Masterarbeit

Spectrum-based Debugging by means of Java Bytecode Manipulation

Sabine ZARL, BSc

Institut für Softwaretechnologie
Technische Universität Graz
Vorstand: Univ.-Prof. Dipl.-Ing. Dr. techn. Wolfgang Slany

Betreuer/Begutachter: Univ.-Prof. Dipl.-Ing. Dr. techn. Franz Wotawa

Graz, im Dezember 2010

Kurzfassung

Dieses Projekt kombiniert Spektrum-basiertes Debugging und Java Bytecode Manipulation. Spektrum-basiertes Debugging ist eine statistische Diagnosetechnik um einen Fehler in Computerprogrammen zu lokalisieren. Java Bytecode Manipulation wird dazu verwendet den Ausführungsablauf des zu testenden Programms zu extrahieren. Es wurde ein Framework implementiert um das Projekt zu realisieren. Es manipuliert `class` Dateien und führt die gegebenen Testfälle aus um Laufzeit-Informationen zu sammeln. Diese Informationen, gemeinsam mit den Ergebnissen der Testfälle werden dazu verwendet, die Befehle des zu testenden Programms anhand ihrer Fehlermöglichkeit zu reihen. Das Framework unterstützt verschiedene Ähnlichkeitskoeffizienten um die Rangliste zu berechnen. Diese Arbeit stellt eine empirische Evaluation des Frameworks bereit.

Abstract

This project combines spectrum-based debugging and Java bytecode manipulation. Spectrum-based debugging is a statistical diagnosis technique to locate a fault in computer programs. Java bytecode manipulation is used to extract the execution trace of the program under test. A framework was implemented to realize the project. It manipulates `class` files and executes the given test cases to collect runtime information. This information in collaboration with the results of each test case is used to rank the statements of the program under test according to their possibility of being faulty. The framework supports different similarity coefficients to calculate the ranking. This paper provides an empirical evaluation of the framework.

Keywords: spectrum-based debugging, spectrum-based diagnosis, fault localization, java bytecode manipulation

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.

....
date (signature)

Acknowledgment

My family and my boyfriend deserve special gratitude. They supported me during the whole studies. Without my parents the university studies have not been possible for me.

I thank the members of the Institute for Software Technology of the Graz University of Technology, especially my advisor and assessor for the master's thesis Univ.-Prof. Dipl.-Ing. Dr. techn. Franz Wotawa.

December 2010 Sabine Zarl

Contents

1. Introduction

Since the first computer system has been invented, the computer is getting better and faster. Therewith, computer programs are getting greater, more complex and thus more fault-prone. This is the reason, why fault localization techniques are more important than ever today. The most promising fault localization approach is spectrum-based debugging. It is simple and moreover effective. Contrary to the model-based approach, it is a statistical, single-fault diagnosis technique, which uses similarity coefficients to rank all program entities according to their possibility of being faulty.

This project combines spectrum-based debugging and Java bytecode manipulation. Java bytecode is manipulated to get the information of the program used by the similarity coefficient at runtime to calculate the ranks. The most obvious advantage of working on Java bytecode is that the source code is not needed. Despite of that, the mapping of bytecode instruction and source line number is available. This information is necessary to realize debugging, or in other words diagnosis. More important for this project is that the source code needs not to be parsed anymore. The Java bytecode can easily be manipulated with the help of bytecode manipulation tools.

The practical part of the project was to realize spectrum-based debugging with Java bytecode manipulation. Therefore, a framework was implemented. It manipulates `class` files and executes the given test cases to collect runtime information. The execution trace and the results of each test case (failed or not) are used to rank the statements of the program under test according to their possibility of being faulty. The framework supports different similarity coefficients to calculate the ranking. Finally, an empirical evaluation of the framework, including the coefficients, was carried out.

