Master's Thesis

Combining relevant Slicing with Constraint solving for Debugging

Daniel Detassis

Graz, 2011

Institute for Software Technology
Graz University of Technology

Supervisor/First reviewer: Univ.-Prof. Dipl.-Ing. Dr. techn. Franz Wotawa

Abstract (English)

From time to time everyone hears and reads about failures in software which sometime cause a disaster. Despite the large part the testing occupies in the software development process, it is not possible to avoid such failures. There are plenty of tools and techniques out there to help the developer test the program. Once a failure is found, the problem is now that there are not so many tools for helping to locate the failure. The tester of the software and the developer are often different persons, so the tester first has to study the code to be able to locate the failure. This takes a lot of time that will be missed elsewhere. To help the tester or developer to be able to locate the failure faster a prototype of a Debugger is presented in this thesis. The Debugger limits the possibly faulty statements of the program in order to focus the search for a failure to less lines of code.

The Debugger contains four theoretical concepts, namely a debugging problem, relevant slicing, calculating minimal hitting-sets and constraint solving. The relevant slicer stores the execution path and calculates the slices to roughly limit the possibly faulty lines of code. Another algorithm then calculates minimal hitting-sets from the above mentioned slices and finally a consistency check is run with the help of Minion, a constraint solver, to further limit the lines. The result is a remarkable reduction in contrast to the original program and therefore leads to a more efficient locating of failures.

Abstract (German)

Immer wieder hört und liest man von Fehlern in Software, die teils katastrophale Folgen haben. Obwohl das Testen mittlerweile schon einen großen Teil des Entwicklungsprozesses einnimmt, kann man diesem Problem nicht Herr werden. Für das Testen an sich gibt es mittlerweile bereits sehr viele Programme und Methoden, die dem Entwickler helfen. Wofür es allerdings nicht so viele Hilfen gibt, ist für das Auffinden eines solchen Fehlers. Oftmals ist der Tester nicht der Entwickler der Software und muss mühsam den Quellcode studieren um den Fehler zu finden. Dies benötigt natürlich alles Zeit, die andernorts dann fehlt. In dieser Arbeit wird nun ein Prototyp eines Debuggers vorgestellt, der an Hand von fehlschlagenden Testfällen die möglichen fehlerhaften Programmzeilen einschränkt, damit sich der Tester bzw. Entwickler auf weniger Programmzeilen konzentrieren kann.

Der Debugger beinhaltet dabei vier große Konzepte. Aufbauend auf einem Debugging Problem speichert ein relevant Slicer den Ausführungspfad und berechnet daraus Slices um die möglichen fehlerhaften Codezeilen grob einzuschränken. Drittens folgt ein Algorithmus um minimale Hitting-Sets aus den zuvor erwähnten Slices zu berechnen und viertens, einer Konsistenzprüfung. Durch diese Überprüfung, die unter zu Hilfenahme von Minion, einem "Constraint solver", durchgeführt wird, werden die übrigen verbliebenen Programmzeilen weiter eingeschränkt. Das Resultat ist eine deutliche Einschränkung gegenüber dem gesamten Programm und führt dadurch zu einem deutlich effizienteren Auffinden von Fehlern.

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 ix

List of Tables xi

Listings xiii

List of Algorithms xv

1. Introduction 1
1.1. Structure of this document . . . 2

2. Theory 3
2.1. Debugging Problem . . . 3
2.2. Relevant Slicing . . . 5
2.2.1. Execution Trace . . . 5
2.2.2. Control Dependency . . . 5
2.2.3. Data Dependency . . . 7
2.2.4. Potential Data Dependency . . . 9
2.2.5. Execution Trace Graph . . . 11
2.2.6. Relevant Slicing Algorithm . . . 11
2.3. Minimal Hitting-Sets . . . 14
2.4. Constraints . . . 15

3. Debugger 17

4. Implementation 23
4.1. Implementation Details . . . 23
4.1.1. Test Cases in JUnit . . . 23
4.1.2. Execution Trace . . . 25
4.1.3. Relevant Slicer . . . 26
4.1.4. Minimal Hitting-Sets . . . 26
4.1.4.1. Changes . . . 26
4.1.4.2. Implementation Details for Staccato . . . 27
4.1.4.3. Modified Staccato Algorithm . . . 28
4.1.4.4. Usage in the Debugger . . . 29
4.1.5. Constraints . . . 30
4.1.5.1. Converter . . . 30
4.1.5.2. TestCase2Convert . . . 30
4.1.5.3. Consistency Check . . . 32

4.2. Limitations 33
4.3. UML 33
4.3.1. Class Diagram 33
4.3.2. Sequence Diagrams 36

5. Test Results **39**
5.1. Test Platform 39
5.2. Single Test Files 39
5.3. JADE Test Cases 45
5.4. Summary 51

6. Open Problems **53**
6.1. Φ-Function 53
6.2. Relevant Slicer 55

7. Conclusion **57**
7.1. Conclusion 57
7.2. Future Work 57

A. Manual **59**

Bibliography **63**

List of Figures

2.1. The Execution Trace of the Power Function with Test Case testPow 6
2.2. The Execution Trace with its Control Dependencies 8
2.3. The Execution Trace with its Data Dependencies 10
2.4. The Execution Trace with its Potential Data Dependencies 12
2.5. The Execution Trace Graph with all its Dependencies 13
2.6. The Result of a Minion Call 16

3.1. The Work Flow of the Debugger 20
3.2. Run of Test Case pow3 21

4.1. Run of a JUnit Class in Eclipse 24
4.2. A SubSetContainer with SubSets 29
4.3. The Class Diagram 34
4.4. Sequence Diagram for Analyzing 37
4.5. Sequence Diagram for Debugging 38

A.1. The Debugger GUI 59

List of Tables

2.1. Overview of the i-th Statement of Execution 7
2.2. Potential relevant Variables (PR) 9

4.1. Matrix after Initialisation 27
4.2. Matrix after calling Method stripComponent 27
4.3. Matrix after calling Method strip 28
4.4. Conversion of a Java Statement to a Minion Constraint 31

5.1. Test Cases for PowerFunction.java 40
5.2. Test Results for PowerFunction.java 41
5.3. Test Cases for AKSWT1.java 41
5.4. Test Results for AKSWT1.java 41
5.5. Test Cases for ProdSum.java 42
5.6. Test Results for ProdSum.java 42
5.7. Test Cases for Multiplication.java 43
5.8. Test Results for Multiplication.java 43
5.9. Test Cases for Divide.java 44
5.10. Test Results for Divide.java 44
5.11. Test Cases for GCD.java 45
5.12. Test Results for GCD.java 45
5.13. Test Cases for BinSearch.java 46
5.14. Test Results for BinSearch.java 46
5.15. Test Cases for IfExamples.java 47
5.16. Test Results for IfExamples.java 47
5.17. Test Cases for SumPowers.java 48
5.18. Test Results for SumPowers.java 48
5.19. Test Cases for TrafficLight.java 49
5.20. Test Results for TrafficLight.java 49
5.21. Test Cases for WhileLoops.java 50
5.22. Test Results for WhileLoops.java 50
5.23. Test Cases for WhileLoops2.java 51
5.24. Test Results for WhileLoops2.java 51
5.25. Overview of the Testing Results 52

6.1. Conversion of a Java logical Operation to a Minion Constraint 54

Listings

2.1. Implemented Power Function . . . 3
2.2. Test Cases for the Power Function . . . 4
2.3. Swapping Function . . . 7
2.4. Function which makes a Variable positive . . . 9
2.5. Multiplication of two Variables in Minion . . . 15

3.1. CSP in Minion for Test Case pow3 . . . 18

4.1. Test Cases for the Power Function in a JUnit Class . . . 24
4.2. Clipping of the modified Java File . . . 25
4.3. The Call to start the relevant Slicer . . . 26
4.4. The call of Staccato to calculate all minimal Hitting-Sets . . . 29
4.5. Short Minion Program to demonstrate Consistency checking . . . 32

5.1. PowerFunction.java . . . 40
5.2. AKSWT1.java . . . 41
5.3. ProdSum.java . . . 41
5.4. Multiplication.java . . . 42
5.5. Divide.java . . . 43
5.6. GCD.java . . . 44
5.7. BinSearch.java . . . 45
5.8. IfExamples.java . . . 46
5.9. SumPowers.java . . . 47
5.10. TrafficLight.java . . . 48
5.11. WhileLoops.java . . . 49
5.12. WhileLoops2.java . . . 50

6.1. CSP of GCD.java and Test Case testGCD_f3 . . . 53
6.2. Small Program for demonstrating the Φ-Function . . . 55
6.3. SSA Representation of the small Program . . . 55

A.1. Entry in the Input/Output File . . . 60

List of Algorithms

1. The relevant Slicing Algorithm 11
2. The Debugger 17
3. The Staccato Algorithm 28

Chapter 1

Introduction

Nowadays it is not a secret that a lot of money and time are spent on testing and debugging during a software development life cycle. Despite this effort there are still a lot of bugs in modern software. Many of them are not really tragic bugs and just result in a small problem to the user, if at all. But some of these bugs can lead to horrible accidents which may cost the life of humans. Other bugs do not harm human life but cost the users and developers a lot of money. In [4] a list of the latest bigger problems with software bugs is shown.

The question is now how is it possible to find and eliminate these bugs. Of course, extending the testing phase of the project would be a solution but time and money are valuable and it will lead to another question - how far shall the testing phase be extended? The other way is to provide the testers and developers with better tools to find and locate these bugs. If it is possible to test the software more efficiently - this could be doing the same amount of tests with the help of a new testing or debugging tool in half the originally estimated time and therefore be able to test other functionality resp. the same functionality in more detail in the saved half of the time - the software will be less error-prone with the same amount of time and money invested. It should be mentioned here, that it will probably never be possible to guarantee the complete correctness of a program.

A lot of testing tools or whole concepts are already available to help the programmers and testers to avoid or locate bugs. The most well-known tool is JUnit [2], the two most well-known concepts are test-driven development and design by contract. Even testing automation processes exist for running tests again and again without being supervised by a programmer or tester. Using these concepts and tools can help to decrease the amount of bugs made by programmers but these tools and concepts cannot help with one problem programmers face after detecting a bug - What is the cause of the bug?

The normal work flow to locate a bug is to follow the execution trace and find out where the faulty statement in the program lies. This work can be done with the help of built-in debuggers in most of the IDE or by printing out variables with its values or similar techniques. This is a lot of work on current software programs with thousands of lines of code. And it is even harder if the tester is not the programmer of the code.

Wotawa presented in [17] a debugging approach that tries to deal with this problem. As a result the Debugger will present all possibly faulty statements for one particular failing test case. To be able to do this there are four basic parts, namely a failing test case, a relevant slicing algorithm, a minimal hitting-set algorithm and the last part is constraint solving, combined in the Debugger. With the interaction of these parts it is possible to speed up the debugging approach by pointing out statements which may cause the bug.

I will present in this Master thesis the practical implementation of the Debugger published by Wotawa in [17]. The starting point is a failing JUnit test case. JUnit is in use in almost all software development

companies and therefore the test cases are already available and do not need extra effort when being programmed. This means that no mentionable additional effort is necessary for the use of the Debugger. The runtime of the Debugger is, as far as tested, in a very acceptable scope, too.

Since this is the first prototype and mainly a proof of concept there are still a few limitations like the lack of a good implementation of relevant slicing and restrictions with reference to Minion, the constraint solver. Despite of these limitations I will show you that the Debugger will further reduce the possibly faulty statements by approximately 25% in contrast to a sole relevant slicing approach.

1.1. Structure of this document

The thesis is organized as follows: The intention of the next chapter is to provide the reader with all of the theoretical background of the four basic parts of the Debugger. The third chapter presents the Debugger by explaining the interaction of the already mentioned parts. A short example is also given to demonstrate how the Debugger will work in practice. The fourth chapter contains the details about the implementation. Further points of this chapter are UML diagrams, the class diagram and two sequence diagrams as well as the limitations of the Debugger. The next chapter presents the test results and before the thesis ends with a conclusion and future working points the open problems of the current implementation are described.

Chapter 2

Theory

The main idea for this thesis has already been published by Wotawa in [17].
Therefore parts of the contents of this chapter can also be found there.

The debugging approach consists of four basic concepts, namely a debugging problem, relevant slicing, minimal hitting-sets and constraints resp. a constraint satisfaction problem. All the theoretical background for these concepts will be described in this chapter.
For an easier understanding the theoretical aspects are demonstrated after their definition in a practical example which is shown in Listing 2.1.

Listing 2.1: Implemented Power Function

```
1   public static int pow(int a, int b) {
2       int base = a;
3       int exponent = b;
4       int tmp = 0;
5       if(exponent == 0)
6           tmp = 1;
7       else
8           tmp = base;
9
10      int counter = 1;
11      while(exponent > counter) {
12          tmp = tmp + base; //ERROR: tmp = tmp * base;
13          counter = counter + 1;
14      }
15      int result = tmp;
16      return result;
17  }
```

The code fragment implements a power function with an error in line 12. The implementation is trickier than necessary but implemented in this way it will contain more aspects of an execution trace graph.

2.1. Debugging Problem

An important part of a debugging problem are test cases. They are necessary to test a program for correctness. Therefore a single test case needs two environment variables. One variable for the different input

variables with its values and the other one for the output variables with its values. It is worth saying that not every test case has to have input variables but every test case has to have at least one output variable. Now we can define a test case.

Test case A test case T is a pair of two environment variables (I,O). I stands for the input variables with its values used in a program P and O for at least one output variable with its expected value.

An example for a file with more test cases for the program in Listing 2.1 can be seen in Listing 2.2

Listing 2.2: Test Cases for the Power Function

```
1  import junit.framework.TestCase;
2
3  public class PowTest extends TestCase {
4
5      public PowTest(String args) {
6          super(args);
7      }
8
9      public void testPow() {
10         assertEquals(4, Calculator.pow(2, 2));
11     }
12     public void testPow1() {
13         assertEquals(1, Calculator.pow(5, 0));
14     }
15     public void testPow2() {
16         assertEquals(2, Calculator.pow(2, 1));
17     }
18     public void testPow3() {
19         assertEquals(27, Calculator.pow(3, 3));
20     }
21     public void testPow4() {
22         assertEquals(64, Calculator.pow(2, 6));
23     }
24 }
```

These test cases are implemented in a JUnit Test class. Further information to JUnit will follow in Chapter 4.1.1. For now, the only important information is that the test file contains passing and failing test cases. A passing test case is a test case where the environment variable O of the test run is equal to the environment variable O of the test case. In other words, the result of the test run is as expected. The test cases testPow, testPow1 and testPow2 are such passing test cases. Taking for example test case testPow - if you call the power function in Listing 2.1 with the input variables a = 2 and b = 2, the output variable will be 4.

We speak of a failing test case when the environment variables O differ from each other. Such a failing test case signals an error - this means that the expected output is different from the output the program has calculated. This applies to the other two test cases in Listing 2.2. The result of the call of the power function with arguments a = 3 and b = 3 will not be 27 as expected but 9. This is because of the error in line 12 of the power function.
With the explanation of passing and failing test cases we are now able to define a debugging problem as follows:

Debugging problem A debugging problem is a tuple (P,T) consisting of a test case T and the corresponding program P and exists if the environment variable O of T is different to O of P. T is therefore a failing test case.

Such a debugging problem is the starting point of the Debugger because if there are not any debugging problems, the use of a Debugger will be senseless.

2.2. Relevant Slicing

Slicing in general was first introduced by Mark Weiser in the early 80ies [14] and [15]. Weiser mentioned that a programmer naturally does slicing when trying to locate an error by following the data and control flow of the programming. This is some kind of slicing because statements irrelevant for the outcome are skipped by the programmer. This method was named program slicing, later on known as static slicing. A disadvantage was that the input of a program and the resulting execution trace were not taken into account, which has led to very large slices. An improvement was made by Korel and Laski by publishing dynamic slicing in the late 80ies [9]. Now the input and the execution graph takes on an important role for slicing a program. Dynamic slicing works only with two of the three later on presented dependencies, namely the control dependency and the data dependency. The problem was that under certain circumstances, see Listing 2.4 for such an example, the dynamic slicing approach does not include the faulty statement. Because of this fact relevant slicing was used in [8] and [18]. For relevant slicing a third dependency, the potential data dependency is needed. The disadvantage of it is that knowledge of the source code is needed to calculate the potential relevant variables and further on the potential data dependencies.

Relevant slicing is used as slicing method in the debugging approach presented in this thesis. The most important argument for this decision was the possibility of skipping the faulty statement when using dynamic slicing.

To be able to calculate relevant slices an execution trace graph is necessary. Before the algorithm for relevant slicing can be presented, the important aspects of an execution trace graph are described.

2.2.1. Execution Trace

To be able to locate the cause of the debugging problem the knowledge about the executed statements of the program is essential. It is obvious that the cause of a certain debugging problem has to lie, at least, in one executed statement. Later on we will see that we are able to skip more statements with the help of relevant slicing but for now we will define the execution trace as follows:

Execution trace An execution trace (P,T) is a sequence $< s(1)^1, ..., s(i)^k >$ where s represents the line number and k represents the execution order of the statements while running a test case T on program P.

The fact of an execution trace being a sequence and not simply a set is very important because the order of execution is crucial.

Running the fourth test case of the test file presented in Listing 2.2 on the program presented in Listing 2.1 will create the execution trace in Figure 2.1. These are just all executed statements of the program.

