program mutation with constraint representation and distinguishing test cases, the efficiency of the algorithm increases. The same happens if we integrate specification knowledge in our approach. However the main drawback of our algorithm is with respect to scalability. We can be efficient (small computation times) only for programs of relatively small size (up to thousand lines of code). Hence we can use our approach to debug programs at method level or for debugging small embedded software systems. But if we combine our method with, let's say, spectrum based technique, and focus only on the components that are high ranked we could use it to debug larger programs.

So, as far as I am concerned, there are two "moral" issues which I would still like to address in a future research:

- **Industry.** trying to make my implementation of the algorithm to work, not only under a set of restriction but also on the general case, e.g., object oriented. In addition I would really like to see how different techniques would work together. For the moment I am concentrating on spectrum based and specification knowledge for improving the scalability of my approach. *A good example in software debugging* is Microsoft that delivered its Microsoft Visual Studio 2010 with an integrated spectrum based debugger: Holmes. In *hardware diagnosis* there exists the RODON tool which proved to be quiet efficient in identifying the faulty components of a system's model (based on its symptoms, i.e., observations).
- **Communication.** The debugging community is becoming smaller and smaller with every year, but still it remains a "one man's game". It would be interesting if we could form some sort of exchange networking, where we could develop together algorithms and, why not, strategies for making our techniques as close as possible to the user requirements. Since 2009 there exists

within the Workshop on Principles of Diagnosis (DX), a competition which tries to unify and compare the available diagnosis techniques. But this ”only” after 20 years of DX existence. This year, DX was even co-located with an industry-closed conference. So perhaps in the near future more steps would be made by the community in this direction.

6.1. Trivia

This part of my theses gathers personal thoughts about the three years that I spent at TU-Graz. This period was both for my personal and professional experience of most importance (although very often my personal life was my professional life or vice-versa).

As I started the project I did not knew much about program debugging, and asked myself back then (like I, after two years, asked my students) if it is possible for a tool to really debug a program and also be feasible. When I started working on the project we used the TREE* algorithm for debugging programs. It worked perfectly fine for boolean programs but was a ”horror” when it dealt with large domains variables. There was however a good part to it. It gave us the idea of using the structural properties of the program's constraint system hypergraph to measure the complexity of debugging. The higher the number of cycles in the graph (measured by the hypertree width) the higher the time for debugging (due to backtracking). They correlated more than perfect, that is for the extensional representation. But debugging was still hard to perform, e.g., it took us, for large domain variables, more than 20 minutes to debug a 20 lines of code program. Surely that was not feasible.

Next, we tried to use a state of the art constraint solver, together with a new encoding of the debugging problem, and I came up with MINION. It was proven to be a very efficient and scalable solution. We could finally debug large methods written in Java (I have made also an Eclipse plug-in version of it). Afterwards we tried to integrate annotations, which was nice but not very practical and last we integrated program mutation together with distinguishing test cases, which I personally found to be very promising. We tried again to see if the structural representation of the programs is a good indicator of the tractability of the debugging problem. To some extend, as I showed in this work, it still is. However, not as accurate, as in the case of using the extensional representation of the constraint system.

Another interesting activity during my stay at TU-Graz, was teaching and working with the students. By teaching two bachelor-lectures and one master-lecture I understood that the best ideas come

not from a ”big-bang”, which you have in the middle of night (although I do not exclude the possibility of this happening), but by asking yourself or be asked the right questions. I am not saying that all students had the right questions (there exist no silly questions, only silly answers), but sometimes there was that ”one question” that help me better understand a problem or a topic, and finally help me come up with the right answer. Hence, never be afraid of asking yourself or be asked questions (as long as the question is well intended) and most important (also a lesson learned when I taught) do not be afraid of the ”I don’t know” answer. It’s no crime to say this, that is, as long as you will try to know the answer by the end of the day.

So, my advice to someone who thinks about doing a PhD, would be: don’t start a PhD because you want to be a ”Doctor”; it does not worth it. Start it in the pursuit of finding answers to questions which bring something, not just to yourself but also to others. Do not try to *slip between*, but walk straight and you will see that in the end you will gain far more satisfaction for your achievements, far more knowledge and far more expertise, as in the case when you would have ”cheated” the system just to get the title. Cause a title without the knowledge that it presumes, becomes in reality an embarrassment...