1.1. Structure of the Document

The paper is indexed as follows. Section 2 refers to some related work and gives the background information. It describes spectrum-based debugging and Java bytecode. In Section 3 the analysis and the design of the software are provided. Section 4 explains how the software was implemented. The empirical evaluation is documented in Section 5. Section 6 concludes the project and points to some possible future work.

2. Related Work and Background Information

This chapter deals with the existing work on spectrum-based debugging and Java bytecode analysis. In Section 2.1 a few definitions are given. Section 2.2 introduces spectrum-based debugging and describes what program spectra are. Section 2.3 explains what Java bytecode is and why it is needed. In Section 2.4 other projects using Java bytecode analysis are referenced. Finally, Section 2.5 describes three different Java bytecode manipulation tools.

2.1. Failure, Error and Fault

These items are defined in [7] as follows:

- Failure: event that occurs when the delivered service differs from the correct one
- Error: deviation of a system state from the correct one
- Fault (Bug): cause of an error

Summarized in one sentence it could be said that a fault causes an error, which leads to a failure. However, it should be noticed that not every error effects a failure.

In other words, a fault is the incorrect source code, the error is the wrong behavior of the program and the failure is the exception which is thrown if the error is identified.

In the following, an example is provided to illustrate these three different definitions.

```
1  public int power(int a)
2  {
3    int res = (a+a); // FAULT! correct: (a*a)
4    return res;
5  }
```

The error is that the program does not calculate the power of the given variable `a`, but the sum. When the program is executed with test inputs, the expected output and the correct output are compared. If they differ, a failure event is output. This example also shows that an error is not always identified. If variable `a` is set to value 2, no failure occurs. However, if variable `a` is set to value 3, the error is detected.

2.2. Spectrum-based Debugging

Debugging is diagnosis applied to computer programs. Diagnosis, in general, is fault localization. [2]

Spectrum-based debugging is a lightweight, statistical, single-fault approach. It uses a program spectrum, an error vector and a similarity coefficient to locate a fault in a computer program. [5, 4, 2, 3, 29, 1] This fault localization approach is simple and effective and thus one of the most promising [30].

2.2.1. Program Spectra

A program spectrum, in general, is a collection of data, providing information about dynamic behavior [5, 4]. In [17] a listing of different types of program spectra is given.

This project uses a hit spectrum and stores in a matrix X whether a program entity, in this case a statement, was executed during a test run or not. Therefore, one dimension of the matrix represents the test cases and the other one the statements. The value 1 indicates that the statement was invoked, when the particular test case was executed. The value 0 represents that it was not invoked. Besides, an error vector $\vec{e}$ is needed to store whether a test case failed (1) or not (0). [5, 4, 2] Figure 2.1 illustrates such a program spectrum.

	test suite				
statements	tc_1	...	tc_j	...	tc_k
s_1	x_{11}	...	x_{1j}	...	x_{1k}
...	...	...	...	...	...
s_i	x_{i1}	...	x_{ij}	...	x_{ik}
...	...	...	...	...	...
s_n	x_{n1}	...	x_{nj}	...	x_{nk}
error vector	e_1	...	e_j	...	e_k

Figure 2.1.: A program spectrum with an error vector, where n is the number of statements and k is the number of test cases. $x_{ij} = \{0,1\}, e_j = \{0,1\}; i = 1 \ldots n, j = 1 \ldots k$

2.2.2. Fault Localization

To locate the fault, the statements have to be ranked in terms of their suspiciousness [18]. In conjunction with spectrum-based debugging, locating a fault means to identify the statement whose run vector is most similar to the error vector. Therefore, a similarity coefficient is used, which rank the program entities regarding their possibility of being faulty. [5]

In [2] four counters to express such similarity coefficients are introduced $i = 1 \dots n, j = 1 \dots k$:

1. a_{11}: $x_{ij} = 1$, $e_j = 1$
2. a_{10}: $x_{ij} = 1$, $e_j = 0$
3. a_{01}: $x_{ij} = 0$, $e_j = 1$
4. a_{00}: $x_{ij} = 0$, $e_j = 0$

For example, a_{11} is the number of runs where statement `i` was executed and where an error has been detected. This means, x_{ij} indicates whether a statement `i` was executed in run `j` (1) or not (0). This information is taken from the matrix X. The error vector $\vec{e}$ declares whether run `j` was faulty $e_j = 1$ or not $e_j = 0$. See also Figure 2.1.