Just the execution trace alone is not that helpful with large programs. Therefore it is necessary to add information about the interaction and relationship between the statements resp. nodes to be able to reduce the amount of possibly faulty statements. There are three important dependencies, which will be described in the following sections.

2.2.2. Control Dependency

The definition of a control dependency is quite simple.

Control dependency A statement $s(i)^i$ of an execution trace $< s(1)^1, ..., s(i)^n >$ is control dependent on another statement $s(j)^j$ if and only if the execution of $s(j)^j$ causes the execution of $s(i)^i$. To express this dependency we write $s(j)^j \longrightarrow_{CD} s(i)^i$.

With a view to the implementation of the Debugger there are two important statements, namely the if- and while-statements.

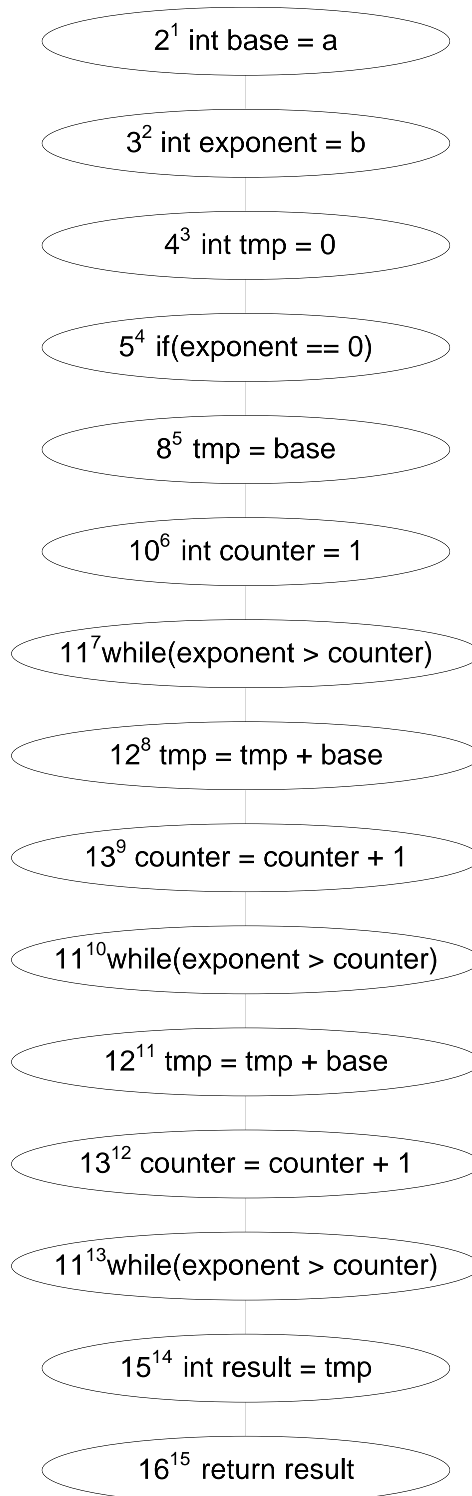

Figure 2.1.: The Execution Trace of the Power Function with Test Case testPow

- **while**
  Looking at a while-loop, this means that if the condition evaluates to TRUE, the body of the while-loop is control dependent on the condition. On the other side, if it evaluates to FALSE, nothing is control dependent.
- **if-then-else**
  If a conditional statement evaluates to TRUE, the then-block is control dependent, otherwise, if the statement evaluates to FALSE, the else-block is control dependent.

It is to note, that the Debugger just supports conditional statements and while-loops at present. A possibility for other constructs, as far as possible, is to change them into one of those two.

An exceptional case are nested if- and while-statements. In this case not all statements of the subblocks are automatically control dependent on the hierarchically highest conditional, but on the last one. Table 2.1 gives an overview of the i-th statement of execution.

<table>
<tr><td>while C do
w_1 ... w_n
end do</td><td>C =
TRUE</td><td>$C^i \longrightarrow_{CD} w_1^{i+1}$
...
$C^i \longrightarrow_{CD} w_n^{i+n}$</td></tr>
<tr><td rowspan="2">if C then
t_1 ... t_n
else
e_1 ... e_n
end if</td><td>C
=
TRUE</td><td>$C^i \longrightarrow_{CD} t_1^{i+1}$
...
$C^i \longrightarrow_{CD} t_n^{i+n}$</td></tr>
<tr><td>C
=
FALSE</td><td>$C^i \longrightarrow_{CD} e_1^{i+1}$
...
$C^i \longrightarrow_{CD} e_n^{i+n}$</td></tr>
</table>

Table 2.1.: Overview of the i-th Statement of Execution

Figure 2.2 shows the execution trace of the power function with all its control dependencies.

2.2.3. Data Dependency

Data dependencies are important to trace variables which are relevant for the outcome of a specific test case.
Have a look at the swapping method in Listing 2.3.

Listing 2.3: Swapping Function

```
public void swap(int a, int b) {
    int irrelevant = 100;
    int tmp = a;
    int a = b;
    int b = a; //ERROR: b = tmp
}
```

Assume that variable b has the wrong value at the end of the method. There is no reason why, for example, line 2 should have to do something with the wrong calculation because the variable irrelevant is never used again in the method. Therefore it does not matter if the variable is assigned to 100, 1000 or anything else. To skip these unnecessary lines, data dependencies are introduced.
To define data dependencies, so-called DEF and REF sets are crucial. Every statement, or later in the execution trace graph every node, has its own.

DEF set A DEF set of a statement or node contains all defined variables of this statement or node.

Taking line 3 of Listing 2.3 the DEF set is [tmp].

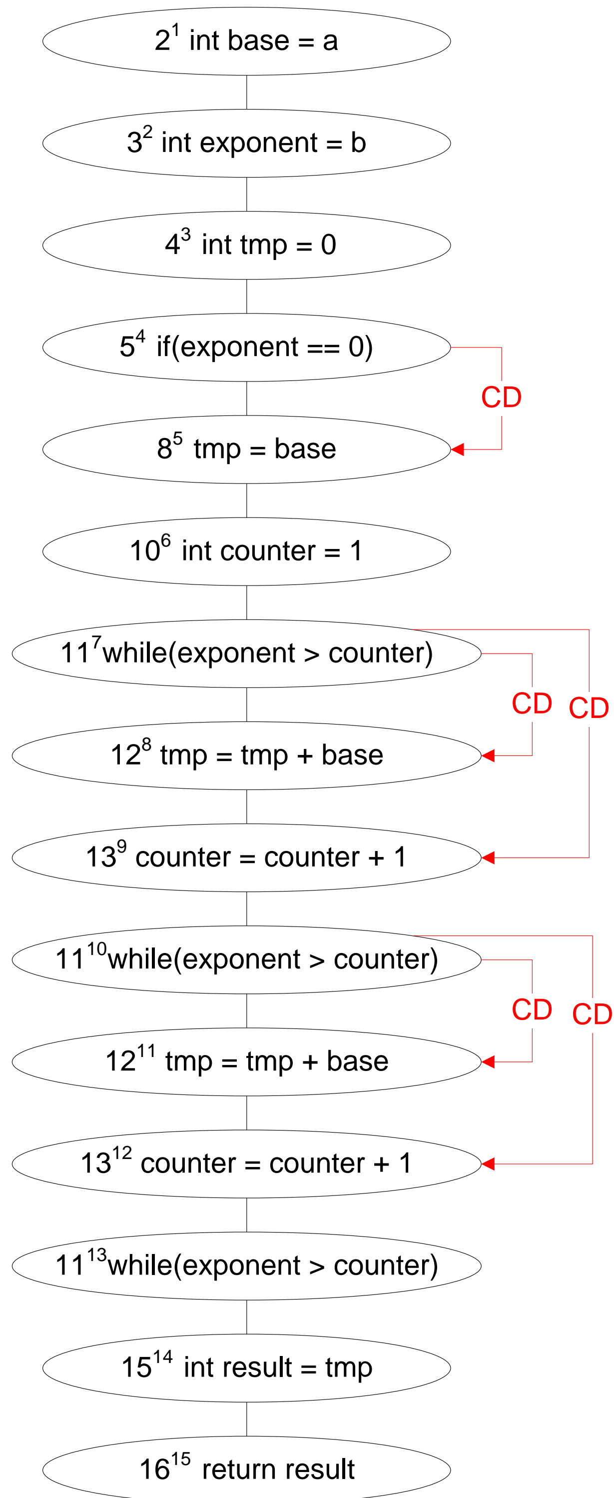

Figure 2.2.: The Execution Trace with its Control Dependencies

REF set A REF set of a statement or node contains all referenced variables of this statement or node.

In contrast to DEF sets conditional statements or while-loops do have REF sets.

- **Line 3 in Listing 2.3:** The REF set is [a].
- **Line 11 in Listing 2.1:** The REF set is [exponent, counter].

Knowing DEF and REF sets the definition of a data dependency is as follows.

Data dependency A statement $s(i)^i$ of an execution trace $< s(1)^1, ..., s(i)^n >$ is data dependent on another statement $s(j)^j$ if and only if a variable x $\in$ DEF of $s(j)^j$ and the same variable x $\in$ REF of $s(i)^i$ and there is no $s(k)^k$ with $i < k < j$ where x is redefined (x $\in$ DEF of $s(k)^k$). To express this dependency we write $s(j)^j \longrightarrow_{DD} s(i)^i$.

Figure 2.3 shows again the execution trace of the power function including its data dependencies.

2.2.4. Potential Data Dependency

The normal data dependency as described in the previous chapter, works with DEF and REF sets to trace variables which may cause the faulty behaviour. But it can happen that an important variable will not be taken into account when later on an algorithm called relevant slicing is run to limit the amount of potential faulty statements. In this case we need the third and last dependency, the potential data dependency. Before it is defined, we will take a look at the example in Listing 2.4.

Listing 2.4: Function which makes a Variable positive

```
1   public void makeVariablePositive(int a) {
2       if (a > 0) {     //ERROR: if(a < 0)
3           int a = a * -1;
4       }
5       result = a;
6       return result;
7   }
```

With input a = -3 the result should be a = 3 but it is -3, because of the fact that the condition in line 2 is evaluated to FALSE and therefore the right value in line 3 cannot be calculated and assigned to variable a. But if the condition had been evaluated to TRUE the variable a would have been redefined to the right value. In order not to ignore such situations every conditional statement or while-loop has its potential relevant variables.
Table 2.2 shows the calculation of them.

Statement s	Condition C	Potential relevant variables PR
while C do	TRUE	PR(s,TRUE = {})
W		
end do	FALSE	PR(s, FALSE = {Union of all defined variables in W})
if C then		
T	TRUE	PR(s, TRUE = {Union of all defined variables in E})
else		
E	FALSE	PR(s, FALSE = {Union of all defined variables in T})
end if		

Table 2.2.: Potential relevant Variables (PR)

This means for our example in Listing 2.4 that the potential relevant variables for line 2 are

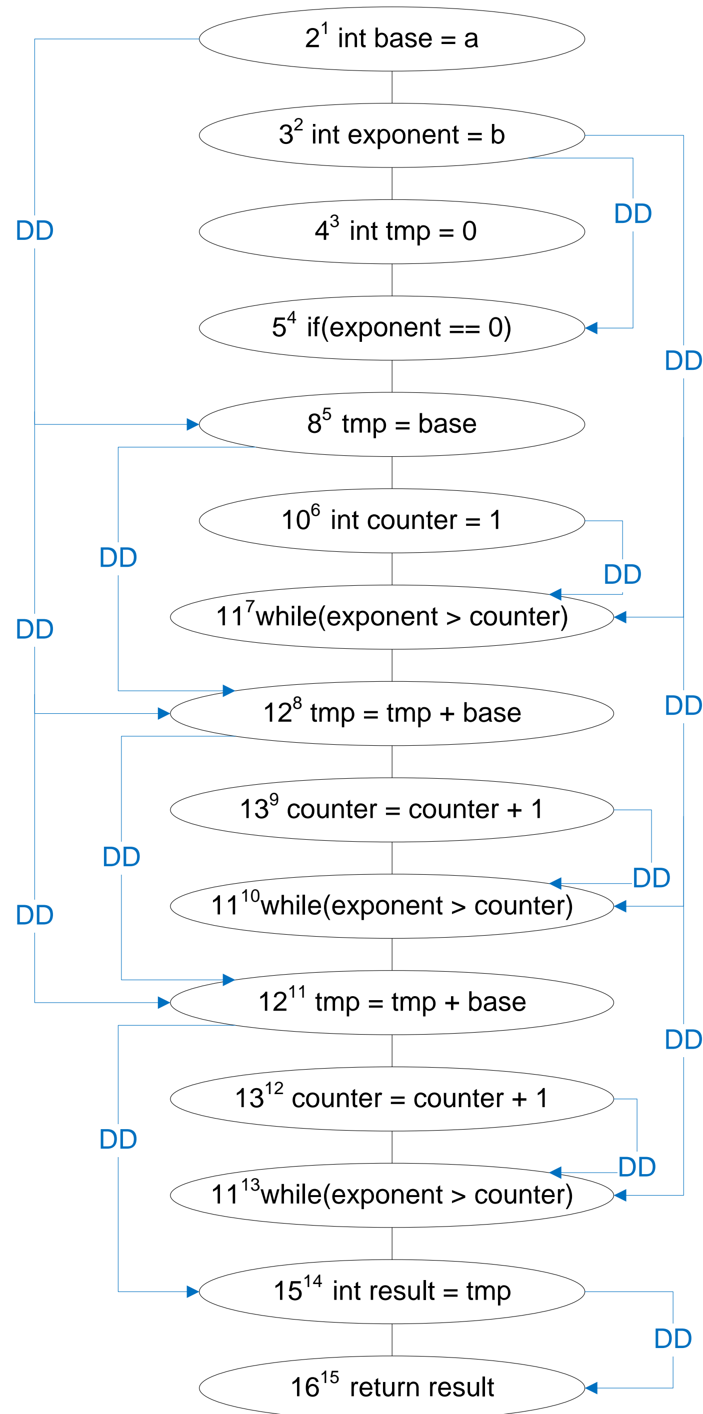

Figure 2.3.: The Execution Trace with its Data Dependencies

PR(2. if(a > 0), FALSE) = a.
PR(2. if(a > 0), TRUE) = {}.

This can be interpreted as: If the condition in line 2 evaluates to TRUE, then variable a will be redefined and in this case the assignment to variable result in line 5 may be different. So the fact what might happen if the condition changed is quite important. With the explanation of potential relevant variables we are now able to define a potential data dependency.

Potential data dependency A statement $s(i)^i$ of an execution trace $< s(1)^1, ..., s(i)^n >$ is potentially data dependent on another statement $s(j)^j$ if and only if the condition of statement $s(j)^j$ evaluates to TRUE resp. FALSE and it exists a variable x $\in$ PR($s(j)^j$, TRUE) resp. PR($s(j)^j$, FALSE) and the same variable x $\in$ REF of $s(i)^i$ and there is no $s(k)^k$ with $i < k < j$ where x is redefined (x $\in$ DEF of $s(k)^k$). To express this dependency we write $s(j)^j \longrightarrow_{PDD} s(i)^i$.

This was the last dependency. In Figure 2.4 the execution trace of the power function with the potential data dependencies is shown.

2.2.5. Execution Trace Graph

With the help of the execution trace and the three different dependencies it is possible to draw an execution trace graph ETG, which will be necessary for the algorithm in the next chapter.

Execution trace graph ETG An ETG(N,E) is a directed graph where every node N represents a single statement and every directed edge E represents one of the three possible dependencies.

The execution trace graph for the program in Listing 2.1 and the fourth test case in Listing 2.2 is shown in Figure 2.5. It is the execution trace with control, data and potential data dependencies simultaneously.

2.2.6. Relevant Slicing Algorithm

To be able to present the algorithm for relevant slicing the definition of a slicing criterion has to be introduced first.

Slicing criterion A slicing criterion (T,v^k) consists of a test case T and the variable of interest v with index k. The index k is thereby not the line number but the k-th statement of execution.

Algorithm 1 shows the relevant slicing algorithm.

Algorithm 1 The relevant Slicing Algorithm

Input: The execution trace $< s(1)^1, ..., s(i)^k >$ of a program P and its test case T and a slicing criterion (T,v^k).
Output: The relevant slice of a program P.

1. Calculate the ETG of the execution trace with the help of the data, control and potential data dependencies.
2. Mark the last node of ETG where v $\in$ DEF(s(i)^j).
3. Mark all conditional statements or while-loops between s(i)^j and s(i)^k which evaluate to a boolean value B and v $\in$ PR(s(i), B).
4. Traverse the ETG backwards from the marked nodes and add every reachable node to a set S.
5. If no new nodes can be added return the set S. Note that the set S contains every statement just once, multiple iterations of the same statement will be unionized.