Last, I should ask myself the question: ”Did it worth it?”. Absolutely. As long as you are doing it with passion for science I would say *it’s worth it*. Being paid to learn is no bad deal. And, writing a PhD, implies a lot of reading, sometimes traveling, and always learning. For me it was a period of important changes both in the way I saw myself and in the way I saw the world. A period which brought me a lot (professionally and -especially- personally), which opened my mind and offered me the chance to experiment things (again professionally and personally) that otherwise I don’t think I would have had the chance to experiment. For all this, I am grateful and say, if I were to go back in time, I would do it all over again...

List of Theorems and Definitions

List of Definitions

Theorems, Corollaries, and Lemmas

Bibliography

[1] A. GRIESMAYER, R. B. AND COOK., B. 2006. Repair of boolean programs with an application to C. *Proc. 18th Conference on Computer Aided Verification (CAV'06)*, 358–371. (Cited on page 20.)

[2] A. J. OFFUTT, A. LEE, G. R. R. U. AND ZAPF, C. 1996. An experimental determination of sufficient mutation operators. *ACM Transactions on Software Engineering Methodology 5*, 99–118. (Cited on page 113.)

[3] ABREU, R., ZOETEWEIJ, P., AND VAN GEMUND, A. J. 2006. On the accuracy of spectrum-based fault localization. In *Proceedings TAIC PART'07*. IEEE, 89–98. (Cited on pages 25 and 26.)

[4] ABREU, R., ZOETEWEIJ, P., AND VAN GEMUND, A. J. 2008. An observation-based model for fault localization. In *WODA'08, Procedings of the 6th Workshop on Dynamic Analysis*, B. Liblit and A. Rountev, Eds. ACM Press, Seattle, WA, USA, 64–70. (Cited on page 25.)

[5] ABREU, R., ZOETEWEIJ, P., AND VAN GEMUND, A. J. 2009. Spectrum-based multiple fault localization. In *Proc. IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 88–99. (Cited on page 25.)

[6] ALICE. Alice programming language. http://www.ps.uni-saarland.de/alice/papers.html. (Cited on page 23.)

[7] ALUR, R., COURCOUBETIS, C., AND YANNAKAKIS, M. 1995. Distinguishing tests for nondeterministic and probabilistic machines. In *STOC '95: Proceedings of the twenty-seventh annual ACM symposium on Theory of computing*. ACM, 363–372. (Cited on page 102.)

[8] ARTAN DERMAKU, TOBIAS GANZOW, G. G. B. M. N. M. M. S. 2005. Heuristic methods for hypertree decompositions. *DBAI-TR-2005-53*. (Cited on page 129.)

[9] ATKINSON, R. joke adapted from rowan atkinson. http://www.berniecode.com/blog/. (Cited on page 9.)

[10] B. PEISCHL, N. R. AND WOTAWA, F. 2008. Model-based reasoning with multiple test cases and its application to debugging. *In Proceedings of the 19th International Workshop on Principles of Diagnosis DX 2008*, 315–323. (Cited on pages 40 and 82.)

[11] BAAH, G. K., PODGURSKI, A., AND HARROLD, M. J. 2008. The probabilistic program dependence graph and its application to fault diagnosis. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 189–200. (Cited on page 23.)

[12] BINKLEY, D. AND HARMAN, M. 2004. A survey of empirical results on program slicing. In *Advances in Software Engineering – Advances in Computers Vol. 62*, M. Zelkowitz, Ed. Academic Press Inc., 106–172. See also citeseer.ist.psu.edu/661032.html. (Cited on page 23.)

[13] BLOG, G. T. Google testing blog. http://googletesting.blogspot.com/. (Cited on page 17.)

[14] BODLAENDER, H. L. 1993. A linear time algorithm for finding tree-decompositions of small treewidth. In *STOC '93: Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*. ACM, New York, NY, USA, 226–234. (Cited on page 122.)

[15] BOND, G. W. 1994. Logic Programs for Consistency-Based Diagnosis. Ph.D. thesis, Carleton University, Faculty of Engineering, Ottawa, Canada. (Cited on page 29.)

[16] BOND, G. W. AND PAGUREK, B. 1994. A Critical Analysis of "Model-Based Diagnosis Meets Error Diagnosis in Logic Programs". Tech. Rep. SCE-94-15, Carleton University, Dept. of Systems and Computer Engineering, Ottawa, Canada. (Cited on page 29.)