2.2.3. Similarity Coefficients

In the following, the most popular similarity coefficients are listed. Comparing these coefficients, [2, 5] and [4] come to the conclusion that the Ochiai coefficient gives the best results.

Ochiai Coefficient:
The Ochiai coefficient is taken from the molecular biology domain [2]. In [27] it is used for genetic cluster analysis.

$$s_i = \frac{a_{11}}{\sqrt{(a_{11}+a_{01})*(a_{11}+a_{10})}}$$

Jaccard Coefficient:
The major application area of this coefficient is the field of data clustering [2]. It is used in the Pinpoint framework [11]. The Pinpoint framework detects system problems and isolates their root causes. It can be applied to almost any J2EE application.

$$s_i = \frac{a_{11}}{a_{11}+a_{01}+a_{10}}$$

Tarantula Coefficient:
The Tarantula system [19, 18] is a fault analysis and visualization tool using two different coefficients. The first one is used for computing the hue of a statement and the other one is used for calculating the suspiciousness. The relevant coefficient for this work is the one calculating the suspiciousness.

$$s_i = \frac{\frac{a_{11}}{a_{11}+a_{01}}}{\frac{a_{11}}{a_{11}+a_{01}}+\frac{a_{10}}{a_{10}+a_{00}}}$$

AMPLE Coefficient:
This coeficient is used by the AMPLE (Analyzing Method Patterns to Locate Errors) tool [2]. The idea of [15] is that some failures appear only through a sequence of method calls. That's why their tool collects sequences of method calls on a per-object basis when instrumenting a given Java program. Thus, a sequence set for each class is created. The aim of the tool is then to rank the classes according to their possibility of being defective.

$$s_i = \left| \frac{a_{11}}{a_{01}+a_{11}} - \frac{a_{10}}{a_{00}+a_{10}} \right|$$

2.2.4. Example

This section provides an example, which should help to get the idea of spectrum-based debugging.

Below a short multiplication method, which takes two integers and calculates their product with performing addition, is written down.

```
1  public int mult(int a, int b)
2  {
3    if (a == 0 || b == 0)
4      return 0;
5
6    int res = a;
7
8    for (int i = 0; i < (b-1); i++)
9      res += b; //correct: res += a;
10
11   return res;
12 }
```

The multiplication method is executed with 8 different input data. The result is compared to the expected output. Besides, the execution trace needs to be stored. However, the information of how often a statement was executed is not necessary. Table 2.1 shows the test cases, the expected output and the execution trace as well as the real output of each test case.

The resulting program spectrum is illustrated in Table 2.2. The visited statements for each test case are marked with the value 1 in the matrix. If the expected output and the result of the multiplication method do not match, the error is marked in the error vector with the value 1.

input	expected output	execution trace	output of mult()
mult(2,3)	6	1, 3, 6, 8, 9, 11	res = 8
mult(0,5)	0	1, 3, 4	res = 0
mult(1,1)	1	1, 3, 6, 8, 11	res = 1
mult(2,1)	2	1, 3, 6, 8, 11	res = 2
mult(1,2)	2	1, 3, 5, 8, 9, 11	res = 3
mult(2,0)	0	1, 3, 4	res = 0
mult(4,3)	12	1, 3, 6, 8, 9, 11	res = 10
mult(5,2)	10	1, 3, 6, 8, 9, 11	res = 7

Table 2.1.: The test suite with the expected output, the execution trace and the real output of the multiplication method.