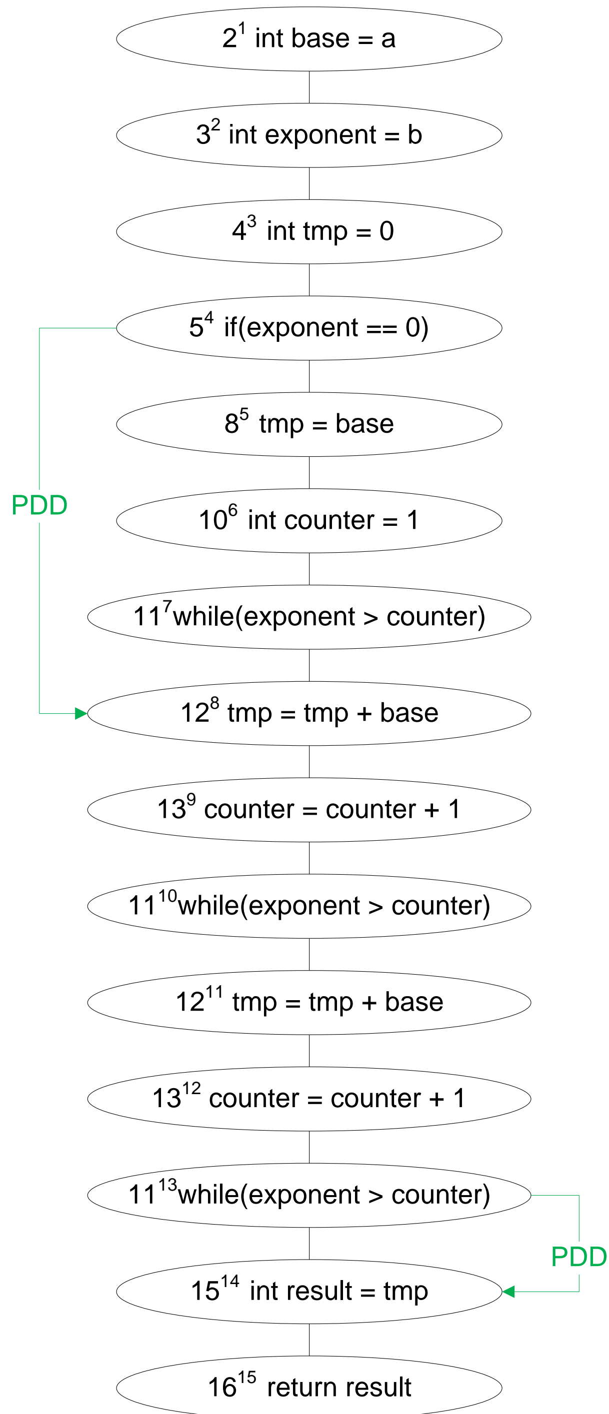

Figure 2.4.: The Execution Trace with its Potential Data Dependencies

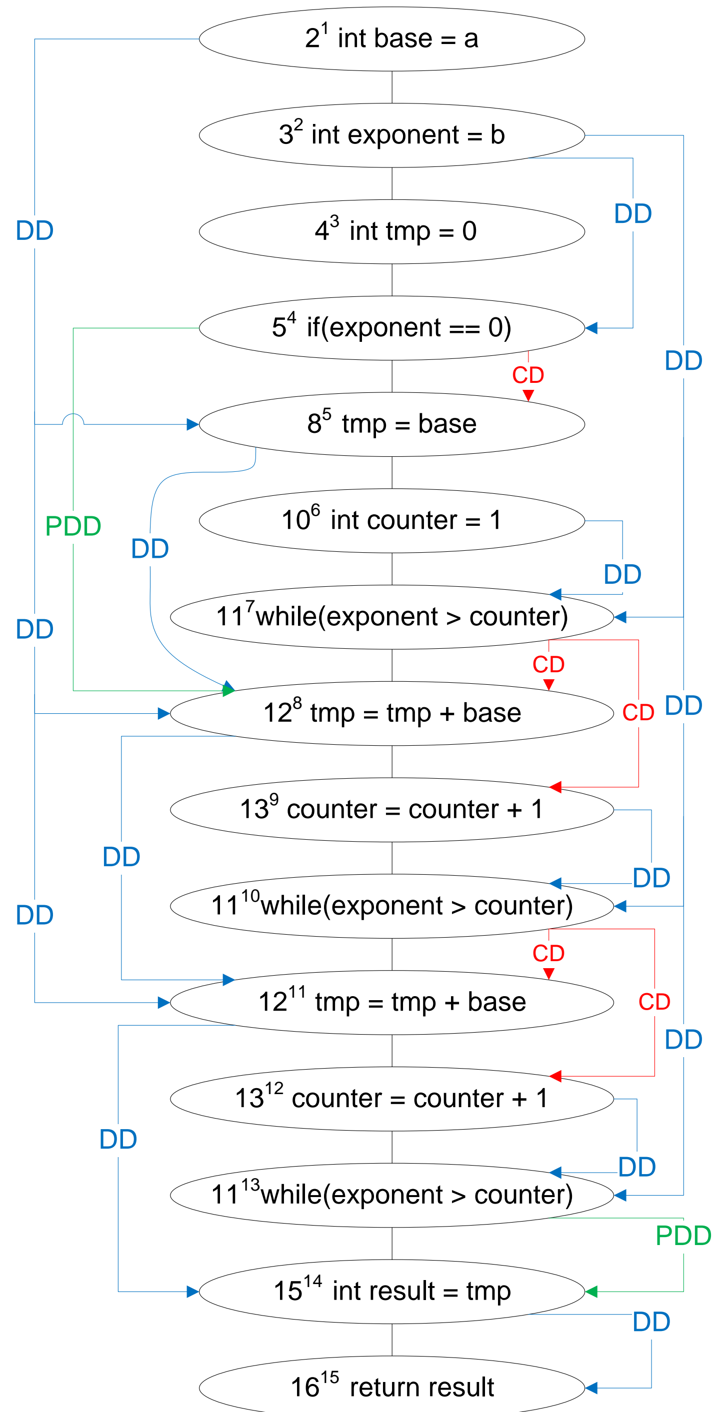

Figure 2.5.: The Execution Trace Graph with all its Dependencies

An important point to notice is that step 3 of the relevant slicing algorithm is not executed in the running example but it will be necessary if a conditional statement or while-loop may redefine the variable of the slicing criterion after its last definition in the ETG. Remember the example in Listing 2.4. The variable of interest is now variable a. Without step 3 of the algorithm the slice would not contain any line because variable a is not defined in any node of the ETG. With step 3 the slice contains line 2 because in the conditional statement the variable a is (re-)defined and therefore a potential relevant variable and the statement has to be marked.

Applying this algorithm to the ETG in Figure 2.5 with the slicing criterion ((I = {a = 3, b = 3}, O = {result = 27}), result[15]) it results in the following set S.

- 02: int base = a
- 03: int exponent = b
- 05: if(exponent == 0)
- 08: tmp = base
- 10: int counter = 1
- 11: while(exponent > counter)
- 12: tmp = tmp + base
- 13: counter = counter + 1
- 15: int result = tmp

2.3. Minimal Hitting-Sets

The next important theoretical concept for the Debugger are minimal hitting-sets. Minimal hitting-sets or in general hitting-sets were first introduced by Reiter in [11].

Hitting-set Suppose C is a collection of sets. A hitting-set for C is a set $H \subseteq \bigcup_{S \in C} S$ such that $H \cap S \neq \{\}$ for each $S \in C$. [11]

From this definition the definition for a minimal hitting-set is according to [11]

Minimal hitting-set A minimal hitting-set is a hitting-set for C which does not contain any subset, which is also a hitting-set for C.

For example, taking two sets $S_1 = \{1, 2, 3, 4\}$ and $S_2 = \{2, 5\}$ the minimal hitting-sets will be {2}, {1, 5}, {3, 5} and {4, 5}.

Now the question is how this helps the Debugger to calculate fault candidates. Assume that we have a relevant slice for every variable of an execution trace that does not match the expected value. The work flow of a programmer is in most cases to take a statement of each slice and assume the faultiness of this statement. What the programmer does here is unconsciously calculate the hitting-sets of the relevant slices. To spare him of this exercise the calculation of minimal hitting-sets from different relevant slices is needed.

Taking our running example from Listing 2.1. The relevant slice for the variable result is

- {2, 3, 5, 8, 10, 11, 12, 13, 15}.

Now slicing the ETG in Figure 2.5 for variable ”tmp” the slice will be

- {2, 3, 5, 8, 10, 11, 12, 13}.

Running an algorithm to calculate minimal hitting-sets will return

- $MHS_1 = \{2\}$
- $MHS_2 = \{3\}$

- MHS_3 = {5}
- MHS_4 = {8}
- MHS_5 = {10}
- MHS_6 = {11}
- MHS_7 = {12}
- MHS_8 = {13}

All of these lines contained in the different minimal hitting-sets are possibly faulty statements and a first set of diagnoses, which will be further limited later on. In these examples this step was not that successful because it is a small example with similar slicing results for both variables. This step helps most in case of a larger program with more false variables and different slices.

There are lots of algorithms which calculate minimal hitting-sets like the Staccato algorithm [13], the Boolean algorithm[10], the Binary Hitting Set Tree algorithm (BHS)[10] or the HST [16]. The main difference is the data structure used to calculate the sets. They differ from directed acyclic graphs over tree structures to boolean arrays. The debugger will use the Staccato algorithm. More information about the algorithm and its implementation details can be found in Chapter 4.1.4.

2.4. Constraints

The last theoretical aspect is constraints and further on solving a constraint satisfaction problem.

Constraints Constraints CO are a set of logical or arithmetic relations which link together different variables V with values from a domain D.

With the help of constraints it is possible to create a constraint satisfaction problem which can be either solvable or not solvable. A problem is solvable if a valid assignment exists for each variable of the constraint satisfaction problem. The definition is according to [17]

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

To solve a CSP a so-called constraint solver is necessary. To get an overview of different constraint solvers and their algorithms have a look at [5].

The Debugger uses Minion [3]. The reasons for this decision were the currentness and the continuous development of Minion. Details of the internal functions and benchmark tests can be found in [7].

Listing 2.5 shows a short Minion program that multiplies two variables. In Minion all of the used variables with their corresponding domain - in this case the variables a, b and result can assign to values between 0 and 92 - in a program have to be defined first. See line 3, 4 and 5 for examples. Afterwards the constraints section follows as you can see in lines 7 - 10. The "product" constraint represents a simple multiplication (a * b = result) and the "eq" constraint represents an assignment (a = 3, b = 5). What Minion now does is to find a solution for this CSP according to the variables and constraints. Variables a and b are already assigned to 3 and 5 by the constraints in line 9 and 10, so the last unknown variable is result.

Listing 2.5: Multiplication of two Variables in Minion

```
1   MINION 3
2   **VARIABLES**
3       DISCRETE a {0..92}
4       DISCRETE b {0..92}
5       DISCRETE result {0..92}
```

```
6      **CONSTRAINTS**
7          product(a, b, result)
8      # TESTCASE
9          eq(a,3)
10         eq(b,5)
11     **EOF**
```

Figure 2.6 shows the result of Minion. The three lines in the middle of the figure which start with "Sol:" list the solution of the CSP. With a = 3, b = 5 and result = 15 - the variables are listed in order of their definition in the Minion file - the problem is solvable.

```
C:\WINDOWS\system32\cmd.exe
C:\Dokumente und Einstellungen\Daniel\Desktop\minion-0.12\bin>minion.exe ../../m
ult.minion
# Minion Version 0.12
# Git version: 65512633daee570de1fdf16a0025d919f6f3753e
# Git last changed date: Mon Feb 8 17:33:56 2010 +0000
#  Run at: UTC Mon Sep 19 09:28:26 2011

#    http://minion.sourceforge.net
#  Minion is still very new and in active development.
#  If you have problems with Minion or find any bugs, please tell us!
#  Mailing list at: https://mail.cs.st-andrews.ac.uk/mailman/listinfo/mug
# Input filename: ../../mult.minion
# Command line: minion.exe ../../mult.minion
Parsing Time: 0.000000
Setup Time: 0.031250
First Node Time: 0.000000
Initial Propagate: 0.000000
First node time: 0.000000
Sol: 3
Sol: 5
Sol: 15

Solution Number: 1
Time:0.000000
Nodes: 1

Solve Time: 0.000000
Total Time: 0.031250
Total System Time: 0.000000
Total Wall Time: 0.078000
Maximum Memory (kB): 0
Total Nodes: 1
Problem solvable?: yes
Solutions Found: 1
```

Figure 2.6.: The Result of a Minion Call

The assignment of values to the different variables is not that important for the Debugger. The important fact is whether a CSP is solvable or not, which can be found in the last but one line of the output.

Chapter 3

Debugger

After we have come to know all the theoretical concepts of the Debugger this chapter describes the co-operation among them. To be able to debug programs and return possibly faulty statements every element has to fulfil its function. Figure 3.1 represents the work flow of the Debugger in a rough way and Algorithm 2 shows the pseudo code. At the end of this chapter an example is demonstrated.

Algorithm 2 The Debugger

Input: A program P and a test case T(I,O).
Output: A set of valid diagnoses.

1. Create the ETG of the failing test case T with the help of the execution trace ET, $\longrightarrow_{CD}$, $\longrightarrow_{DD}$ and $\longrightarrow_{PDD}$.
2. Call Converter(ET, T) to convert the execution trace and test case to a constraint satisfaction problem CSP.
3. Let SS be the empty set of slices.
4. While there exists a variable v $\in$ O that differs from the expected value call RelevantSlicer(ETG, v^k) and add slice to SS.
5. Let MHS be the empty set of minimal hitting-sets.
6. Call HittingSet(SS) and add all calculated minimal hitting-sets to MHS. Each minimal hitting-set represents one possible solution for the failing test case.
7. Let VD be the empty set of valid diagnoses.
8. While MHS $\neq \{\}$
   a) Take $Set_1 ... Set_n \in$ MHS and combine it with CSP.
   b) Run Minion.
   c) If CSP combined with Set_i is solvable add Set_i to VD, otherwise discard Set_i.
9. Return VD.

Let us simulate the algorithm with the already known power function from Listing 2.1 and test case "testPow3" from Listing 2.2.

1. The test case ”testPow” with input variables a = 3 and b = 3 and the expected outcome of 27 is run. But the outcome is 9 and this implies the faultiness of the power function and the starting point of the Debugger.
2. The execution trace, which can be seen in Figure 2.1, of the test run is extracted.
3. The execution trace together with the entered input and entered output variables of the test case is converted to Minion constraints. Note that every variable and its values has to be entered by the user or read in with the help of a file, but this is explained in more detail in Appendix A. For now we assume the variables a = 3, b = 3 and result = 27 as entered.
4. Together with the data dependencies, control dependencies and potential data dependencies the execution trace graph is built, Figure 2.5, and the slices for the output variables are calculated.
   The slice for the variable result is:
   - result: {2, 3, 5, 8, 10, 11, 12, 13, 15}
5. The calculation of minimal hitting-sets leads to the following diagnoses:
   a) MHS_1 = {2}
   b) MHS_2 = {3}
   c) MHS_3 = {5}
   d) MHS_4 = {8}
   e) MHS_5 = {10}
   f) MHS_6 = {11}
   g) MHS_7 = {12}
   h) MHS_8 = {13}
   i) MHS_9 = {15}
6. Nine different CSPs are created with the help of the converted constraints and variables together with the nine diagnoses. Look at Listing 3.1 for one concrete CSP. The other eight just differ in the ”ab”-variable which is always set according to the minimal hitting-set. If the CSP is solvable, the diagnose remains valid and the contained line is resp. the contained lines are possibly faulty statements. If minion does not find any solution, the diagnose is not valid any more and can be discarded.
7. The valid diagnoses are line 2, 5, 8, 11, 12 and 15 as it can be seen in Figure 3.2. The programmer can now concentrate on these six lines instead of the nine above. This is a reduction by a third.
   To run the method with the Debugger a Java class was necessary - therefore the line numbers are increased by 2 in the figure.