[17] BRANDIS, M. M. AND MÖSSENBÖCK, H. 1994. Single-pass generation of static assignment form for structured languages. *ACM TOPLAS 16(6)*, 1684–1698. (Cited on page 49.)

[18] CEBALLOS, R., GASCA, R. M., VALLE, C. D., AND BORREGO, D. 2006. Diagnosing errors in dbc programs using constraint programming. *Lecture Notes in Computer Science 4177*, 200–210. (Cited on pages 29, 43, 83, 96, and 122.)

[19] CHOCO TEAM, T. 2008. choco: an open source java constraint programming library. In *In: The Third International CSP Solver Competition*. 31–40. (Cited on page 54.)

[20] CHOMSKY, N. 1956. Three models for the description of language. *IRE Transactions on Information Theory*, 113124. (Cited on page 37.)

[21] CLEVE, H. AND ZELLER, A. 2005. Locating causes of program failures. In *Proc. 27th International Conference on Software Engineering (ICSE 2005)*. St. Louis, Missouri, USA. (Cited on page 28.)

[22] COLLAVIZZA, H. AND RUEHER, M. 2007. Exploring different constraint-based modelings for program verification. In *In* Principles and Practice of Constraint Programming (CP 2007). Providence, RI, USA, 49–63. (Cited on pages 16, 42, 54, and 96.)

[23] CONSOLE, L., FRIEDRICH, G., AND DUPRÉ, D. T. 1993. Model-based diagnosis meets error diagnosis in logic programs. In *Proceedings* 13^{th} *International Joint Conf. on Artificial Intelligence*. Chambery, 1494–1499. (Cited on pages 29 and 31.)

[24] CYTRON, R., FERRANTE, J., ROSEN, B. K., WEGMAN, M. N., AND ZADECK, F. K. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM TOPLAS 13,* 4, 451–490. (Cited on pages 49 and 51.)

[25] DALAL, S. R., JAIN, A., KARUNANITHI, N., LEATON, J. M., LOTT, C. M., PATTON, G. C., AND HOROWITZ, B. M. 1999. Model-based testing in practice. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*. ACM, New York, NY, USA, 285–294. (Cited on page 31.)

[26] DALLMEIER, V., L. C. AND ZELLER, A. 2005. Lightweight defect localization for java. *Springer-Verlag, Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP05) 3586*, 528550. (Cited on pages 27 and 28.)

[27] DAN HAO, LINGMING ZHANG, L. Z. J. S. AND MEI, H. 2009. Vida: Visual interactive debugging. *Proceedings of the 2009 IEEE 31st Internationa Conference on Software Engineering*, 583–586. (Cited on page 28.)

[28] DAVIS, R. 1984. Diagnostic reasoning based on structure and behavior. *Artificial Intelligence 24*, 347–410. (Cited on page 20.)

[29] DE KLEER, J. AND WILLIAMS, B. C. 1987. Diagnosing multiple faults. *Artificial Intelligence 32,* 1, 97–130. (Cited on pages 20 and 31.)

[30] DEBROY, V. AND WONG, W. E. 2010. Using mutation to automatically suggest fixes for faulty programs. In *Third International Conference on Software Testing, Verification and Validation (ICST 2010)*. IEEE. (Cited on pages 106 and 117.)

[31] DECHTER, R. 2003. *Constraint Processing*. Morgan Kaufmann. (Cited on pages 54, 76, and 127.)

[32] DEMILLO, R. A. AND OFFUTT, A. J. 1991. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering 17(9)*, 900–910. (Cited on page 96.)

[33] DEMILLO, R. A., PAN, H., AND SPAFFORD, E. H. 1996. Critical slicing for software fault localization. In *International Symposium on Software Testing and Analysis (ISSTA)*. 121–134. (Cited on page 23.)

[34] DUCASSÉ, M. 1993. A pragmatic survey of automatic debugging. In *Proceedings of the 1st International Workshop on Automated and Algorithmic Debugging, AADEBUG '93*. Springer LNCS 749. 1–15. (Cited on page 22.)

[35] DWYER, M. B., HATCLIFF, J., HOOSIER, M., RANGANATH, V., ROBBY, AND WALLENTINE, T. 2006. Evaluating the effectiveness of slicing for model reduction of concurrent object-oriented programs. In *Proceedings TACAS*. Springer LNCS 3920, 73–89. (Cited on page 23.)