statements	tc_1	tc_2	tc_3	tc_4	tc_5	tc_6	tc_7	tc_8
1	1	1	1	1	1	1	1	1
2	0	0	0	0	0	0	0	0
3	1	1	1	1	1	1	1	1
4	0	1	0	0	0	1	0	0
5	0	0	0	0	0	0	0	0
6	1	0	1	1	1	0	1	1
7	0	0	0	0	0	0	0	0
8	1	0	1	1	1	0	1	1
9	1	0	0	0	1	0	1	1
10	0	0	0	0	0	0	0	0
11	1	0	1	1	1	0	1	1
12	0	0	0	0	0	0	0	0
error vector	1	0	0	0	1	0	1	1

Table 2.2.: The program spectrum and the error vector of the multiplication example.

Statement	Counters
1, 3	$a_{00} = 0$ $a_{01} = 0$ $a_{10} = 4$ $a_{11} = 4$
2, 5, 7, 10, 12	$a_{00} = 4$ $a_{01} = 4$ $a_{10} = 0$ $a_{11} = 0$
4	$a_{00} = 2$ $a_{01} = 4$ $a_{10} = 2$ $a_{11} = 0$
6, 8, 11	$a_{00} = 2$ $a_{01} = 0$ $a_{10} = 2$ $a_{11} = 4$
9	$a_{00} = 4$ $a_{01} = 0$ $a_{10} = 0$ $a_{11} = 4$

Table 2.3.: The counters for each statement.

State-ment	Jaccard $s_i = \frac{a_{11}}{a_{11}+a_{01}+a_{10}}$	possibility Ochiai $s_i = \frac{a_{11}}{\sqrt{(a_{11}+a_{01})*(a_{11}+a_{10})}}$	Tarantula $s_i = \frac{\frac{a_{11}}{a_{11}+a_{01}}}{\frac{a_{11}}{a_{11}+a_{01}}+\frac{a_{10}}{a_{10}+a_{00}}}$
1	$\frac{4}{4+0+4} = 0.5$	$\frac{4}{\sqrt{(4+0)*(4+4)}} = 0.7071$	$\frac{\frac{4}{4+0}}{\frac{4}{4+0}+\frac{4}{4+0}} = 0.5$
2	$\frac{0}{0+4+0} = 0.0$	$\frac{0}{\sqrt{(0+4)*(0+0)}} = 0.0$	$\frac{\frac{0}{0+4}}{\frac{0}{0+4}+\frac{0}{0+4}} = 0.0$
3	$\frac{4}{4+0+4} = 0.5$	$\frac{4}{\sqrt{(4+0)*(4+4)}} = 0.7071$	$\frac{\frac{4}{4+0}}{\frac{4}{4+0}+\frac{4}{4+0}} = 0.5$
4	$\frac{0}{0+4+2} = 0.0$	$\frac{0}{\sqrt{(0+4)*(0+2)}} = 0.0$	$\frac{\frac{0}{0+4}}{\frac{0}{0+4}+\frac{2}{2+2}} = 0.0$
5	$\frac{0}{0+4+0} = 0.0$	$\frac{0}{\sqrt{(0+4)*(0+0)}} = 0.0$	$\frac{\frac{0}{0+4}}{\frac{0}{0+4}+\frac{0}{0+4}} = 0.0$
6	$\frac{4}{4+0+2} = 0.6667$	$\frac{4}{\sqrt{(4+0)*(4+2)}} = 0.8165$	$\frac{\frac{4}{4+0}}{\frac{4}{4+0}+\frac{2}{2+2}} = 0.6667$
7	$\frac{0}{0+4+0} = 0.0$	$\frac{0}{\sqrt{(0+4)*(0+0)}} = 0.0$	$\frac{\frac{0}{0+4}}{\frac{0}{0+4}+\frac{0}{0+4}} = 0.0$
8	$\frac{4}{4+0+2} = 0.6667$	$\frac{4}{\sqrt{(4+0)*(4+2)}} = 0.8165$	$\frac{\frac{4}{4+0}}{\frac{4}{4+0}+\frac{2}{2+2}} = 0.6667$
9	$\frac{4}{4+0+0} = 1.0$	$\frac{4}{\sqrt{(4+0)*(4+0)}} = 1.0$	$\frac{\frac{4}{4+0}}{\frac{4}{4+0}+\frac{0}{0+4}} = 1.0$
10	$\frac{0}{0+4+0} = 0.0$	$\frac{0}{\sqrt{(0+4)*(0+0)}} = 0.0$	$\frac{\frac{0}{0+4}}{\frac{0}{0+4}+\frac{0}{0+4}} = 0.0$
11	$\frac{4}{4+0+2} = 0.6667$	$\frac{4}{\sqrt{(4+0)*(4+2)}} = 0.8165$	$\frac{\frac{4}{4+0}}{\frac{4}{4+0}+\frac{2}{2+2}} = 0.6667$
12	$\frac{0}{0+4+0} = 0.0$	$\frac{0}{\sqrt{(0+4)*(0+0)}} = 0.0$	$\frac{\frac{0}{0+4}}{\frac{0}{0+4}+\frac{0}{0+4}} = 0.0$