Listing 3.1: CSP in Minion for Test Case pow3

```
1  MINION 3
2  **VARIABLES**
3    BOOL ab[8]
4    DISCRETE b_0 {-10..128}
5    DISCRETE a_0 {-10..128}
6    DISCRETE result_0 {-10..128}
7    DISCRETE exponent_0 {-10..128}
8    DISCRETE counter_0 {-10..128}
9    DISCRETE counter_1 {-10..128}
10   DISCRETE counter_2 {-10..128}
11   DISCRETE base_0 {-10..128}
12   DISCRETE tmp_0 {-10..128}
13   DISCRETE tmp_1 {-10..128}
```

```
14    DISCRETE tmp_2 {-10..128}
15    DISCRETE tmp_3 {-10..128}
16    DISCRETE created_tmp_var_1 {-10..128}
17    DISCRETE created_tmp_var_2 {-10..128}
18    DISCRETE created_tmp_var_3 {-10..128}
19    DISCRETE created_tmp_var_4 {-10..128}
20  **CONSTRAINTS**
21    watched-or({element(ab,0,1),eq(base_0,a_0)})
22    watched-or({element(ab,1,1),eq(exponent_0,b_0)})
23    watched-or({element(ab,2,1),eq(tmp_0,0)})
24    watched-or({element(ab,3,1),eq(tmp_1,base_0)})
25    watched-or({element(ab,4,1),eq(counter_0,1)})
26    watched-or({element(ab,5,1),weightedsumgeq([1,1],[tmp_1,base_0],
        created_tmp_var_1)})
27    watched-or({element(ab,5,1),weightedsumleq([1,1],[tmp_1,base_0],
        created_tmp_var_1)})
28    watched-or({element(ab,5,1),eq(tmp_2,created_tmp_var_1)})
29    watched-or({element(ab,6,1),weightedsumgeq([1,1],[counter_0,1],
        created_tmp_var_2)})
30    watched-or({element(ab,6,1),weightedsumleq([1,1],[counter_0,1],
        created_tmp_var_2)})
31    watched-or({element(ab,6,1),eq(counter_1,created_tmp_var_2)})
32    watched-or({element(ab,5,1),weightedsumgeq([1,1],[tmp_2,base_0],
        created_tmp_var_3)})
33    watched-or({element(ab,5,1),weightedsumleq([1,1],[tmp_2,base_0],
        created_tmp_var_3)})
34    watched-or({element(ab,5,1),eq(tmp_3,created_tmp_var_3)})
35    watched-or({element(ab,6,1),weightedsumgeq([1,1],[counter_1,1],
        created_tmp_var_4)})
36    watched-or({element(ab,6,1),weightedsumleq([1,1],[counter_1,1],
        created_tmp_var_4)})
37    watched-or({element(ab,6,1),eq(counter_2,created_tmp_var_4)})
38    watched-or({element(ab,7,1),eq(result_0,tmp_3)})
39  # TESTCASE
40    eq(ab[0],1)
41    eq(ab[1],0)
42    eq(ab[2],0)
43    eq(ab[3],0)
44    eq(ab[4],0)
45    eq(ab[5],0)
46    eq(ab[6],0)
47    eq(ab[7],0)
48    eq(b_0,3)
49    eq(a_0,3)
50    eq(result_0,27)
51  **EOF**
```

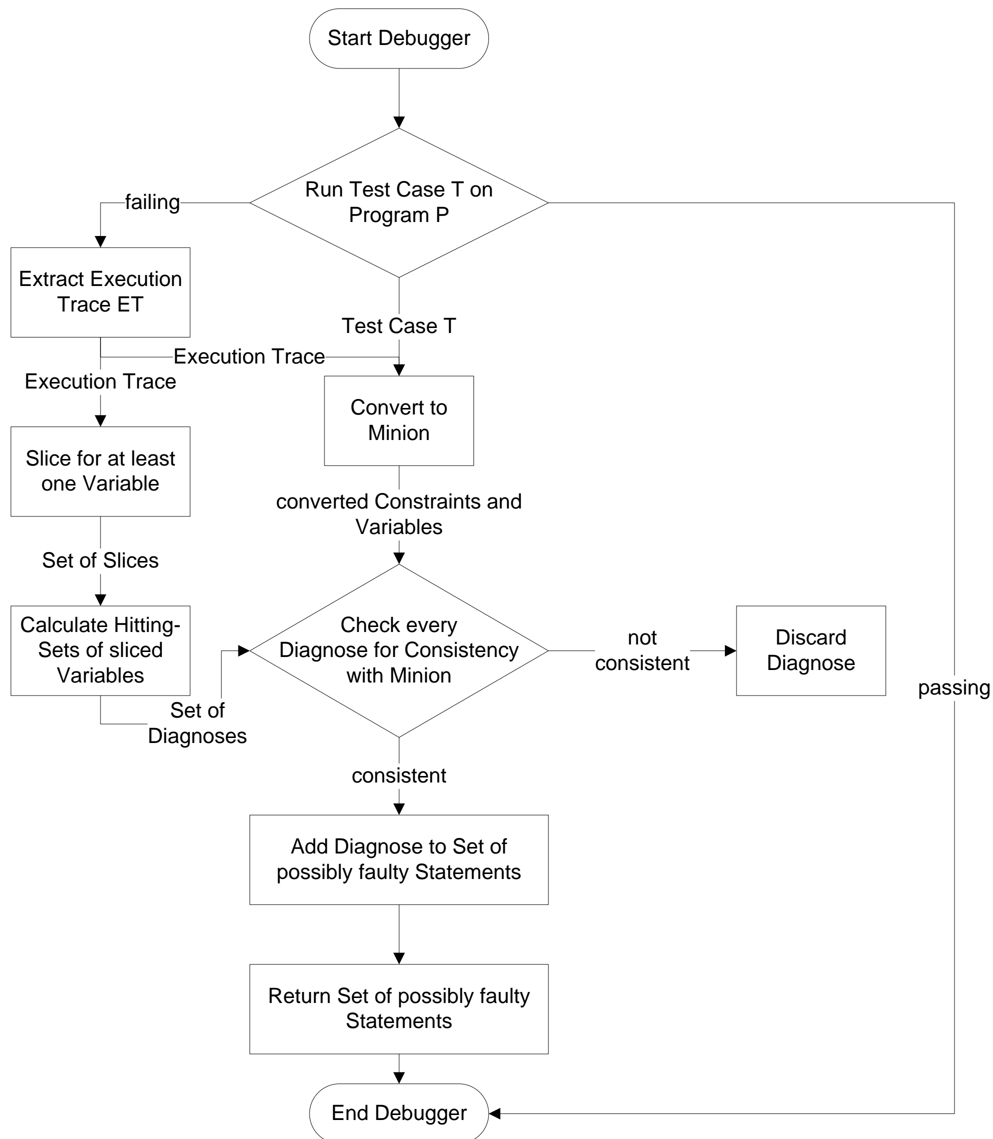

Figure 3.1.: At the beginning the desired test case is run by the Debugger and the execution trace is stored. If the test case passes, the work has been done because there is not any error in the program. The stored execution trace can be ignored. The interesting point is in the case of failing. Then the execution trace is extracted and together with the test case converted to Minion. This is necessary to be able to check the diagnoses for consistency later on. The second step is to calculate these diagnoses. To achieve this the relevant slicer and a minimal hitting-set algorithm are necessary. The relevant slicer slices the extracted execution trace for every different output variable. Afterwards the slices are used to calculate minimal hitting-sets. These calculated sets are equal to the set of diagnoses. The last step is to check whether every diagnose is consistent or not. Let us assume that it is possible to solve the CSP created from execution trace and input and output of the test case with assumptions about correctness and incorrectness of the set of diagnoses. If this is possible, the diagnose will be valid, otherwise it can be ignored.

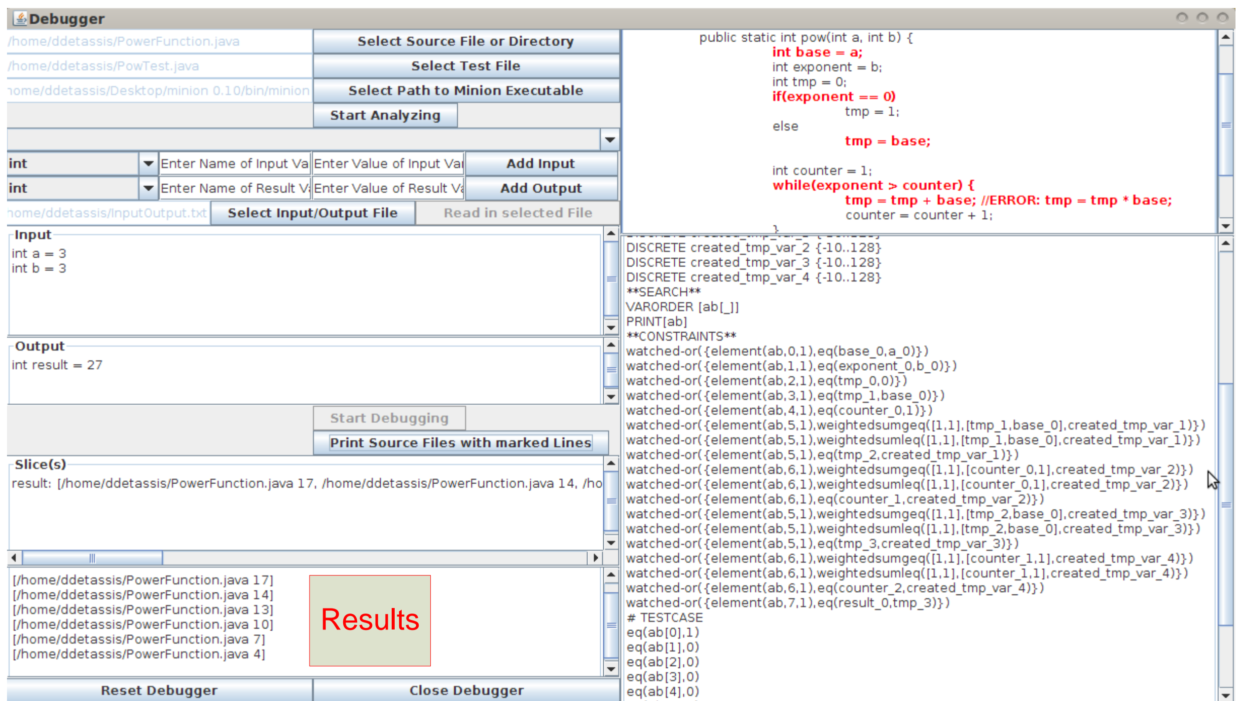

Figure 3.2.: Run of Test Case pow3

Chapter 4

Implementation

This chapter explains the relevant aspects of the implementation and the design of the Debugger. In the first section the implementation details of the theoretical concepts presented in Chapter 2 are described in a precise way. The second section contains a list of all limitations respectively not yet implemented features. The concluding section consists of UML diagrams to explain the correlation between the classes and the application flow in general.

4.1. Implementation Details

4.1.1. Test Cases in JUnit

The starting point for the Debugger is a failing test case. To avoid additional work for the programmer by creating extra test files or test cases our decision was to simply rerun JUnit test cases. Nowadays, JUnit is already in use in most software projects and therefore these test cases already exist. There are also plug-ins to use it out-of-the-box with Eclipse or similar IDEs. More information about JUnit can be found on the official homepage [2].

A run of a test class can be seen in Figure 4.1. Two of the five test cases in the test class fail because of the error in line 10. The failure trace shows the expected output value and the actual one.

It is not necessary to have an IDE and a plug-in, it is enough to have just a JUnit test class. For those who do not know JUnit test classes, Listing 4.1 shows a JUnit class which is ready to be used by the Debugger.

There are a few points to note

- **a)**
  The TestCase class of the JUnit package has to be imported.
- **b)**
  The test class has to be inherited from TestCase.
- **c)**
  Each test class has to have a constructor which takes a String as argument and calls the constructor of the base class.
- **d)**
  First of all each test case has to start with ”test” to be identified as test case by the Debugger. The second point is that each test case shall only contain one assert statement because JUnit does not

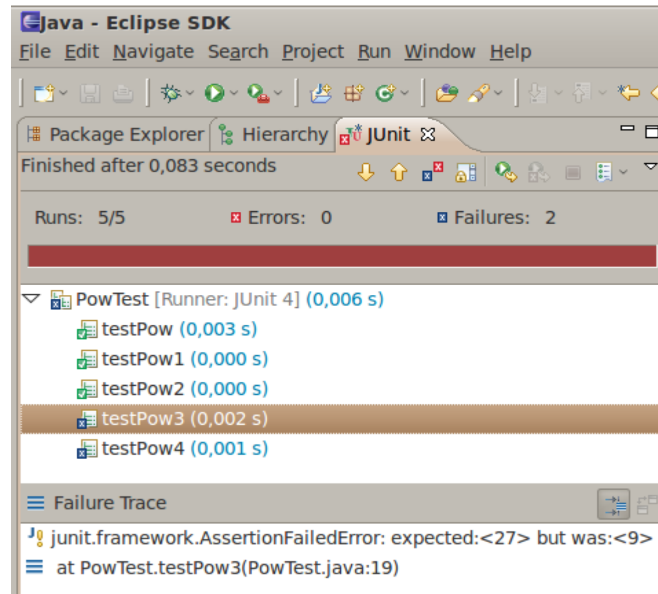

Figure 4.1.: Run of a JUnit Class in Eclipse

Listing 4.1: Test Cases for the Power Function in a JUnit Class

```
1  import junit.framework.TestCase;                     a)
2
3  public class PowTest extends TestCase {              b)
4
5      public PowTest(String args) {                    c)
6          super(args);
7      }
8
9      public void testPow() {                          d)
10         assertEquals(4, Calculator.pow(2, 2));
11     }
12     public void testPow1() {
13         assertEquals(1, Calculator.pow(2, 0));
14     }
15     public void testPow2() {
16         assertEquals(2, Calculator.pow(2, 1));
17     }
18     public void testPow3() {
19         assertEquals(27, Calculator.pow(3, 3));
20     }
21     public void testPow4() {
22         assertEquals(64, Calculator.pow(2, 6));
23     }
24 }
```

complain which assert statement failed but which test case failed. The consequence is that if there are more assert statements in one test case, the Debugger will not know which one failed.

In general this is the look of a JUnit 3.8.x test case. The test cases for JUnit versions higher than 3.8.x are not implemented yet.

4.1.2. Execution Trace

The extraction of the execution trace was implemented in the course of [12]. This Master thesis is about a ”Spectrum-based Debugger” and already needs the execution trace of a program. To avoid reinventing the wheel every time, the extraction is reused.

The extraction works as follows. First of all, the source files are modified by the JavaLexer.java and JavaParser.java files. Remember the running example in this thesis from Listing 2.1 and compare it to Listing 4.2. As can be seen, every statement from the original file is now represented by a SpectrumEntry. The SpectrumEntry contains the information of the add-call.

Listing 4.2: Clipping of the modified Java File

```
1  public static int pow(int a, int b){
2  if (!Spectrum.getInstance().containsEntry(idOfSpectrumEntry)) {
3    Spectrum.getInstance().add(0,0,72,84,"[base]:[a]","/path/to/
         PowerFunction.java",4,"0−/path/to/PowerFunction.java","0−/path/to
         /PowerFunction.java","none",null);}
4
5        int base = a;
6
7  if (!Spectrum.getInstance().containsEntry("idOfSpectrumEntry")) {
8    Spectrum.getInstance().add(1,0,88,104,"[exponent]:[b]","/path/to/
         PowerFunction.java",5,"1−/path/to/PowerFunction.java","0−/path/to
         /PowerFunction.java","none",null);}
9
10       int exponent = b;
11   ...
```

The following list gives the information with a short description.

- id: The ID of the statement. Every statement has its own unique ID.
- type: The type of the statement e.g. declaration, return statement, while-loop etc.
- begin: The index of the char character, where the statement in the original Java file begins.
- end: The index of the char character, where the statement in the original Java file ends.
- defRef: The variables which are defined and referenced in the statement.
- fileName: The path to the file.
- line: The line number in the original Java file.
- key: The key of this SpectrumEntry. It consists of the ”id-fileName”.
- methodId: The ID of the method to which the statement belongs. Every method of the original Java file has its own ID.
- parent: The parent of the statement. This is important for the control dependency of the relevant slicer.
- childDefs: The defined variables in the child nodes. It is a beginning for the implementation of the potential relevant variables and the potential data dependency.

For more details about this class see spectrum/SpectrumEntry.java.

If the test case is now run and a statement in the Java program is executed, this information about the statement will be stored in the so-called Spectrum. At the end of the test run the Spectrum contains the sequence of all called statements and of course the above described information about them. Since the Spectrum is implemented as Singleton it is guaranteed that there exists just one sequence and it can be accessed from everywhere.

4.1.3. Relevant Slicer

The algorithm for the relevant slicer was already described in Chapter 2.2.6. It must be noted that for the Debugger the algorithm is not really implemented in this way. The Debugger was built up on a "Spectrum-based Debugger" which was implemented in the course of [12] and already used a relevant slicer. This slicer works well for simple programs but has its problems with complex object-oriented programs, which use lots of member variables, inheritance and similar characteristics of high-level programming languages. For the first prototype of the Debugger it was decided, that it is sufficient to be able to solve simpler programs and test the benefits of the whole approach first. But in the near future the need for a good relevant slicer is obligatory.

The current implementation of the relevant slicer can be found in the file slicer/StaticSliceAnalyzer.java. With the comments it is quite simple to understand the implementation.

The call of the relevant slicer is not really complex either. Since it is a static method and the execution trace is already stored in the above mentioned Spectrum component, all that is needed is to call the slice method. The parameters therefore are the stored execution trace, the variable to slice for and a boolean variable which indicates if it should be sliced for the last defined variable in the Java program or for the passed one. Have a look at Listing 4.3 to see an example for a call.

Listing 4.3: The Call to start the relevant Slicer

```
LinkedHashSet<String> set =
    StaticSliceAnalyzer.analyzeTestcaseStaticSlice(
        Spectrum.getInstance().getExactlyStatementSequence(),
        var, false);
```

4.1.4. Minimal Hitting-Sets

As already mentioned in the theoretical chapter of this thesis there are plenty of algorithms which calculate minimal hitting-sets. In my Master-Project at the Institute for Software Technology at the Graz University of Technology [6] the implementation of two of them was necessary. The faster and more recent algorithm, namely Staccato, is reused. Staccato means STAtistiCs-direCted minimAl hiTing set algOrithm and was presented at the eighth Symposium on Abstraction, Reformulation and Approximation (SARA) in the year 2009. As mentioned in the paper, "STACCATO uses a heuristic borrowed from a statistics-based software fault diagnosis approach, called spectrum-based fault localization (SFL). SFL uses sets of component involvement in nominal and failing program executions to yield a ranking of components in order of likelihood to be at fault."[13] Due to the fact that in this thesis only the computation of minimal hitting-sets is relevant, the algorithm can be changed a little bit. The original version can be found in [13]

4.1.4.1. Changes

The following changes do not change the internal functionality of the algorithm. They just skip unnecessary aspects. Of course, the aspects are not unnecessary in the original sense but they are not needed for the purpose of this thesis.