[36] ECLIPSE. The ibm eclipse project. http://www.eclipse.org/. (Cited on page 64.)

[37] EDMUND M. CLARKE, O. G. AND LONG., D. E. 1994. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems 16(5)*, 15121542. (Cited on page 16.)

[38] FATTAH, Y. E. AND DECHTER, R. 1995. Diagnosing tree-decomposable circuits. In *Proceedings* 14^{th} *International Joint Conf. on Artificial Intelligence*. 1742 – 1748. (Cited on page 11.)

[39] FRIEDRICH, G., STUMPTNER, M., AND WOTAWA, F. 1999. Model-based diagnosis of hardware designs. *Artificial Intelligence 111,* 2 (July), 3–39. (Cited on pages 20 and 29.)

[40] GENT, I. P., JEFFERSON, C., AND MIGUEL, I. 2006. Minion: A fast, scalable, constraint solver. *17th European Conference on Artificial Intelligence ECAI-06*. (Cited on pages 54, 65, 97, and 135.)

[41] GOTLIEB, A., BOTELLA, B., AND RUEHER, M. 1998. Automatic test data generation using constraint solving techniques. In *In Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. (Cited on pages 42, 54, 96, and 104.)

[42] GOTLIEB, A., BOTELLA, B., AND RUEHER, M. 2000. A clp framework for computing structural test data. *In Proceedings of the International Conference on Computational Logic (CL) Springer LNAI 1861*, 399–413. (Cited on pages 96 and 104.)

[43] GOTTLOB, G., LEONE, N., AND SCARCELLO, F. 1999a. A comparison of structural csp decomposition methods. In *Proceedings* 16^{th} *International Joint Conf. on Artificial Intelligence*. Stockholm, Sweden, 394–399. (Cited on pages 122 and 128.)

[44] GOTTLOB, G., LEONE, N., AND SCARCELLO, F. 1999b. Hypertree Decomposition and Tractable Queries. In *Proc. 18th ACM SIGACT SIGMOD SIGART Symposium on Principles of Database Systems (PODS-99)*. Philadelphia, PA, 21–32. (Cited on pages 122 and 127.)

[45] GOTTLOB, G., LEONE, N., AND SCARCELLO, F. 1999c. On Tractable Queries and Constraints. In *Proc. 12th International Conference on Database and Expert Systems Applications DEXA 2001*. Florence, Italy. (Cited on pages 122 and 127.)

[46] GOTTLOB, G., LEONE, N., AND SCARCELLO, F. 2000. A comparison of structural CSP decomposition methods. *Artificial Intelligence 124,* 2 (December), 243–282. (Cited on pages 122, 127, and 129.)

[47] GREEN, M. J. AND JEFFERSON, C. 2008. Structural tractability of propagated constraints. In *CP '08: Proceedings of the 14th international conference on Principles and Practice of Constraint Programming*. Springer-Verlag, Berlin, Heidelberg, 372–386. (Cited on pages 134, 135, and 147.)

[48] GUPTA, N., HE, H., ZHANG, X., AND GUPTA, R. 2005. Locating faulty code using failure-inducing chops. In *Automated Software Engineering (ASE)*. 263–272. (Cited on pages 6, 23, and 123.)

[49] HAILPERN, B. AND SANTHANAM, P. 2002. Software debugging, testing, and verification. *IBM Systems Journal 41(1),* 412. (Cited on pages iii and v.)

[50] HAMLET, R. G. 1977. Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering 3(4)*, 279–290. (Cited on page 102.)

[51] HEBRARD, E. 2008. Mistral, a constraint satisfaction library. In *In Proceedings of the Third International CSP Solver Competition*. (Cited on page 54.)

[52] HOARE, C. A. R. 1969. An axiomatic basis for computer programming. *Communications of the ACM 12(10)*, 576–583. (Cited on pages 83 and 85.)

[53] HTTP://WWW.DBAI.TUWIEN.AC.AT/PROJ/HYPERTREE/INDEX.HTML. (Cited on pages 129 and 137.)

[54] ILOG. Ilog constraint solver. http://www-01.ibm.com/software/websphere/ilog-migration/. (Cited on page 54.)

[55] JACKSON, D. 2006. *Software abstractions: logic, language, and analysis*. MIT Press. (Cited on page 43.)