Table 2.4.: The ranks for all source lines calculated by three different similarity coefficients.

The next step is to calculate the 4 counters for each statement. Table 2.3 shows the counters for each statement.

The counter value of statement 9 attracts attention. The statement is visited every time a failure occurred and never visited when no failure occurred. It can also be seen that statements 1 and 3 are always visited and that statements 2, 5, 7, 10 and 12 are never visited.

The results of the similarity coefficients are listed in Table 2.4. The possibilities of being faulty are calculated for each source line.

Every similarity coefficient diagnoses that statement 9 has the highest similarity to the error vector. In this easy example, there is even a similarity of 100 %.

2.2.5. Other Application Areas

Spectrum-based fault localization is, as described above, a statistical, single-fault approach. In [3] a spectrum-based multiple-fault approach is presented. Therefore, spectrum-based fault localization (SFL) is combined with model-based diagnosis (MBD). MBD considers multiple faults, but it is more complex than SFL and cannot be applied to large programs. They try to unify the best of both to enable multiple fault localization on large, real-world programs.

The project of [1] also combines the spectrum-based and the model-based approach. They use the model-based approach to refine the spectrum-based ranking. The program's semantics is examined to filter out the components that do not explain the observed failures.

Spectrum-based fault localization can also be used for test case grouping, like [29] describes. The test cases are generated with the model-based approach. These test cases are grouped by means of spectrum-based fault localization applied to the specification. Consequently, test cases are in the same group, which most likely detect the same failure. This grouping enables that not all failed test cases need to be analyzed, but only one per group. Thus, the post analysis time is reduced.

The technical report [30] deals with spectrum-based fault localization without test oracles. This means, it deals with programs of which the correctness of the output is unable or too expensive to be verified. To minimize this oracle problem metamorphic slices are used. Such metamorphic slices are related to a property of the algorithm being implemented, a metamorphic relation. With multiple inputs and the output of the algorithm it can be verified if these relations are satisfied. The violation or non-violation of a metamorphic relation can then be treated as alternative to the test result `failed` or `passed` in spectrum-based fault localization.

2.3. Java Virtual Machine

The Java Virtual Machine (JVM) executes a Java program. The specification of the JVM can be found at [21]. The Java Platform was designed to meet some network requirements [23]:

1. compiled code had to survive transport across networks
2. compiled code had to operate on any client
3. assure the client that the program was safe to run

The Java Virtual Machine [23] is the most important component of a Java Platform. It affords hardware- and operating system-independency, the small size of compiled code and protects users from malicious programs. It is an abstract computing machine, which has its own instruction set like a real computing machine.