- Spectrum-based Fault Localization (SFL)
  First of all it is not necessary to distinguish between an error set and a nominal set, therefore there is no need for the usage of SFL. Staccato normally takes error sets and nominal sets and ranks the components involved in these sets in order of their fault likelihood. Afterwards this ranking is used to focus the search.
  The Debugger needs to compute all minimal hitting-sets in order not to ignore the faulty statement and therefore does not need this ranking.

- Stopping Criteria
  The other change is that the original algorithm contains two stopping criteria, one which determines how many hitting-sets shall be found by the algorithm and another one which determines how many components shall be processed. This is not reasonable for the Debugger because there again the chance to ignore the faulty statement exists.

4.1.4.2. Implementation Details for Staccato

1. *Internal Data Structure*
   In the initialization phase the algorithm converts the conflict set to a two-dimensional matrix filled up with booleans. The conflict set contains, in the sense of the Debugger, the relevant slices of the different output variables.
   Assume the following conflict set S = {{1,3,5}, {1,2}, {2,4,5}, {3}} which contains four sets Set_1 .. Set_4 is transformed into the matrix in Table 4.1.
   Each row stands for a single set of the conflict set and each column for a specific component. If the

	1	2	3	4	5
Set_1	1	0	1	0	1
Set_2	1	1	0	0	0
Set_3	0	1	0	1	1
Set_4	0	0	1	0	0

Table 4.1.: Matrix after Initialisation

   component is part of the set there is a 1, otherwise there is a 0.
   The data structure in Java is a LinkedList⟨LinkedList⟨Boolean⟩⟩.

2. *Method STRIP-COMPONENT(matrix, index of the component)*
   This method removes the column of the specified component from the matrix. Taking the matrix from above and assuming that the index is 3, the result can be seen in Table 4.2. The column of the

	1	2	4	5
Set_1	1	0	0	1
Set_2	1	1	0	0
Set_3	0	1	1	1
Set_4	0	0	0	0

Table 4.2.: Matrix after calling Method stripComponent

   specified component, in this example it is the 3rd column, has been deleted.

3. *Method STRIP(matrix, index of the component)*
   This method returns a new two-dimensional matrix, in which the column of the specified component as well as all sets in which the specified component is involved, are removed. Taking again the original matrix and the index of 3, the result is shown in Table 4.3. The column of the component,

	1	2	4	5
Set_2	1	1	0	0
Set_3	0	1	1	1

Table 4.3.: Matrix after calling Method strip

again the 3rd column, and additionally the sets in which it was included, the 1st and 4th column, have also been deleted.

4. *Method IS-NOT-SUBSUMED(found hitting sets, set to check)*
   This method checks if the "set to check" is a subset of one or more sets already found in a deeper recursion. If it is true, those sets will be deleted and the new one will be added.

4.1.4.3. Modified Staccato Algorithm

After describing the changes and implementation details the modified algorithm is presented in Algorithm 3.

Algorithm 3 The Staccato Algorithm

Input: Matrix A, Components M
Output: minimal hitting-set D

1: **for all** $j \in M$ **do**
2: check if j is involved in all or none set
3: **if** $j \in$ all sets **then**
4: $push(D, j)$
5: $STRIP - COMPONENT(A, j)$
6: $remove(M, j)$
7: **else if** $j \in$ none set **then**
8: $STRIP - COMPONENT(A, j)$
9: $remove(M, j)$
10: **end if**
11: **end for**
12: **while** $M \neq 0$ **do**
13: $j = pop(M)$
14: $A' = STRIP(A, j)$
15: $STRIP - COMPONENT(A, j)$
16: $D' = STACCATO(A', M)$
17: **while** $D' \neq 0$ **do**
18: $j' = pop(D)$
19: $j' = j \cup j'$
20: **if** $IS - NOT - SUBSUMED(D, j')$ **then**
21: $push(D, j')$
22: **end if**
23: **end while**
24: **end while**
25: **return** D

Description of the algorithm

- Lines 1-11
  Here it is checked whether a certain component is involved in all sets or none of the sets. In the first

case it will be added to the minimal hitting-sets and the component is deleted from both structures, the matrix and the component stack. In the latter case it will only be deleted.

- Lines 12-24
  Until the component stack is not empty, the algorithm will take every component of the stack, strip the matrix with it and recall itself again with the stripped matrix and remaining components. (Lines 12-16)
  After returning a result the component will be unionised with this result and it will be checked if the new set is a minimal hitting-set or not. (Lines 17-24)

4.1.4.4. Usage in the Debugger

The last important point is how to use the Staccato algorithm to calculate all minimal hitting-sets. In Listing 4.4 a fragment of the Debugger can be seen. The code is from the Debugger.java file.

Listing 4.4: The call of Staccato to calculate all minimal Hitting-Sets

```
123 AlgorithmInterface algo = new Staccato(Parser.getElementsFrom(
        container), container);
124 algo.compute(-1, -1, -1);
125 SubSetContainer result = algo.returnDiagnoses();
```

The handling of the algorithm is not really hard. The only point to mention is the ”SubSetContainer”. It represents the data structure the algorithm works with. Every slice is represented by a SubSet and as a consequence of that, a SubSetContainer consists of one or more SubSets. Have a look at the example in Figure 4.2 where it is graphically demonstrated.

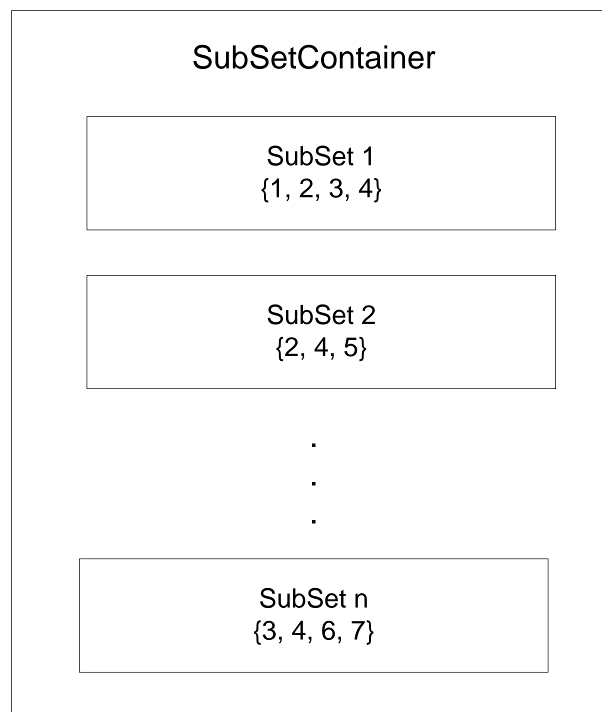

Figure 4.2.: A SubSetContainer with SubSets

All files related to the calculation of minimal hitting-sets can be found in the package hittingSet.

4.1.5. Constraints

There are two classes which are relevant for the handling of the constraints, namely the Converter.java and the TestCase2Convert.java. The Converter is responsible for converting the execution trace of the Java program to constraints in Minion and the TestCase2Convert contains then all necessary information about the program. This includes the execution trace as well as the input and output variables with its values.

4.1.5.1. Converter

The Converter is responsible for converting only the execution trace of a Java program. Wotawa stated in [17] that this is done in three steps but this turns out to be not entirely correct.

1. Eliminating Test Elements
   The first step is the problematic step. According to [17] the test elements can be deleted because "an execution trace stores one particular path through the program. The while or conditional statements that are part of this path are known to be evaluated to true or false. The test elements cause the execution of statements but these statements are elements of the execution trace. Consequently, we eliminate all test elements..."

   Eliminating these test elements has two consequences for the Debugger. First, it is not possible to check conditional and while statements for consistency. Therefore they have to be automatically added to the set of possibly faulty statements because these statements can also be the cause of the debugging problem. Second, by eliminating these statements the relationship between variables may also be eliminated, which leads to wrong results. See Chapter 6.1 for a precise description of the problem and a solution approach.
   In the implementation of the prototype the test elements are eliminated from the conversion and automatically added to the possibly faulty statements.

2. Static Single Assignment (SSA) Conversion
   The next step is to convert the statements left to a static single assignment form. This means that no variables on the left side, called defined variables, should have the same name because Minion can not solve a CSP including the same variable assigned to different values. It is impossible in this case not to violate the constraints.
   To implement this conversion it is enough to add an index to the defined variable which starts with 0 and increases every time the variable is redefined.

3. Constraint Conversion
   To convert the path of a Java program to Minion the equivalent constraints have to be found and used in the right way. Since Minion does only support boolean and integer variables the Debugger is also limited to these two primitive data types. In Table 4.4 the conversion of a Java statement to a Minion constraint is shown.

Although a work-around for multiplication and division of negative variables is implemented in the Converter, the division in Minion cannot even handle negative numbers. Therefore be aware of this when using negative numbers or negative variables in combination with the division constraint. The same problem exists for the modulo operation.

Further information and more details about the constraints used by the Debugger can be found in the manual on the Minion homepage [3].

4.1.5.2. TestCase2Convert

The TestCase2Convert contains all information about the executed Java program. This includes the above described conversion of the execution trace to Minion constraints as well as the test case itself with the input

<table>
<tr><th>Java</th><th>Minion</th><th>Comment</th></tr>
<tr><td>int a</td><td>DISCRETE a {x..x}</td><td>the range between the brackets are the domain of the int variable</td></tr>
<tr><td>boolean a</td><td>BOOL a</td><td>boolean variables does not need a domain</td></tr>
<tr><td>int[] a</td><td>DISCRETE a[size] {x..x}</td><td>arrays cannot resized in Minion</td></tr>
<tr><td>bool[] a</td><td>Bool a[size]</td><td>arrays cannot resized in Minion</td></tr>
<tr><td>a = b</td><td>eq(a,b)</td><td>assignment of two variables</td></tr>
<tr><td>a = 234</td><td>eq(a,234)</td><td>assignment of a int variable</td></tr>
<tr><td>a = false</td><td>eq(a,0)</td><td>assignment of a boolean variable to false</td></tr>
<tr><td>a = true</td><td>eq(a,1)</td><td>assignment of a boolean variable to true</td></tr>
<tr><td>a = b[1]</td><td>eq(a,b[1])</td><td>works as long as the index is a number and not a variable itself</td></tr>
<tr><td>a = b[i]</td><td>element(b, i, tmpVar1)
eq(a, tmpVar1)</td><td>store variable b[i] in a temporarily variable first and use this variable afterwards</td></tr>
<tr><td>c = a + b

c = a - b</td><td>weightedsumgeq([1,1][a,b],c)
weightedsumleq([1,1][a,b],c)
weightedsumgeq([1,-1][a,b],c)
weightedsumleq([1,-1][a,b],c)</td><td>the first vector represents the sign of the integers - 1 stands for a positive number and -1 for a negative number</td></tr>
<tr><td>c = a * b</td><td>product(a,b,c)</td><td></td></tr>
<tr><td>c = a * -b</td><td>product(b,-1,tmpVar1)
product(a,tmpVar1,c)</td><td>Minion does not support multiplication with negative variables - negative numbers will work</td></tr>
<tr><td>c = a / b</td><td>div(a,b,c)</td><td>Calculates $\lfloor \frac{a}{b} \rfloor$ and is only allowed for positive numbers</td></tr>
<tr><td>c = a / -b</td><td>product(b,-1,tmpVar1)
div(a,tmpVar1,c)</td><td>same work-around as by multiplication with negative variables</td></tr>
<tr><td>c = a % b</td><td>modulo(a,b,c)</td><td>the modulo operator just works with values >= 1</td></tr>
<tr><td>a != b</td><td>diseq(a,b)</td><td></td></tr>
<tr><td>c = a&&b</td><td>reify(sumgeq([a,b],2),c)</td><td></td></tr>
<tr><td>c = a || b</td><td>reify(sumgeq([a,b],1),c)</td><td></td></tr>
</table>

Table 4.4.: Conversion of a Java Statement to a Minion Constraint

and output variables. A runnable Minion file can be created out of the String which the method getContentOfMinionFile(SubSet minimalHittingSet) returns. The parameter in this case is a minimal hitting-set that has to pass the consistency check to be a valid diagnose.
This object represents, generally speaking, the whole test case in Minion format.

4.1.5.3. Consistency Check

There are two additional Minion constraints necessary for checking a converted execution trace for consistency, namely the "watched-or" and the "element" constraint.

The abnormal array and the "element" constraint
First of all it is necessary to define a boolean array with the size of the converted statements. Afterwards every element of the array is intended to be 1 and added to the corresponding line, this means the first element of the array, ab[0], to the first converted line of the Java program, ab[1] to the second one and so on. Note that a single statement in the Java program can be converted to Minion with the help of more than one constraint. In this case it always has to be the same abnormal variable which corresponds to these constraints.

The "watched-or" constraint
The above described mapping will now be combined in a "watched-or" constraint. This constraint ensures that one of the two conditions must be satisfied. In other words, either the element of the array is 1 or the converted statement has to be fulfilled.

The last part needed is the diagnose which shall be tested. Remember that the algorithm for calculating minimal hitting-sets calculates these diagnoses. The abnormal variable that belongs to the line(s) of the diagnose is, resp. are, set to 1 with the help of the "eq" constraint. The other elements are set to 0. These settings can be found in the "#TESTCASE" section of the CSP. After running Minion the diagnose is either consistent and valid or inconsistent and not valid.

In Listing 4.5 a short example is demonstrated. It is obvious that the program does not make much sense but it is good to demonstrate how the consistency check works. Assume the input a = 3 and b = 5 and an expected output of 14. By setting the ab[0] to 0, the CSP is not solvable because the "watched-or" constraint cannot ensure that one of both conditions is true. The "element" constraint cannot be true because in line 10 ab[0] is set to 0 and the "product" constraint also cannot be true because a * b cannot be 14. Setting ab[0] to 1 in line 10 will cause the CSP to be solvable because the "watched-or" constraint can then ensure that the "element" constraint is true.

Listing 4.5: Short Minion Program to demonstrate Consistency checking

```
1  MINION 3
2    **VARIABLES**
3      BOOL ab[1]
4      DISCRETE a {0..92}
5      DISCRETE b {0..92}
6      DISCRETE result {0..92}
7    **CONSTRAINTS**
8      watched-or({element(ab,0,1),product(a, b, result)})
9    # TESTCASE
10     eq(ab[0],0)
11     eq(a,3)
12     eq(b,5)
13     eq(result,14)
14 **EOF**
```

Listing 3.1 shows a larger example.

4.2. Limitations

Many of the following limitations have already been mentioned in the previous chapters. This section now summarizes them all with the intention of having kind of a ”don’ts” list.

- **General**
  - Works only with Java programs.
  - Use one statement per line.
- **Test File**
  - The test class has to be inherited from junit.framework.TestCase.
  - The test class has to have a constructor which takes a String as argument and calls the constructor of the base class.
  - Each test case has to start with ”test”.
  - Each test case shall only contain one assert statement.
- **Relevant Slicer**
  - Slices for large object-oriented programs are wrong.
  - Slices for programs with frequent use of member variables are sometimes wrong.
  - The relevant Slicer works best for simpler programs which do not use a lot of high-level programming language concepts.
- **Constraints**
  - Just integer, boolean and arrays of those types can be converted to Minion because Minion can not handle other types of variables.
  - Only if-then-else statements and while-loops are supported.
  - Assignments to an array after it has been already initialized are not supported yet.
  - Instead of using short versions like i++ or i += 4 or similar, use the normal assignment i = i + 1.
  - Do not use the ”?”-operator to shorten if-statements. Use the if-then-else construct.
  - The range of possible values for the variables in Minion is limited from -10 to 128. The division and modulo constraint have problems with a negative range.

4.3. UML

This section contains the class diagram with a short description of the tasks of the individual classes and the sequence diagrams for the analyzing and debugging task.

4.3.1. Class Diagram

The class diagram in Figure 4.3 shows the relationships between the classes. In the following a short description of every class is given. For more detailed information have a look at the source code or the Javadoc generated out of it. To keep the diagram well-structured and understandable only the important methods of every class are listed.