[56] JACOP. Jacop constraint solver. http://www.jacop.eu/. (Cited on page 54.)

[57] JONES, J. A. AND HARROLD, M. J. 2005. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings ASE'05*. ACM Press, 273–282. (Cited on pages 25 and 28.)

[58] KAMKAR, M. 1995. An overview and comparative classification of program slicing techniques. *J. Systems Software 31*, 197–214. (Cited on page 23.)

[59] KAMKAR, M. 1998. Application of program slicing in algorithmic debugging. *Information and Software Technology 40*, 637–645. (Cited on page 23.)

[60] KO, A. J. AND MYERS, B. A. 2004. Designing the whyline: a debugging interface for asking questions about program behavior. *In CHI 04: Proceedings of the SIGCHI conference on Human factors in computing systems*, 151158. (Cited on page 23.)

[61] KÖB, D. AND WOTAWA., F. 2006. Fundamentals of debugging using a resolution calculus. *Fundamental Approaches to Software Engineering (FASE'06) 3922*. (Cited on page 20.)

[62] KOREL, B. AND LASKI, J. 1988. Dynamic Program Slicing. *Information Processing Letters 29*, 155–163. (Cited on page 22.)

[63] KRINKE, J. 2004. Advanced slicing of sequential and concurrent programs. In *20th International Conference on Software Maintenance*. IEEE. (Cited on page 23.)

[64] KRINKE, J. AND SNELTING, G. 1998. Validation of measurement software as an application of slicing and constraint solving. *Information and Software Technology 40*, 661–675. (Cited on page 23.)

[65] KUSUMOTO, S., NISHIMATSU, A., NISHIE, K., AND INOUE, K. 2002. Experimental evaluation of program slicing for fault localization. *Empirical Software Engineering 7*, 49–76. (Cited on page 23.)

[66] LIVER, B. 1993. Repair of communication systems by working around failures. In *Proceedings of the Fourth International Workshop on Principles of Diagnosis*. Aberystwyth, UK, 270–277. (Cited on page 29.)

[67] LIVER, B. 1994. Modeling software systems for diagnosis. In *Proceedings of the Fifth International Workshop on Principles of Diagnosis*. New Paltz, NY, 179–184. (Cited on page 29.)

[68] M. NICA, S. N. AND WOTAWA, F. 2010. Does testing help to reduce the number of potentially faulty statement in debugging. *Proceedings of The Testing: Academic and Industrial Conference - Practice and Research Techniques TAIC-PART 2010*. (Cited on pages 13, 17, and 82.)

[69] MATEIS, C., STUMPTNER, M., WIELAND, D., AND WOTAWA, F. 2000. Model-Based Debugging of Java Programs. In *Proceedings of the 4th International Workshop on Automated and Algorithmic Debugging, AADEBUG '00*. Munich, Germany. (Cited on pages 29 and 31.)

[70] MATEIS, C., STUMPTNER, M., AND WOTAWA, F. 2000. Modeling Java Programs for Diagnosis. In *Proceedings of the European Conference on Artificial Intelligence (ECAI)*. Berlin, Germany. (Cited on page 20.)

[71] MAYER, W. 2007. Static and hybrid analysis in model-based debugging. *PhD Thesis, School of Computer and Information Science University of South Australia*. (Cited on pages 10 and 77.)

[72] MAYER, W., ABREU, R., STUMPTNER, M., AND VAN GEMUND, A. J. 2009. Prioritising model-based debugging diagnostic reports. In *Proceedings of the International Workshop on Principles of Diagnosis (DX)*. (Cited on pages 25 and 117.)

[73] MAYER, W. AND STUMPTNER, M. 2003. Model-based debugging using multiple abstract models. *Proceedings of the 5th International Workshop on Automated and Algorithmic Debugging AADEBUG-03*, 55–70. (Cited on pages 10 and 76.)

[74] MAYER, W., STUMPTNER, M., WIELAND, D., AND WOTAWA, F. 2002a. Can ai help to improve debugging substantially? debugging experiences with value-based models. In *Proceedings of the European Conference on Artificial Intelligence (ECAI)*. IOS Press, Lyon, France, 417–421. (Cited on page 20.)

[75] MAYER, W., STUMPTNER, M., WIELAND, D., AND WOTAWA, F. 2002b. Towards an integrated debugging environment. In *Proceedings of the European Conference on Artificial Intelligence (ECAI)*. IOS Press, Lyon, France, 422–426. (Cited on page 29.)