A Java Virtual Machine knows a particular binary format, the `class` file format. Such `class` files contain Java Virtual Machine **Bytecode instructions** and some

additional information. One `class` file defines one class or interface. [25] To ensure security, there are strong format and structural constraints on the `class` file code [23].

A `class` file consists of the following elements [24]:

magic: number to identify the class file format with the value: `0xCAFEBABE`

minor_version: minor version number of the class file

major_version: major version number of the class file

constant_pool_count: the number of entries in the constant_pool table plus one

constant_pool: a table of constants, indexed from 1 to constant_pool_count-1

access_flags: used to denote access permissions to and properties of this class or interface (public, final, super, interface, abstract)

this_class: the value represents the index of a CONSTANT_Class_info in the constant_pool; this CONSTANT_Class_info structure represents the class or interface defined by this class file

super_class: the value is zero (for the `Object` class) or the index of a CONSTANT_Class_info in the constant_pool; this CONSTANT_Class_info represents the direct superclass of the class defined by this class file

interfaces_count: the number of direct superinterfaces of this class or interface type

interfaces: an array, indexed from 0 to interfaces_count-1, of values representing the indexes of CONSTANT_Class_info items in the constant_pool; these CONSTANT_Class_info structures represent interfaces that are direct superinterfaces of this class or interface type

fields_count: the number of field_info structures in the fields table

fields: an array of field_info structures; it does not include fields that are inherited from superclasses or superinterfaces

methods_count: the number of method_info structures in the methods table

methods: an array of method_info structures; it does not include methods that are inherited from superclasses or superinterfaces

attributes_count: the number of attributes in the attributes table of this class

attributes: an array of attribute structures; certain attributes are predefined, one of them is `LineNumberTable`

The `LineNumberTable` [24] is an optional attribute, which supplies important information for debuggers. It provides a mapping between JVM instructions and the line numbers in the original source file. The LineNumberTable attributes may appear in any order and need not be one-to-one with source lines. This means, multiple LineNumberTable attributes may represent one source line number.

Hint: Compiling *.java files with `javac -g` generates all debugging information.

Since a Java Virtual Machine need not to know and does not know anything about the programming language, any language can be hosted by it. The only requirement is that the functionality of the language can be embodied in a valid `class` file. [23]

Figure 2.2 shows that source code is compiled to Java bytecode, which can be executed by a JVM. Figure 2.3 illustrates that Java bytecode can be executed on any computer platform for which a JVM is implemented.