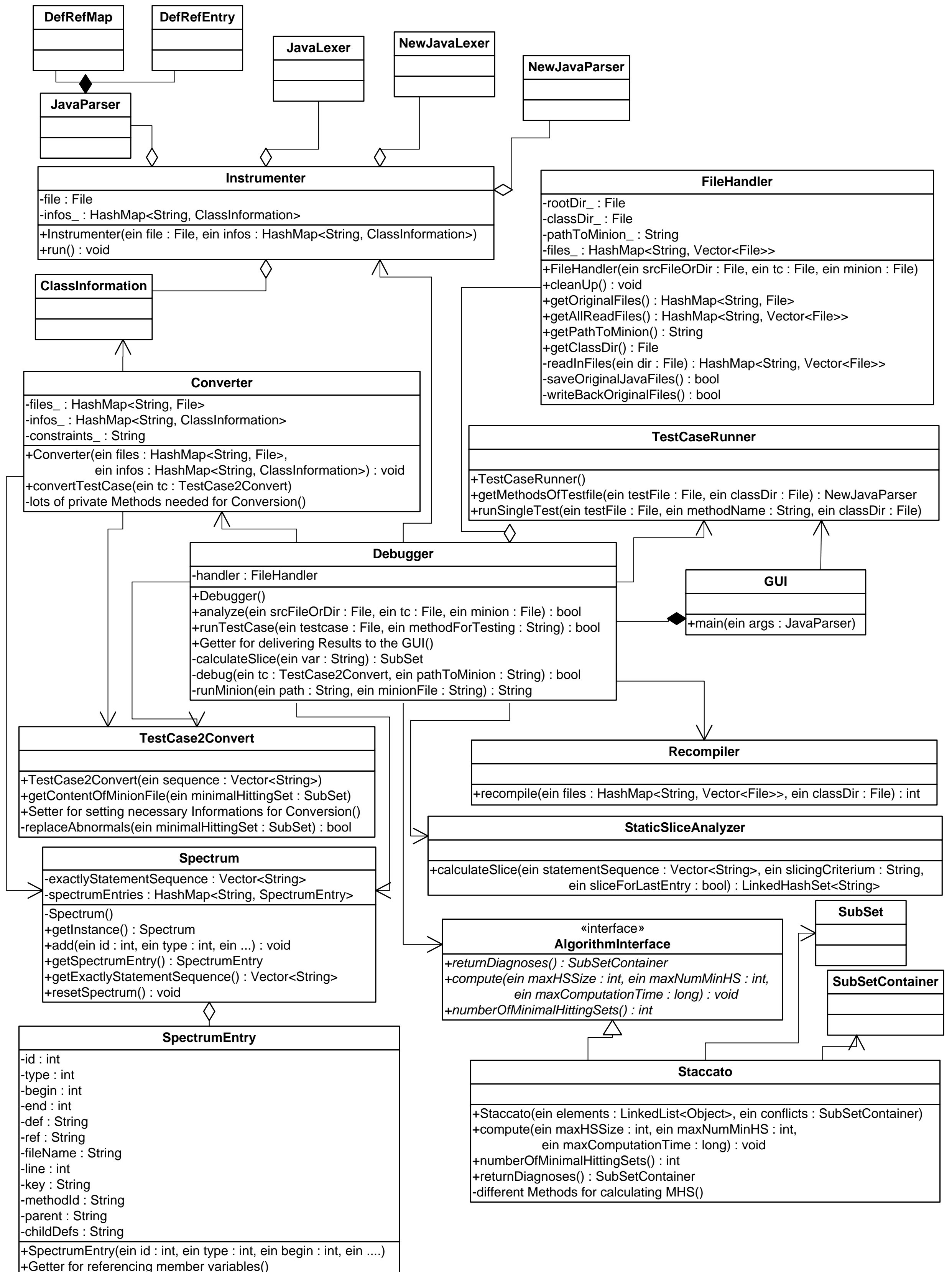

Figure 4.3.: The Class Diagram

- GUI
  The graphical user interface of the Debugger. First, the user enters the source files or directory, the test file and the path to Minion and lets them be analyzed. Next, the test method has to be chosen and the input and output variables with their corresponding types and values have to be entered or read in. Last, the whole program will be debugged and the results are presented.
- Debugger
  In this class all necessary actions for the debugging approach are done. The first step hereby is to prepare everything, this means to read in all of the files, create the class directory and analyze and recompile them. The second step is to run the test case followed by the third and last step, to convert the execution trace to Minion, calculate the slices of the entered output variables and run Minion to eliminate impossible faulty candidates.
- *FileHandler*
  The class is responsible for handling all required files. This includes all the source files, the test file and the jars needed for execution of the program. The source files will be read from the hard disk, temporarily modified and later on restored with the original content. Furthermore it is responsible for creating and deleting the class directory and class files.
- Converter
  This class is responsible for converting the execution trace of the Java program to Minion constraints. Other important information, like the input and output, is not converted here but direct in the TestCase2Convert class.
- TestCase2Convert
  Stores all relevant test run information that is necessary to generate the content of the Minion file.
- *Recompiler*
  This class recompiles the given Java program and stores the class files in the class directory.
- *TestCaseRunner*
  Handles the usage of the test file. It provides a method to get all test methods of the test file and runs a specific test case.
- *Instrumenter*
  This class analyzes the original Java files with the help of the automatically generated Lexer and Parser. There are two analyzing procedures. The first analysis is for filtering out important information about the file itself (needed for the class ClassInformation) and the second one is to modify the original Java files to get the execution trace and information about the individual statements. (needed for the SpectrumEntry)
- ClassInformation
  Stores necessary information of the original Java classes. This Information is mainly necessary for a conversation to Minion.
- *JavaParser*
  Automatically generated Parser from an ANTLR-Grammar file. The grammar file was from project [12] and is not available any more.
- *JavaLexer*
  Automatically generated Lexer from the above mentioned ANTLR-Grammar file.
- *DefRefMap*
  Helper class for the JavaParser.
- *DefRefEntry*
  Helper class for the JavaParser.
- NewJavaParser
  Because the original grammar file was not available and additional information was necessary for the conversation the only possibility was to create a new ANTLR-Grammar file and generate new Parser and Lexer.

- NewJavaLexer
  Automatically generated Lexer from the new ANTLR-Grammar file.
- *StaticSliceAnalyzer*
  The slicer of the Debugger. It slices the execution trace for a specific variable or for the referenced variables of the last statement of the execution trace.
- *Spectrum*
  The Spectrum class is implemented as Singleton and stores the execution trace of a Java program. Every executed line in the program is represented as SpectrumEntry.
- *SpectrumEntry*
  This class represents a statement in the original Java program.
- AlgorihtmInterface
  Interface for the Staccato algorithm.
- Staccato
  This class contains the ”Low-Cost Approximate Minimal Hitting-Set Algorithm”, named STACCATO, by Rui Abreu and Arjan J. C. van Gemund. [13]
- SubSet
  The algorithm works with SubSets that represent a set of different objects. In the case of the Debugger it is the slice of a single output variable.
- SubSetContainer
  Contains all of the Subsets to calculate the minimal hitting-sets.

The italic marked classes are partially taken from the thesis in [12] and were modified to the purpose of the Debugger.

4.3.2. Sequence Diagrams

In this section the two main tasks - analyze the Java program and debug it - are explained with the help of sequence diagrams.

- **Analyzing the Program**

The user of the Debugger has to enter the source file or directory, the path to the test file and to the Minion executable. After pressing the analyze button the debugger starts. First the FileHandler is called to read in the source file or all files of the chosen directory and the test file. Thereby a class directory is also created. Afterwards the Debugger sets the class path for the Java program and calls the Instrumenter to modify the Java files and parse out important information. This is done via automatically generated Parsers and Lexers. The generation was done via ANTLR-Grammar files. The next step is to recompile the Java program to generate the class files, which are needed for the run of the test case. The last step in the analyzing phase is to parse out all test methods of the test file. If everything has been successfully done, the user can now choose the test method and enter the input and output variables. This procedure can be seen in Figure 4.4.

- **Debug the Program**

The second main task, namely the debugging of a faulty Java program, see Figure 4.5, starts after the user has chosen the test method and has entered the input and output variables. At the beginning the Debugger runs the test case. If it is a passing test case, the Debugger will be finished, otherwise a TestCase2Convert object is created and the statement sequence, which was traced by the Spectrum, is set. Then the debug method is called. The first step in this method is to convert the statements to Minion. Afterwards the slices for the entered output variables are calculated. The last step is to calculate the minimal hitting-sets among the slices. The last general step before the Debugger has finished its work is to call the Minion executable to eliminate inconsistent candidates. The results are now presented in the GUI.

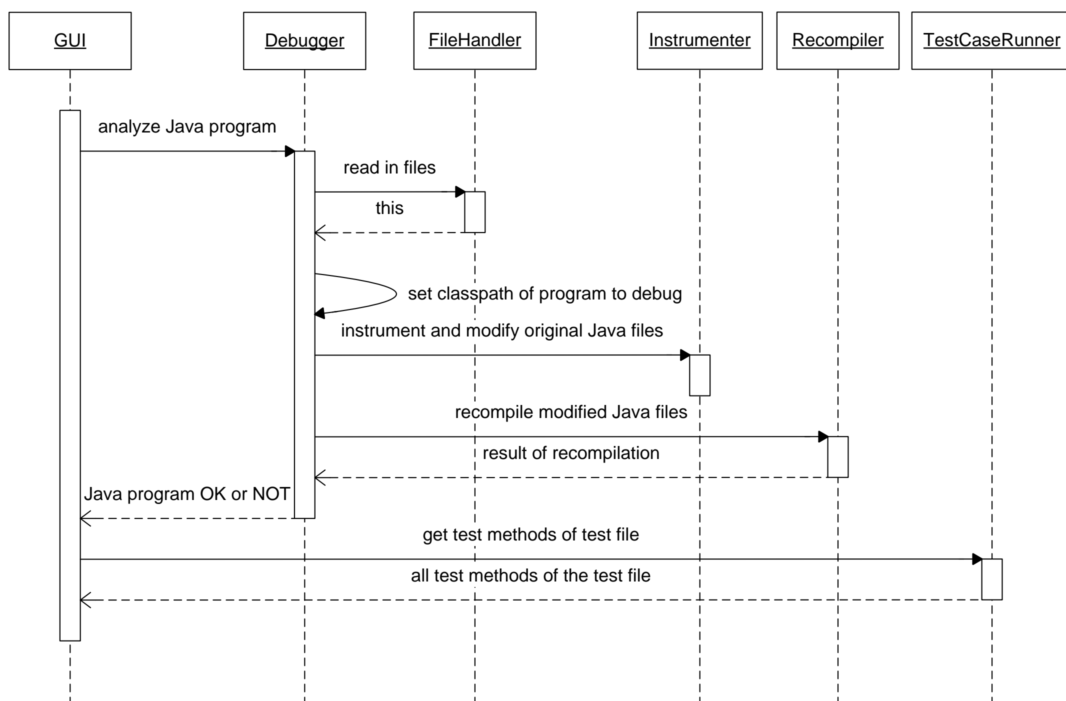

Figure 4.4.: Sequence Diagram for Analyzing

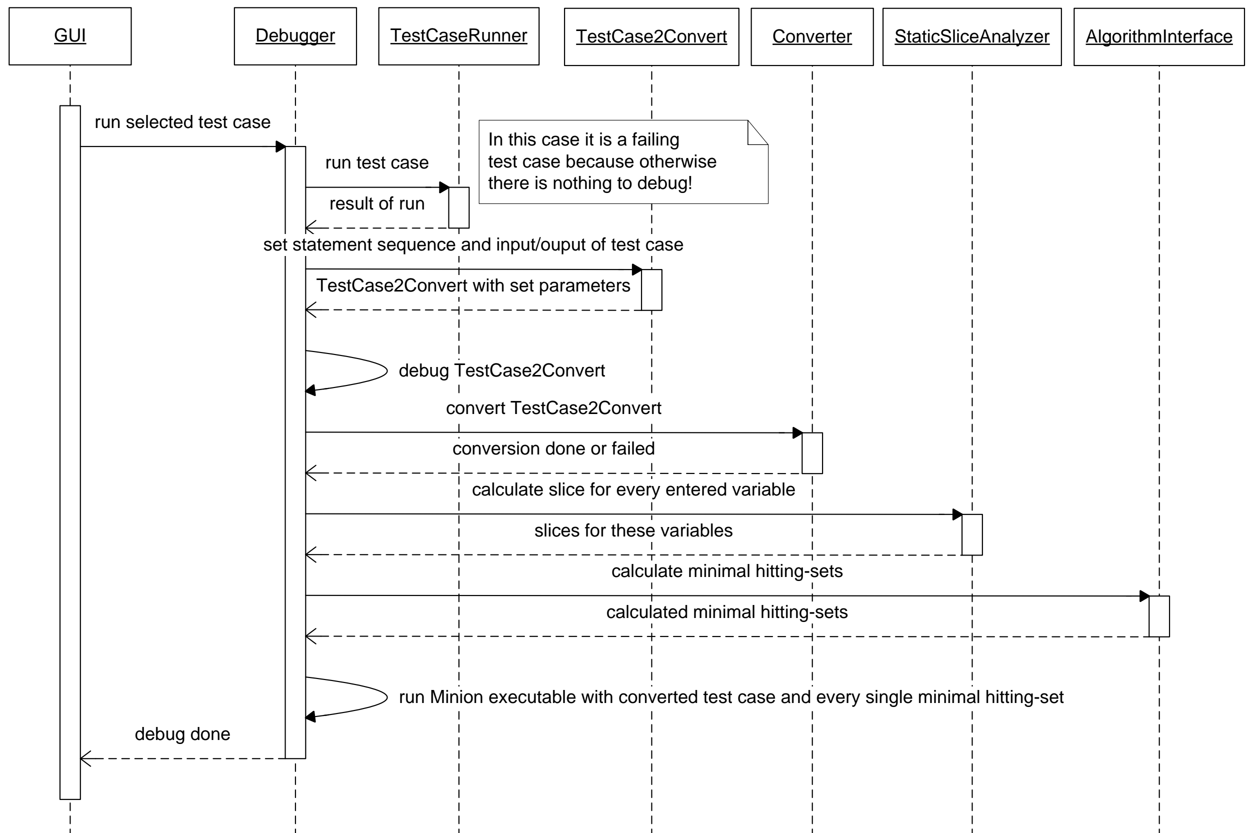

Figure 4.5.: Sequence Diagram for Debugging

Chapter 5

Test Results

At the beginning of this chapter the details of the testing platform are listed. Then the test results are presented and finally, a short summary is given.

5.1. Test Platform

The tests were run on an Intel Core Duo CPU with 2.20 GHz and 3.25 GB memory. In order not to be influenced by certain circumstances on the reference platform each test case was carried out 3 times and afterwards the average of all runs was calculated and represented in the following graphs and tables. The OS was Microsoft Windows XP Professional with Service Pack3.

5.2. Single Test Files

This section contains single test files which were found in various papers relating to debugging. Therefore they are a good base for producing first results with the prototype of the Debugger.

Every test consists of three parts. First the source code, second the test cases and third the test results. In the following a short description of the tables of the test cases and the test results is given.

Test Cases

- Test Case: The name of the test case in the test file.
- Input: The name and values of the input variables.
- Output: The name and values of the output variables.
- E: The line with the error.
- original Line: The ”right” line of the program.
- modified Line: The ”wrong” line of the program.

To reproduce the test results it is necessary to replace the original line with the modified line and run the test case with the given input and output variables.

Test Results

- File: The name of the tested file.

- Method to test: The name of the tested method, in case of having more than one method in a Java file.
- Time: The time needed for debugging the program.
- Statements: The amount of lines of the tested method(s).
- relevant Slicer: The amount of lines which is returned after running the relevant Slicer and the calculation of minimal hitting-sets.
- Result: The amount of lines the Debugger returns. This further excludes the lines which do not pass the consistency check.
- faulty Line: If the faulty line is contained in the result set or not.

PowerFunction

This short method represents the running example of this thesis.

Listing 5.1: PowerFunction.java

```
1  public class PowerFunction {
2
3    public static int pow(int a, int b) {
4      int base = a;
5      int exponent = b;
6      int tmp = 0;
7      if(exponent == 0)
8        tmp = 1;
9      else
10       tmp = base;
11
12     int counter = 1;
13     while(exponent > counter) {
14       tmp = tmp + base;
15       counter = counter + 1;
16     }
17     int result = tmp;
18     return result;
19   }
20 }
```

Test Case	Input	Output	E	original Line	modified Line
Pow3	a = 3, b = 3	result = 27	14	tmp = tmp + base;	tmp = tmp * base;
Pow4	a = 2, b = 6	result = 64	14	tmp = tmp + base;	tmp = tmp * base;

Table 5.1.: Test Cases for PowerFunction.java

AKSWT1

The method is taken from the exercise of the course ”AK Softwaretechnologie 1” in the winter term 2010/11 at the Graz University of Technology.

File	Method to test	Time	Statements	relevant Slicer	Result	faulty Line
PowerFunction.java	pow3	505 ms	12	9	6	Y
PowerFunction.java	pow4	525 ms	12	9	5	Y

Table 5.2.: Test Results for PowerFunction.java

Listing 5.2: AKSWT1.java

```
1  public class AKSWT1 {
2
3    public static int Example(int a, int b) {
4      int tmp = a;
5      int result;
6      if (b == 0) {
7        result = -1;
8      } else {
9        result = 0;
10       while (tmp > 0) {
11         result = result + 1;
12         tmp = tmp - b;
13       }
14     }
15     return result;
16   }
17 }
```

Test Case	Input	Output	E	original Line	modified Line
testExample	a = 5, b = 2	result = 3	11	result = result + 1;	result = result + 2;
testExample1	a = 6, b = 2	result = 3	4	int tmp = a;	int tmp = a + 1;
testExample2	a = 6, b = 2	result = 3	9	result = 0;	result = 1;
testExample3	a = 6, b = 0	result = -1	7	result = -1;	result = 1;
testExample4	a = 8, b = 2	result = 4	10	while (tmp > 0)	while (tmp >= 0)

Table 5.3.: Test Cases for AKSWT1.java

File	Method to test	Time	Statements	relevant Slicer	Result	faulty Line
AKSWT1.java	Example	280 ms	9	6	4	Y
AKSWT1.java	Example1	302 ms	9	6	4	N
AKSWT1.java	Example2	297 ms	9	6	4	Y
AKSWT1.java	Example3	82 ms	9	2	2	Y
AKSWT1.java	Example4	297 ms	9	6	4	Y

Table 5.4.: Test Results for AKSWT1.java

ProdSum

This example was already published in [17].