[76] MCILRAITH, S. AND REITER, R. 1992. On tests for hypothetical reasoning. In *Readings in Model-Based Diagnosis*. Morgan Kaufmann, 89–96. (Cited on page 102.)

[77] MEYER, B. 1997. *Object-Oriented Software Construction, 2nd edn.* OSE Press. (Cited on page 89.)

[78] MIHAI NICA, M. I. AND WOTAWA, F. 2009. Representing program debugging as constraint satisfaction problem. *Nordic Workshop on Programming Theory'09*. (Cited on pages 13 and 32.)

[79] MIKE BARNETT, K. RUSTAN, W. S. 2004. The spec# programming system: An overview. *LNCS Springer 3362*. (Cited on page 16.)

[80] MINION. 2009. The minion constraint solver. http://minion.sourceforge.net. (Cited on pages 93 and 97.)

[81] MUSLIU, N. AND SCHAFHAUSER, W. 2007. Genetic algorithms for generalised hypertree decompositions. *European Journal of Industrial Engineering 1(3)*, 317 – 340. (Cited on pages 127, 129, 130, 137, and 146.)

[82] NAISH, H. J. L. L. AND RAMAMOHANARAO, K. 2010. A model for spectra-based software diagnosis. *ACM Transactions on Software Engineering and Methodology*. (Cited on pages 27 and 28.)

[83] NICA, M., PEISCHL, B., AND WOTAWA, F. 2008. A constraint model for automated deployment of automotive control software. In *SEKE*. 899–904. (Cited on page 54.)

[84] NICA, M., WEBER, J., AND WOTAWA, F. 2008. How to debug sequential code by means of constraint representation. *19th International Workshop on Principles of Diagnosis (DX-08)*. (Cited on pages 13, 29, 32, 43, 83, and 122.)

[85] NICA, M., WEBER, J., AND WOTAWA, F. 2009. On the use of specification knowledge in program debugging. *20th International Workshop on Principles of Diagnosis (DX-09)*. (Cited on pages 10, 13, 29, 82, and 122.)

[86] NICA, M. AND WOTAWA, F. 2008. From constraint representations of sequential code and program annotations to their use in debugging. In *Proceeding of the 2008 conference on ECAI 2008*. IOS Press, Amsterdam, The Netherlands, The Netherlands, 797–798. (Cited on pages 12 and 82.)

[87] NIELSON, H. R. AND NIELSONL, F. 1992. Semantics with applications : A formal introduction. *John Wiley & Sons Chichester*. (Cited on page 16.)

[88] PATTON, R. 2006. *Software Testing (2nd Edition)*. Sams Publishing. (Cited on page 16.)

[89] PEISCHL, B., NICA, M., ZANKER, M., AND SCHMID, W. 2009. Recommending effort estimation methods for software project management. In *Web Intelligence/IAT Workshops*. 77–80. (Cited on page 54.)

[90] R. DEMILLO, R. L. AND SAYWARD, F. 1978. Hints on test data selection: Help for the practicing programmer. *IEEE Computer 11(4)*, 34–41. (Cited on page 102.)

[91] RANGANATH, V. P., AMTOFT, T., BANERJEE, A., HATCLIFF, J., AND DWYER, M. B. 2006. A new foundation for control dependence and slicing for modern program structures. Tech. Rep. #2004-8, Kansas State University. Part of this work was published in the Proceedings of European Symposium on Programming (ESOP) 2005. (Cited on page 23.)

[92] REITER, R. 1987. A theory of diagnosis from first principles. *Artificial Intelligence 32,* 1, 57–95. (Cited on pages 11, 12, 20, 29, 31, 40, and 61.)

[93] RESEARCH, M. 2009. Holmes: Automated statistical debugging for .net. http://research.microsoft.com/en-us/projects/holmes. (Cited on pages 25 and 28.)

[94] ROTHERMEL, G. AND HARROLD, M. J. 1990. Empirical studies of a safe regression test selection technique. *IEEE Transactions on Software Engineering 24,* 6, 401–419. Siemens test suite is available at http://www-static.cc.gatech.edu/aristotle/Tools/subjects/. (Cited on page 76.)

[95] S. STABER, B. J. AND BLOEM., R. 2005. Finding and fixing faults. *Springer-Verlag,Proc. 13th Conference on Correct Hardware Design and Verification Methods 3725*, 35–49. (Cited on page 20.)

[96] SHAPIRO, E. 1983. *Algorithmic Program Debugging*. MIT Press, Cambridge, Massachusetts. (Cited on pages 22, 23, and 29.)

[97] SLAM. 2002. Microsoft SLAM project. http://research.microsoft.com/en-us/projects/slam/. (Cited on page 16.)

[98] STEGMANN, R., KOCH, M., LACHER, M., LECKNER, T., AND RENNEBERG, V. 2003. Generating personalized recommendations in a model-based product configurator system. In *in IJCAI*. (Cited on page 31.)

[99] STRUSS, P. 1994. Testing physical systems. In *AAAI '94: Proceedings of the twelfth national conference on Artificial intelligence (vol. 1)*. American Association for Artificial Intelligence, 251–256. (Cited on page 102.)

[100] STUMPTNER, M. AND WOTAWA, F. 1998a. Mbd research activities at vienna university of technology. In *Proceedings of the ECAI-98 Workshop W5 'Model-based systems and qualitative reasoning'*. Brighton, UK. (Cited on page 22.)

[101] STUMPTNER, M. AND WOTAWA, F. 1998b. Model-based reconfiguration. In *Proceedings Artificial Intelligence in Design*. Lisbon, Portugal. (Cited on page 54.)

[102] STUMPTNER, M. AND WOTAWA, F. 1999. Debugging Functional Programs. In *Proceedings 16th International Joint Conf. on Artificial Intelligence*. Stockholm, Sweden, 1074–1079. (Cited on pages 20 and 29.)

[103] STUMPTNER, M. AND WOTAWA, F. 2001. Diagnosing tree-structured systems. *Artificial Intelligence 127,* 1, 1–29. (Cited on pages 11 and 134.)

[104] STUMPTNER, M. AND WOTAWA, F. 2003. Coupling CSP decomposition methods and diagnosis algorithms for tree-structured systems. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI-03)*. Acapulco, Mexico, 388–393. (Cited on page 11.)

[105] TAI, K.-C. AND SU, H.-K. 1987. Test generation for Boolean expressions. *Proceedings of the Eleventh Annual International Computer Software and Applications Conference (COMPSAC) 3(4)*, 278–284. (Cited on page 104.)

[106] TAMURA, N. AND BANBARA, M. 2008. Sugar: A csp to sat translator based on order encoding. In *In Proceedings of the Second International CSP Solver Competition*. editors, M.R.C.van Dongen, Christophe Lecoutre, and Olivier Roussel, 65–69. (Cited on page 54.)

[107] THORUP, M. 1998. All structured programs have small tree-width and good register allocation. *Inf. Comput. 142,* 2, 159–181. (Cited on pages 121, 131, and 132.)

[108] TIP, F. 1995. A Survey of Program Slicing Techniques. *Journal of Programming Languages 3,* 3 (Sept.), 121–189. (Cited on page 23.)

[109] TOM JANSSEN, R. A. AND VAN GEMUND, A. J. 2009. Zoltar: a spectrum-based fault localization tool. *In SINTER '09: Proceedings of the 2009 ESEC/FSE workshop on Software integration and evolution @ runtime*, 2330. (Cited on page 28.)

[110] TRETMANS, J. 1996. Test generation with inputs, outputs and repetitive quiescence. *Software - Concepts and Tools 17,* 3, 103–120. (Cited on pages 52 and 60.)

[111] WEGMAN, M. AND ZADEK, F. 1991. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems 13(2).* (Cited on page 49.)

[112] WEIMER, W., NGUYEN, T. V., GOUES, C. L., AND FORREST, S. 2009. Automatically finding patches using genetic programming. In *ACM/IEEE International Conference on Software Engineering (ICSE)*. 512–521. (Cited on pages 17, 29, 30, 63, 106, and 117.)

[113] WEISER, M. 1982. Programmers use slices when debugging. *Communications of the ACM 25,* 7 (July), 446–452. (Cited on page 22.)

[114] WEISER, M. 1984. Program slicing. *IEEE Transactions on Software Engineering 10,* 4 (July), 352–357. (Cited on page 22.)

[115] WOODS, S. AND Q. YANG, Q. 1998. Program understanding as constraint satisfaction: Representation and reasoning techniques. *Automated Software Engineering 5(2)*, 147–181. (Cited on page 96.)