Listing 5.3: ProdSum.java

```
1  public class ProdSum {
2
3    public static int Example(int[] a, int size) {
4      int sum = 0;
5      int i = 0;
6      while( i < size ) {
7          sum = sum + a[i];
8          i = i+1;
9      }
10     int prod = sum * (size + 1);
11     int all = 0;
12     if (size > 0) {
13       all = sum + size;
14     } else {
15       all = sum * size;
16     }
17     return prod;
18   }
19 }
```

Test Case	Input	Output	E	original Line	modified Line
testExample_f1	a = [1,1], size = 2	prod = 6	10	int prod = sum * (size+1);	int prod = sum * size;
testExample_f2	a = [1,1], size = 0	prod = 0	4	int sum = 0;	int sum = 1;

Table 5.5.: Test Cases for ProdSum.java

File	Method to test	Time	Statements	relevant Slicer	Result	faulty Line
ProdSum.java	Example_f1	380 ms	11	6	4	Y
ProdSum.java	Example_f2	244 ms	11	4	3	Y

Table 5.6.: Test Results for ProdSum.java

Multiplication

Listing 5.4: Multiplication.java

```
1  public class Multiplication {
2
3    public static int multiply_f1(int factor1, int factor2) {
4      int sign = 1;
5
6      if(factor1 < 0) {
7        sign = sign * -1;
8        factor1 = factor1 * -1;
9      }
10
11     if(factor2 < 0) {
12       sign = sign * -1;
```

```
13        factor2 = factor2 * -1;
14      }
15
16      int result = 0;
17      int tmp = factor1;
18      while(factor2 > 1) {
19        tmp = tmp + factor1;
20        factor2 = factor2 - 1;
21      }
22
23      int finalResult = tmp * sign;
24      return finalResult;
25    }
26  }
```

Test Case	Input	Output	E	original Line	modified Line
testMultiply_f1	factor1=4, factor2=-2	finalResult=-8	23	finalResult = tmp*sign;	finalResult = tmp*-sign;
testMultiply_f2	factor1=3, factor2=5	finalResult=15	19	tmp = tmp+factor1;	tmp = tmp*factor1;
testMultiply_f3	factor1=-2, factor2=-9	finalResult=18	11	if(factor2<0)	if(factor2>0)

Table 5.7.: Test Cases for Multiplication.java

File	Method to test	Time	Statements	relevant Slicer	Result	faulty Line
Multiplication.java	multiply_f1	481 ms	14	10	7	Y
Multiplication.java	multiply_f2	335 ms	14	8	5	Y
Multiplication.java	multiply_f3	350 ms	14	8	6	Y

Table 5.8.: Test Results for Multiplication.java

Division

This implementation of a division is quite similar to the one in [12].

Listing 5.5: Divide.java

```
1   public class Divide {
2
3     public static int divide_f1(int divident, int divisor) {
4       int sign = 1;
5
6       if(divident < 0) {
7         sign = sign * -1;
8         divident = divident * -1;
9       }
10
11      if(divisor < 0) {
12        sign = sign * -1;
13        divisor = divisor * -1;
14      }
15
```

```
16      int result = 0;
17      while(divident >= divisor) {
18        result = result + 1;
19        divident = divident - divisor;
20      }
21
22      int finalResult = result * sign;
23      return finalResult;
24    }
25  }
```

Test Case	Input	Output	E	original Line	modified Line
testDivide_f1	divident=15, divisor=-3	finalResult=-5	12	sign = sign * -1;	sign = sign * 1;
testDivide_f2	divident=-10, divisor=-2	finalResult=5	18	result = result + 1;	result = result - 1;
testDivide_f3	divident=20, divisor=5	finalResult=4	16	int result = 0;	int result = 1;

Table 5.9.: Test Cases for Divide.java

File	Method to test	Time	Statements	relevant Slicer	Result	faulty Line
Divide.java	divide_f1	533 ms	13	10	8	Y
Divide.java	divide_f2	739 ms	13	12	9	Y
Divide.java	divide_f3	359 ms	13	8	6	Y

Table 5.10.: Test Results for Divide.java

GCD

The calculation of the greatest common divisor can also be found at `http://rosettacode.org/wiki/Greatest_common_divisor#Iterative_Euclid.27s_Algorithm`.

To be able to test this method it is necessary to manually change the domain of the variables in the CSP because the ”modulo” constraint in Minion cannot handle variables with the domain $<= 0$. Afterwards the consistency check has to be done by hand, running Minion with every single constraint satisfaction problem. Doing this by hand is also the reason for the missing time measurement in Table 5.12.

Listing 5.6: GCD.java

```
1   public class GCD {
2
3     public static int gcd_f1(int a, int b) {
4       while(b > 0)
5       {
6         int c = a % b;
7         a = b;
8         b = c;
9       }
10      return a;
11    }
12  }
```

Test Case	Input	Output	E	original Line	modified Line
testGCD_f1	a = 14, b = 10	a = 2	7	a = b;	a = b*2;
testGCD_f2	a = 9, b = 6	a = 3	6	int c = a % b;	int c = a % b-2
testGCD_f3	a = 14, b = 9	a = 1	8	b = c;	b = 0;

Table 5.11.: Test Cases for GCD.java

File	Method to test	Time	Statements	relevant Slicer	Result	faulty Line
GCD.java	gcd_f1	– ms	5	4	3	Y
GCD.java	gcd_f2	– ms	5	4	2	Y
GCD.java	gcd_f3	– ms	5	3	2	N

Table 5.12.: Test Results for GCD.java

5.3. JADE Test Cases

The JADE project was run at the Vienna University of Technology under the administration of Markus Stumptner and Franz Wotawa. A short description of the project is, according to [1], the following. ”The JADE project dealt with the automatic localization of source code bugs in Java programs.” There are numerous test cases published on the homepage under the section ”Testcases”. These test cases are also a good base for testing the Debugger.

There are two points to be mentioned.

1. Because the relevant slicer is not that good with slicing complexer programs, the use of the simpler test cases is necessary.
2. The test cases have to be modified in such a way as to represent JUnit test cases, but the original intent of the test cases stays the same.

In the following some of the test cases are presented together with their results. All of them can be found on the homepage of the JADE project [1].

BinSearch

To be able to test this method it is necessary to manually change the domain of the variables in the CSP because the ”div” constraint in Minion cannot handle negative variables. In an analogously manner to the GCD.java the consistency check has to be done by hand and the needed time for debugging cannot be reasonably measured.

Listing 5.7: BinSearch.java

```
1  public class BinSearch {
2
3    public static int binSearch(int value, int[] values, int start,
       int length) {
4      int result = -1;
5      while ((length > 0) && (result < 0)) {
6        if (length == 1) {
7          if (values[start] == value)
8            result = start + 1;
9          else
10           length = 0;
```

```
11            } else {
12              int m = start + length / 2;
13              if (value < values[m])
14                length = length / 2;
15              else {
16                length = length - length / 2;
17                start = m;
18              }
19            }
20        }
21        return result;
22      }
23    }
```

Test Case	Input	Output	E	original Line	modified Line
testBinSearch	value = 21, values = [1,7,13,20,21,30,40,50,60,61], start = 0, length = 10	result = 4	8	result = start;	result = start + 1;

Table 5.13.: Test Cases for BinSearch.java

File	Method to test	Time	Statements	relevant Slicer	Result	faulty Line
BinSearch.java	binSearch	– ms	12	10	9	Y

Table 5.14.: Test Results for BinSearch.java

IfExamples

Listing 5.8: IfExamples.java

```
1  public class IfExamples {
2
3      public static int test1(int a, int b) {
4      int i;
5      i = 0;
6      if (a>0) {
7          i = 2*b;
8      }
9      return i;
10     }
11
12     public static int test2(int a, int b) {
13     int i;
14     if(a>0) {
15         i = 2*b;
16     } else {
17         i = 4*b;
18     }
19     return i;
20     }
```

```
21
22      public static int test3(int a, int b) {
23      int i;
24      if (a == 0) {
25          if (b>0) {
26            i = 1;
27          } else {
28            i = 2;
29          }
30      } else {
31          if (b>0) {
32            i = 10;
33          } else {
34            i = 4;
35          }
36      }
37      return i;
38      }
39
40  }
```

Test Case	Input	Output	E	original Line	modified Line
test_test1	a = 1, b = 2	i = 4	7	i = 2*b;	i = 2*i;
test_test2	a = 0, b = 2	i = 8	17	i = 4*b;	i = 3*b;
test_test3	a = 1, b = 2	i = 10	32	i = 10;	i = 3;

Table 5.15.: Test Cases for IfExamples.java

File	Method to test	Time	Statements	relevant Slicer	Result	faulty Line
IfExamples.java	test1	135 ms	5	3	3	Y
IfExamples.java	test2	72 ms	5	2	2	Y
IfExamples.java	test3	73 ms	9	3	3	Y

Table 5.16.: Test Results for IfExamples.java

SumPowers

Listing 5.9: SumPowers.java

```
1  public class SumPowers{
2
3    static int sumpowers_f1(int from, int to, int z) {
4      int start;
5      int stop;
6
7      if (from < to) {
8        start = from;
9        stop = to;
10     } else {
11       start = to;
```

```
12        stop   = from;
13      }
14
15      int sum = 0;
16      int i = start;
17      while (i < stop) {
18        sum = sum + power(z, i);
19        i = i + 1;
20      }
21       return sum;
22    }
23
24    static int power(int x, int e) {
25      int power = 1;
26      while(e>0) {
27        power = power * x;
28        e = e - 1;
29      }
30      return power;
31    }
32
33  }
```

Test Case	Input	Output	E	original Line	modified Line
test_sumpowers_f1	from = 1, to = 3, z = 3	sum = 12	17	while (i < stop)	while (i <= stop)
test_sumpowers_f2	from = 1, to = 3, z = 3	sum = 12	7	if (from < to)	if (from > to);
test_sumpowers_f3	from = 1, to = 3, z = 3	sum = 12	8,9	start = from; stop = to;	start = to; stop = from;
test_sumpowers_f4	from = 1, to = 3, z = 3	sum = 12	18	sum = sum + power(z, i);	sum = power(z, i);
test_sumpowers_f5	from = 1, to = 3, z = 3	sum = 12	27	power = power * x;	power = power + x;

Table 5.17.: Test Cases for SumPowers.java

File	Method to test	Time	Statements	relevant Slicer	Result	faulty Line
SumPowers.java	sumpowers_f1	726 ms	18	13	7	Y
SumPowers.java	sumpowers_f2	276 ms	18	6	3	Y
SumPowers.java	sumpowers_f3	273 ms	18	6	3	N
SumPowers.java	sumpowers_f4	593 ms	18	12	6	Y
SumPowers.java	sumpowers_f5	703 ms	18	13	8	Y

Table 5.18.: Test Results for SumPowers.java

TrafficLight

Listing 5.10: TrafficLight.java

```
1  public class TrafficLight {
2
3    public static int method1_f1() {
4      int i = 0;
```

```
 5      int state = 0;
 6
 7      while (i < 10) {
 8        if (state == 0) {
 9          state = 1;
10        } else {
11          if (state == 1) {
12            state = 2;
13          } else {
14            if (state == 2) {
15              state = 3;
16            } else {
17              state = 0;
18            }
19          }
20        }
21        i = i + 1;
22      }
23      return state;
24    }
25  }
```

Test Case	Input	Output	E	original Line	modified Line
testMethod1_f1	none	state = 2	17	state = 0;	state = 3;
testMethod1_f2	none	state = 2	9	state = 1;	state = 0;

Table 5.19.: Test Cases for TrafficLight.java

File	Method to test	Time	Statements	relevant Slicer	Result	faulty Line
TrafficLight.java	method1_f1	536 ms	12	11	5	Y
TrafficLight.java	method1_f2	310 ms	12	6	3	Y

Table 5.20.: Test Results for TrafficLight.java

WhileLoops

Listing 5.11: WhileLoops.java

```
 1  public class WhileLoops {
 2
 3    static public int test1() {
 4      int a,b,c,d,i;
 5      i = 1;
 6      a = 1;
 7      b = 2;
 8      c = 3;
 9      d = 4;
10      while (i < 10) {
11        a = b;
12        b = c;
```

```
13          c = d;
14          i = i + 1;
15        }
16        return a;
17    }
18  }
```

Test Case	Input	Output	E	original Line	modified Line
test_test1_f1	none	a = 4	9	d = 4;	d = 5;
test_test1_f3	none	a = 4	13	c = d;	c = b;

Table 5.21.: Test Cases for WhileLoops.java

File	Method to test	Time	Statements	relevant Slicer	Result	faulty Line
WhileLoops.java	test1_f1	478 ms	12	7	5	Y
WhileLoops.java	test1_f3	496 ms	12	7	5	Y

Table 5.22.: Test Results for WhileLoops.java

WhileLoops2

Listing 5.12: WhileLoops2.java

```
1   public class WhileLoops {
2
3     static public int test2() {
4         int a,b,i,j;
5         i = 0;
6         j = 0;
7         a = 0;
8         b = 0;
9         while (i < 5) {
10            j = 0;
11            b = a;
12            while (j < 3) {
13                b = b + 1;
14                j = j + 1;
15            }
16            a = a + 1;
17            i = i + 1;
18        }
19        return b;
20     }
21  }
```

Test Case	Input	Output	E	original Line	modified Line
test_test1_f1	none	b = 7	7	a = 0;	a = 1;
test_test1_f2	none	b = 7	9	while (i < 5)	while (i ¡ 4)
test_test1_f3	none	b = 7	13	b = b + 1;	b = a + 1;

Table 5.23.: Test Cases for WhileLoops2.java

File	Method to test	Time	Statements	relevant Slicer	Result	faulty Line
WhileLoops2.java	test2_f1	781 ms	14	10	6	Y
WhileLoops2.java	test2_f2	771 ms	14	10	6	Y
WhileLoops2.java	test2_f3	708 ms	14	9	5	Y

Table 5.24.: Test Results for WhileLoops2.java

5.4. Summary

To have a better overview of the test results all of them are listed again in Table 5.25. All if- and while-statements are automatically added to the possibly faulty statements because it is not yet possible to check them for consistency. Despite of this the Debugger is able to further limit the possibly faulty statements in contrast to the sole relevant slicing approach in almost all test cases.

The bad news is the exclusion of the faulty statement in some test cases. The reason for this is described in Chapter 6.1

File	Method to test	Time	Statements	relevant Slicer	Result	faulty Line
PowerFunction.java	pow3	505 ms	12	9	6	Y
PowerFunction.java	pow4	525 ms	12	9	5	Y
AKSWT1.java	Example	280 ms	9	6	4	Y
AKSWT1.java	Example1	302 ms	9	6	4	N
AKSWT1.java	Example2	297 ms	9	6	4	Y
AKSWT1.java	Example3	82 ms	9	2	2	Y
AKSWT1.java	Example4	297 ms	9	6	4	Y
ProdSum.java	Example_f1	380 ms	11	6	4	Y
ProdSum.java	Example_f2	244 ms	11	4	3	Y
Multiplication.java	multiply_f1	481 ms	14	10	7	Y
Multiplication.java	multiply_f2	335 ms	14	8	5	Y
Multiplication.java	multiply_f3	350 ms	14	8	6	Y
Divide.java	divide_f1	533 ms	13	10	8	Y
Divide.java	divide_f2	739 ms	13	12	9	Y
Divide.java	divide_f3	359 ms	13	8	6	Y
GCD.java	gcd_f1	– ms	5	4	3	Y
GCD.java	gcd_f2	– ms	5	4	2	Y
GCD.java	gcd_f3	– ms	5	3	2	N
BinSearch.java	binSearch	– ms	12	10	9	Y
IfExamples.java	test1	135 ms	5	3	3	Y
IfExamples.java	test2	72 ms	5	2	2	Y
IfExamples.java	test3	73 ms	9	3	3	Y
SumPowers.java	sumpowers_f1	726 ms	18	13	7	Y
SumPowers.java	sumpowers_f2	276 ms	18	6	3	Y
SumPowers.java	sumpowers_f3	273 ms	18	6	3	N
SumPowers.java	sumpowers_f4	593 ms	18	12	6	Y
SumPowers.java	sumpowers_f5	703 ms	18	13	8	Y
TrafficLight.java	method1_f1	536 ms	12	11	5	Y
TrafficLight.java	method1_f2	310 ms	12	6	3	Y
WhileLoops.java	test1_f1	478 ms	12	7	5	Y
WhileLoops.java	test1_f3	496 ms	12	7	5	Y
WhileLoops2.java	test2_f1	781 ms	14	10	6	Y
WhileLoops2.java	test2_f2	771 ms	14	10	6	Y
WhileLoops2.java	test2_f3	708 ms	14	9	5	Y

Table 5.25.: This table represents all of the testing results at once.
The reason for the missing time measurement of the examples GCD.java and BinSearch.java is the following: The consistency check for these two examples have to be done by hand because the ”div” and ”modulo” constraint in Minion cannot handle variables with a negative domain and therefore the needed time for debugging cannot be reasonably measured.

Chapter 6

Open Problems

A short description of the open problems and a solution approach for them is given in this chapter. For further development and improvement of the Debugger these two points shall be handled first.