[116] WOTAWA, F. 2000. Debugging VHDL Designs using Model-Based Reasoning. *Artificial Intelligence in Engineering 14,* 4, 331–351. (Cited on page 29.)

[117] WOTAWA, F. 2001. On the Relationship between Model-based Debugging and Programm Mutation. In *Proceedings of the Twelfth International Workshop on Principles of Diagnosis*. Sansicario, Italy. (Cited on page 29.)

[118] WOTAWA, F. 2002a. Debugging Hardware Designs using a Value-Based Model. *Applied Intelligence 16,* 1, 71–92. (Cited on page 29.)

[119] WOTAWA, F. 2002b. On the Relationship between Model-Based Debugging and Program Slicing. *Artificial Intelligence 135,* 1–2, 124–143. (Cited on pages 23 and 24.)

[120] WOTAWA, F. 2008. Bridging the gap between slicing and model-based diagnosis. In *Proc. of the 20th Intl. Conference on Software Engineering and Knowledge Engineering (SEKE)*. 836–841. (Cited on pages 23 and 24.)

[121] WOTAWA, F. AND NICA, M. 2008a. Converting programs into constraint satisfaction problems. In *Advances in Intelligent and Distributed Computing*, C. Badica and M. Paprzycki, Eds. Studies in Computational Intelligence, vol. 78. Springer Berlin / Heidelberg, 228–236. (Cited on pages 12, 32, and 122.)

[122] WOTAWA, F. AND NICA, M. 2008b. On the compilation of programs into their equivalent constraint representation. *Informatika 32*, 359–371. (Cited on pages 12, 29, 32, 96, and 122.)

[123] WOTAWA, F., NICA, M., AND AICHERNIG, B. K. 2010a. Generating distinguishing tests using the minion constraint solver. In *ICSTW '10: Proceedings of the 2010 Third International Conference on Software Testing, Verification, and Validation Workshops*. IEEE Computer Society, Washington, DC, USA, 325–330. (Cited on pages 13, 82, and 102.)

[124] WOTAWA, F., NICA, M., AND AICHERNIG, B. K. 2010b. Generating distinguishing tests using the minion constraint solver. In *CSTVA 2010: Proceedings of the 2nd Workshop on Constraints for Testing, Verification and Analysis*. IEEE. (Cited on page 113.)

[125] WOTAWA, F., NICA, M., AND IULIA, M. Automated debugging based on a constraint model of the program and a test case. (Cited on pages 13 and 32.)

[126] WOTAWA, F., WEBER, J., NICA, M., AND CEBALLOS, R. 2010. On the complexity of program debugging using constraints for modeling the programs syntax and semantics. In *Current Topics in Artificial Intelligence*, P. Meseguer, L. Mandow, and R. Gasca, Eds. Lecture Notes in Computer Science, vol. 5988. Springer Berlin / Heidelberg, 22–31. (Cited on pages 13 and 122.)

[127] YU-SEUNG MA, J. O. AND KWON., Y. R. 2005. Mujava : An automated class mutation system. *Software Testing, Verification and Reliability 15*, 97–133. (Cited on page 113.)

[128] ZANKER, M., ASCHINGER, M., AND JESSENITSCHNIG, M. 2007. Development of a collaborative and constraint-based web configuration system for personalized bundling of products and services. In *in 8 th International Conference on Web Information Systems Engineering (WISE*. Springer, 273–284. (Cited on page 54.)

[129] ZDC. Zdc constraints solving system. http://www.bracil.net/CSP/cacp/cacpdemo.html. (Cited on page 54.)

[130] ZELLER, A. 2002. Isolating cause-effect chains from computer programs. In *Proc. ACM SIGSOFT 10th International Symposium on the Foundations of Software Engineering (FSE-10)*. ACM, Charleston, South Caroline, USA. (Cited on page 28.)

[131] ZELLER, A. AND HILDEBRANDT, R. 2002. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering 28,* 2 (feb). (Cited on pages 23 and 28.)

[132] ZHANG, X., HE, H., GUPTA, N., AND GUPTA, R. 2005. Experimental evaluation of using dynamic slices for fault localization. In *Sixth International Symposium on Automated & Analysis-Driven Debugging (AADEBUG)*. 33–42. (Cited on pages 23 and 24.)