6.1. Φ-Function

As it can be seen in Table 5.25, some of the errors are not found by the Debugger. In more detail, this means that the Debugger marks the wrong line as correct. Listing 6.1 shows the Minion code of the CSP generated with Listing 5.6 and test case "testGCD_f3" at Table 5.11.

Listing 6.1: CSP of GCD.java and Test Case testGCD_f3

```
1  MINION 3
2  **VARIABLES**
3  BOOL ab[3]
4  DISCRETE b_0 {1..1000}
5  DISCRETE b_1 {-10..1000}
6  DISCRETE a_0 {1..1000}
7  DISCRETE a_1 {-10..1000}
8  DISCRETE c_0 {-10..1000}
9  DISCRETE created_tmp_var_1 {1..1000}
10 **CONSTRAINTS**
11 watched-or({element(ab,0,1),modulo(a_0,b_0,created_tmp_var_1)})
12 watched-or({element(ab,0,1),eq(c_0,created_tmp_var_1)})
13 watched-or({element(ab,1,1),eq(a_1,b_0)})
14 watched-or({element(ab,2,1),eq(b_1,0)})
15 # TESTCASE
16 eq(ab[2],0)
17 eq(ab[0],0)
18 eq(ab[1],1)
19 eq(b_0,9)
20 eq(a_0,14)
21 eq(a_1,1)
22 **EOF**
```

The important lines are 13, 19 and 21. With these three lines the CSP is only solvable if the abnormal variable of line 13 is set. In all other cases the CSP is not solvable because a_1 cannot be 1 (line21) and

9 (line 13 - the value of b_0 is 9) at the same time. Therefore all other CSPs will be discarded and the error according to the Debugger lies in the assignment of b_0 to a_1. But that is wrong, the error lies in the statement in line 14. This problem is caused by the elimination of the if- and while-statements. If b is not set to 0 in line 8 of the original program in Listing 5.6, the loop will continue and reassign the variable a. This reassignment may cause the program to be correct. In order not to remove such connections, the if- and while-statements have also to be converted to Minion and set into relation with the variables.

Convert If- and While-Statement

The conversion of the if- and while-statements is the easier part. These two statements can be treated in the same way, because a while-loop is nothing else than an if-statement repeated multiple times. Table 6.1 shows the conversion of different logical operators.

Java	Minion
x == y	eq(x,y)
x != y	diseq(x,y)
x < y	ineq(x,y,-1)
x <= y	ineq(x,y,0)
x > y	ineq(y,x,-1)
x >= y	ineq(y,x,0)

Table 6.1.: Conversion of a Java logical Operation to a Minion Constraint

Afterwards the Minion constraint has to be combined with another constraint, namely the "reify" constraint. In the manual of Minion, which can be found at [3], this constraint is described as follow:

reify(constraint, r) where r is a 0/1 var
ensures that r is set to 1 if and only if constraint is satisfied. That is, if r is 0 the constraint must NOT be satisfied; and if r is 1 it must be satisfied as normal. Conversely, if the constraint is satisfied then r must be 1, and if not then r must be 0.

This means for the conversion that the boolean variable r stores the evaluation of the conditional statement. To distinguish the different conditional evaluations every statement has to have its own boolean variable. This can be guaranteed by an unique index.

Φ-Function

To get to a relationship between the variables and the conditional statement a Φ-Function is introduced.

$$\Phi(\text{v_true, v_else, cond_i}) = \begin{cases} v_true, & \text{if } cond_i \text{ evaluates to true,} \\ v_else, & \text{otherwise.} \end{cases}$$

This function has to be taken into account when calculating the exit variables of conditional statements. In the following a short example is given to demonstrate the Φ-Function and the concept of exit variables.

Assume the small program in Listing 6.2 and its SSA representation in Listing 6.3. The Φ-Function for the exit variable a_3 is Φ(a_1, a_2, cond_0) - if the boolean variable cond_0 is true, a_3 is equal to a_1 else it is equal to a_2. In the case of having just an if-then statement without an else path the v_else of the Φ-Function is the last definition of the corresponding variable outside the conditional statement. Taking the example of Listing 6.2 and deleting the else path in line 4 and 5, the last definition outside the conditional statement is in line 1. In this case the Φ-Function will be Φ(a_1, a_0, cond_0).

This procedure has to be carried out for every variable defined inside a conditional.

Listing 6.2: Small Program for demonstrating the Φ-Function

```
1 int a = 0;
2 if(b == 0)
3   a = 10;
4 else
5   a = 100;
```

Listing 6.3: SSA Representation of the small Program

```
1 a_0 = 0;
2 reify(eq(b,0), cond_0)
3 a_1 = 10;
4 a_2 = 100;
5 a_3 = ...
```

Now the problem is to implement this Φ-Function. Therefore a knowledge of all the defined variables in conditional and loop statements is necessary. These variables are the same as the potential relevant variables the relevant slicer needs. When having this variables, which is not the case yet, it will be easy to implement this function. As a consequence of that, it will be possible to keep the relation between the conditional statement and the variables and it will also be possible to check the conditional statements for consistency. This will probably further limit the possibly faulty statements.

6.2. Relevant Slicer

As already mentioned in Chapter 4.1.3 the algorithm for calculating the relevant slice of a variable is implemented in a different way than described in this thesis. This implementation works for the first prototype because of the use of simpler programs. For further improvement and testing results the demand for a good relevant slicer algorithm is crucial. A good implementation of the algorithm described in Algorithm 1 should solve this problem.

Chapter 7

Conclusion

7.1. Conclusion

I have presented a Debugger for helping programmers and testers to locate the cause of a failing test case in this thesis. I have done this with the help of four basic parts, namely a debugging problem, a relevant slicing algorithm, a minimal hitting-set algorithm and constraint solving.

The produced results are an average reduction of approximately 25% in contrast to just using relevant slicing. Since it is just possible to test smaller and simple programs no conclusion about the results of the Debugger can be reached when dealing with complex software.

Since this was a prototype and also a proof of concepts it is obvious that the Debugger cannot produce the above mentioned results in software programs with thousands of lines of code yet. Improvements and further development have to be done to get the Debugger to this level. The next section lists some of those.

7.2. Future Work

This list is just a reflection about future tasks which can be done to improve the Debugger. The crucial points of this list have already been described in more detail in Chapter 6.

- **Φ-Function**
  A Φ-Function is needed to check the conditional statements for consistency and thereby further limit the possibly faulty statements and to eliminate negative side effects as described in Chapter 6.1.
- **Test Cases**
  The supported test cases are now limited to JUnit versions <= 3.8. A first improvement is to support all JUnit versions. The next step then may be to be independent from JUnit and be able to use other testing tools, too. Another possibility may be to create an own testing framework.
- **Relevant Slicing Algorithm**
  The implemented relevant slicing algorithm used now by the Debugger works fine for the first prototype and smaller examples. This was enough to make a proof of concept and get first results. But for usage in larger programs and producing further results the algorithm has to be adapted to the presented algorithm in this thesis.
- **Expansion to other programming Languages**
  A last point may be the expansion to other programming languages. Now the Debugger works only for Java programs. A future goal is to add other languages, like the C family or other currently often

used ones. Widening the scope of programming languages is a necessary step for increasing the usage of the Debugger.

Appendix A

Manual

This chapter gives an overview of the GUI of the Debugger and of how to work with it. It is highly recommended to read first Chapter 4.2 to be informed about the limitations and to be able to produce good results with the Debugger.

As it can be seen in Figure A.1 the Debugger needs different types of input and produces different types of output. The fields and buttons are described in the following.

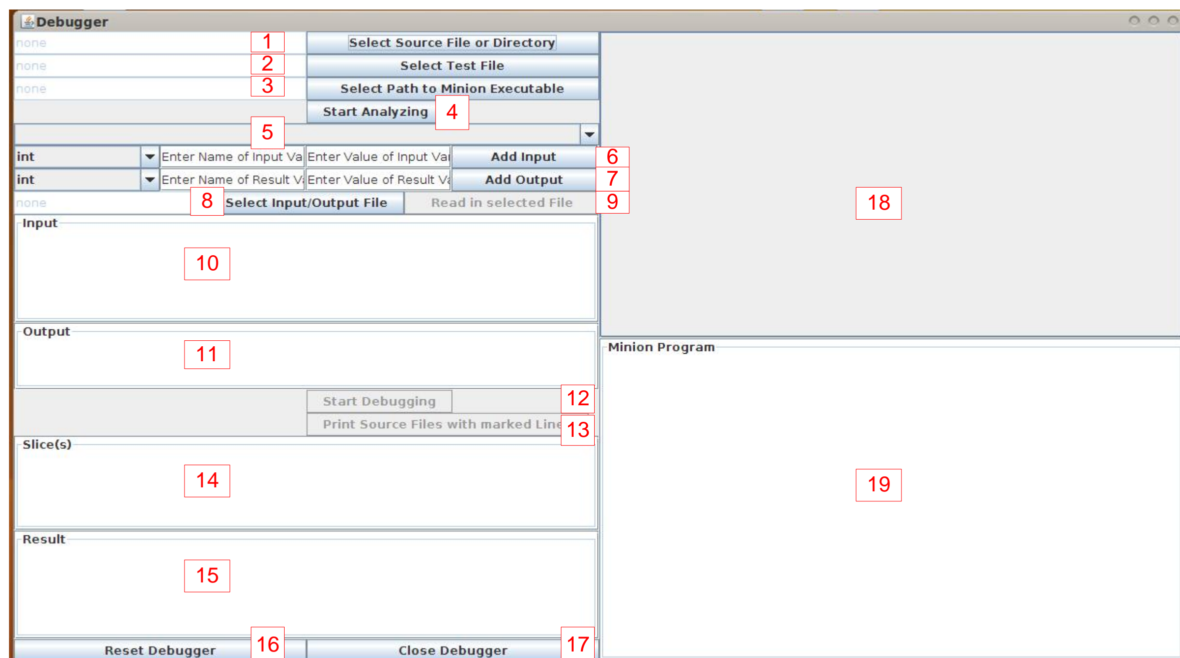

Figure A.1.: The Debugger GUI

1. Select Source File or Directory
   After pressing the button a file dialog will be opened and there is the possibility to choose between selecting a single Java file or a whole directory. When selecting a whole directory it will also be searched for Java files in all the subdirectories.

2. Select Test File
   A file dialog is opened again and a Java test file should be chosen. This test file should contain the test case to be executed.

3. Select Path to Minion Executable
   The last parameter to choose in the analyzing phase is the path to the Minion executable. When downloading Minion from [3] the binary can be found in the /bin directory.

4. Start Analyzing
   Pressing this button starts the analyzing phase of the Debugger. Thereby the class directory and jars, which are contained in the chosen directory, are added to the classpath. Afterwards the source files of the program will be analyzed and modified with the help of automatically generated lexer and parser - generation has been done via ANTLR 3.2. At the end the modified files will be recompiled to generate the class files in the class directory.

5. Select Test Method
   After analyzing the source and test files this drop-down list contains all the names of the test cases contained in the test file. The one which shall be executed has to be chosen.

6. Add Input
   It is very difficult to read out the input and output variables and its values from a JUnit test case. Therefore the decision was to enter the input and output variables manually. The way of how to enter these variables is quite easy. The drop-down list contains the four supported data types, namely ”int”, ”boolean”, ”int[]” and ”boolean[]”. The first text field is for the name of the variable. This name has to be equal to the name in the Java program. The second text field is for the value of the variable. To enter an array simply type in the values separated by ”,”. To store the variable press ”Add Input”.
   For example, entering a boolean[] a = [true, false, true, false] will consist of the following four steps.

   a) Choose boolean[] from the drop-down list.
   b) Enter the name ”a” in the first text field.
   c) Enter the corresponding value ”true,false,true,false” in the second text field.
   d) Press ”Add Input”.

7. Add Output
   The output variables work in the same way as the input variables.

8. Select Input/Output File
   The second possibility to enter input and output variables is with the help of a .txt file. This file should have the following syntax:
   <Path/to/the/TestFile>:<NameOfTheTestCase>
   INPUT: type<SPACE>nameOfTheVariable<SPACE>=<SPACE>valueOfTheVariable;
   OUTPUT: type<SPACE>nameOfTheVariable<SPACE>=<SPACE>valueOfTheVariable;
   Note the three spaces and the ”;” at the end of a variable declaration. In case of having more than one input or output variable add it to the corresponding line. When the program does not need input variables skip the input line and start with the output line. A single .txt file can of course contain more than one test case.

   The example in Listing A.1 shows a correct entry in the file:

Listing A.1: Entry in the Input/Output File

```
1  path/to/PowTest.java:testPow
2  INPUT: int a = 2; int b = 2;
3  OUTPUT: int result = 4;
4  path/to/PowTest.java:testPow1
5  INPUT: int a = 2; int b = 0;
6  OUTPUT: int result = 1;
7  ...
```

9. Read in selected File
After choosing the path to the file with the input and output variables press this button to read in the variables necessary for the test case. The read in variables are listed in the input or output field.

10. Input
All entered input variables are listed here.

11. Output
All entered output variables are listed here.

12. Start Debugging
After pressing this button the Debugger starts its work. The debugging approach starts by running the test case. If it is a failing test case, it will continue with converting the execution trace to Minion. Then the slice for the output variables is calculated and finally Minion checks the diagnoses for consistency.

13. Print Source File(s) with marked Lines
Prints out the source code of the program. The possibly faulty statements are marked red and bold.

14. Slice(s)
This field contains the slices of all output variables.

15. Result
This field contains the valid diagnoses for the program. In other words these are the possibly faulty statements of the the Java program.

16. Reset Debugger
Resets the Debugger to the initial state. After pressing this button you have to start again by entering the source file or directory.

17. Close Debugger
Closes the Debugger.

18. Source File(s) with marked Lines
Shows the source code of the Java classes in which possibly faulty statements are marked red and bold.

19. Minion Program
Shows one of the Minion programs. This program is equal to one CSP. As mentioned before, the Minion files differ just in the assignment of the ”ab”-variable. Therefore the decision was to show just one of them in this field.

Bibliography

[1] Jade project homepage. http://www.dbai.tuwien.ac.at/proj/Jade/. 12.10.2011. (Cited on page 45.)

[2] Junit. http://www.junit.org/. 19.09.2011. (Cited on pages 1 and 23.)

[3] Minion. http://minion.sourceforge.net/index.html. 19.09.2011. (Cited on pages 15, 30, 54, and 60.)

[4] Wikipedia - list of software bugs. http://en.wikipedia.org/wiki/List_of_software_bugs. 07.10.2011. (Cited on page 1.)

[5] DECHTER, R. 2003. *Constraint Processing*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. (Cited on page 15.)

[6] DETASSIS, D. 2011. The Calculation of Minimal Hitting Sets with different Algorithms. Master-Project at the Institute for Software Technology at the Graz University of Technology. (Cited on page 26.)

[7] GENT, I. P., JEFFERSON, C., AND MIGUEL, I. 2006. Minion: A fast, scalable, constraint solver. In *Proceeding of the 2006 conference on ECAI 2006: 17th European Conference on Artificial Intelligence August 29 – September 1, 2006, Riva del Garda, Italy*. IOS Press, Amsterdam, The Netherlands, The Netherlands, 98–102. (Cited on page 15.)

[8] GYIMÓTHY, T., BESZÉDES, A., AND FORGÁCS, I. 1999. An efficient relevant slicing method for debugging. In *Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*. ESEC/FSE-7. Springer-Verlag, London, UK, 303–321. (Cited on page 5.)

[9] KOREL, B. AND LASKI, J. 1988. Dynamic program slicing. *Inf. Process. Lett. 29*, 155–163. (Cited on page 5.)

[10] LIN, L. AND JIANG, Y. 2003. The Computation of Hitting Sets: Review and new Algorithms. *Information Processing Letters 86,* 4 (May), 177–184. (Cited on page 15.)

[11] REITER, R. 1987. A theory of diagnosis from first principles. *Artif. Intell. 32*, 57–95. (Cited on page 14.)

[12] RIEDL, M. 2011. Spectrum Based Diagnosis. M.S. thesis, Graz University of Technology. (Cited on pages 25, 26, 35, 36, and 43.)

[13] Rui Abreu and Arjan J. C. van Gemund 2009. *A Low-Cost Approximate Minimal Hitting Set Algorithm and its Application to Model-Based Diagnosis*. Rui Abreu and Arjan J. C. van Gemund, Symposium on Abstraction, Reformulation and Approximation (SARA). (Cited on pages 15, 26, and 36.)

[14] WEISER, M. 1982. Programmers use slices when debugging. *Commun. ACM 25*, 446–452. (Cited on page 5.)

[15] WEISER, M. 1984. Program slicing. *Software Engineering, IEEE Transactions on SE-10,* 4 (july), 352 –357. (Cited on page 5.)

[16] WOTAWA, F. 2001. A Variant of Reiter's Hitting-Set Algorithm. *Information Processing Letters 79,* 1 (May), 45–51. (Cited on page 15.)

[17] WOTAWA, F. 2011. On the use of constraints in dynamic slicing for program debugging. *Software Testing Verification and Validation Workshop, IEEE International Conference 0*, 624–633. (Cited on pages 1, 3, 15, 30, and 41.)

[18] ZHANG, X., HE, H., GUPTA, N., AND GUPTA, R. 2005. Experimental evaluation of using dynamic slices for fault location. In *AADEBUG'05: Proceedings of the sixth international symposium on Automated analysis-driven debugging*. ACM, New York, NY, USA, 33–42. (Cited on page 5.)