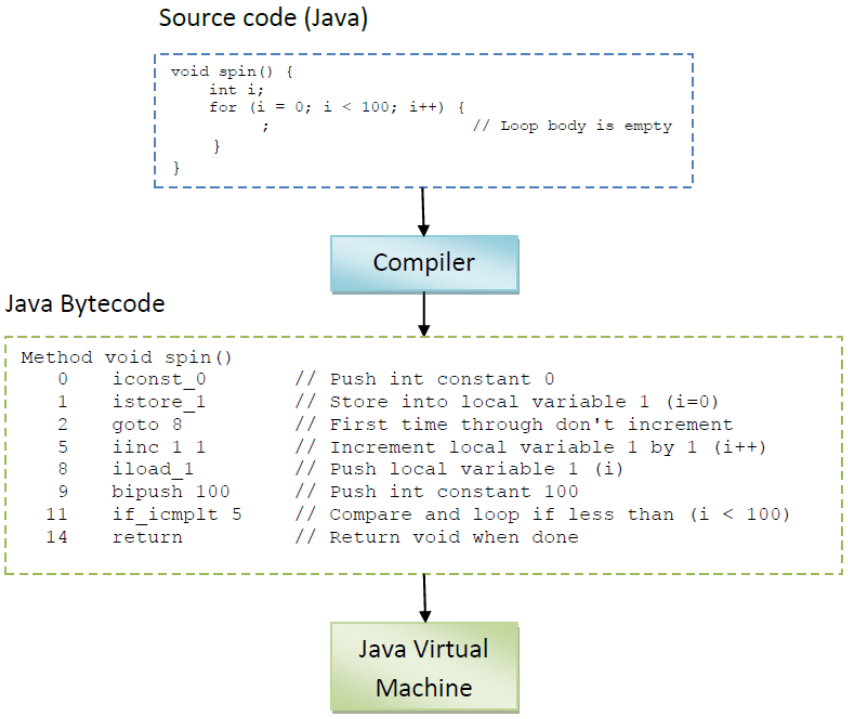

Figure 2.2.: A compiler transforms source code into Java bytecode. The JVM executes Java bytecode. This and more examples of compiling Java source code for the JVM are provided by [22].

As we can see in Figure 2.2 the JVM is stack-oriented. Many operations take operands from the operand stack of the current frame and/or push a result. Each

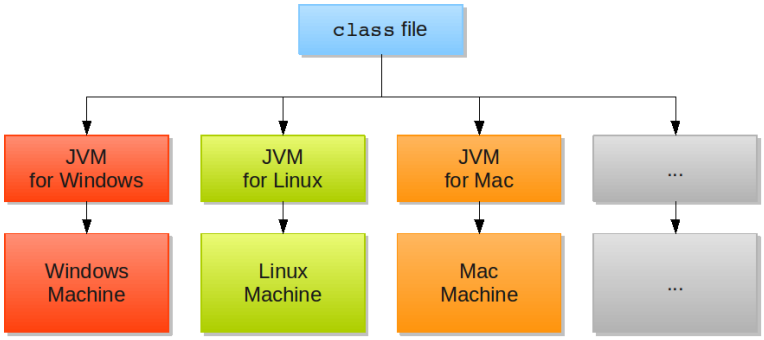

Figure 2.3.: A Java `class` file can be executed on any computer platform for which a JVM is implemented.

time a method is invoked, a new frame is created with a new operand stack and a set of local variables. The "i" in front of many opcodes stands for the type of the values, thus `int`. dconst_0, e.g., would push a double constant 0.0 The number in front of the opcodes is the index of the opcode in the byte-array of the JVM code of the method. [22]

2.4. Java Bytecode Analysis

The project of [20] deals with bytecode-based program analysis. This approach builds a front-end that processes bytecode to gather data, such as control flow graph, symbol table and def-use data. The back-end could then compute analytic information. The reason of processing bytecode is that that way a program analysis tool can be built more cheaply. The problems of performing syntax analysis and symbol table construction are delegated to existing tools, such as compilers.

In the paper [6] cost analysis of Java bytecode is established. They use bytecode, because sometimes there is no access to the source code and despite that one wants to have the cost information. The generated cost relations define at compile-time the cost of a program as a function of its input data size.

A debugging approach of Java bytecode programs is presented in [28]. They compute dynamic slices without needing access to source code. That's why programs written in any source language and compiled into Java bytecode could be diagnosed. Therefore, they implemented an instrumented JVM to produce the execution trace of a program. Additionally, their toolbox contains a slicer that reads the execution trace to compute the dynamic slices. Of course, source code line number information is needed to express a slice, but this information is present in compiled code.

2.5. Java Bytecode Manipulation Tools

The following sections describe tools, which allow manipulating Java bytecode, or in other words, manipulating Java `class` files. Of course, there are many different tools, but I decided to survey three of them: BCEL, ASM and Javassist, all three with satisfactory documentation.

2.5.1. BCEL

BCEL [16] (Bytecode Engineering Library) provides on a high level of abstraction static analysis and dynamic creation or transformation of Java `class` files.

The BCEL API consists of three parts [16]:

1. A package that reflects the `class` file format. It is not intended for bytecode manipulation. It is used to read and write `class` files. The main data structure is `JavaClass`. See Figure 2.4.

2. A package that enables to dynamically create and modify `JavaClass` or `Method` objects.

3. Utilities, such as code examples, `class` file viewer and converter (class file → HTML or class file → Jasmin assembly language)

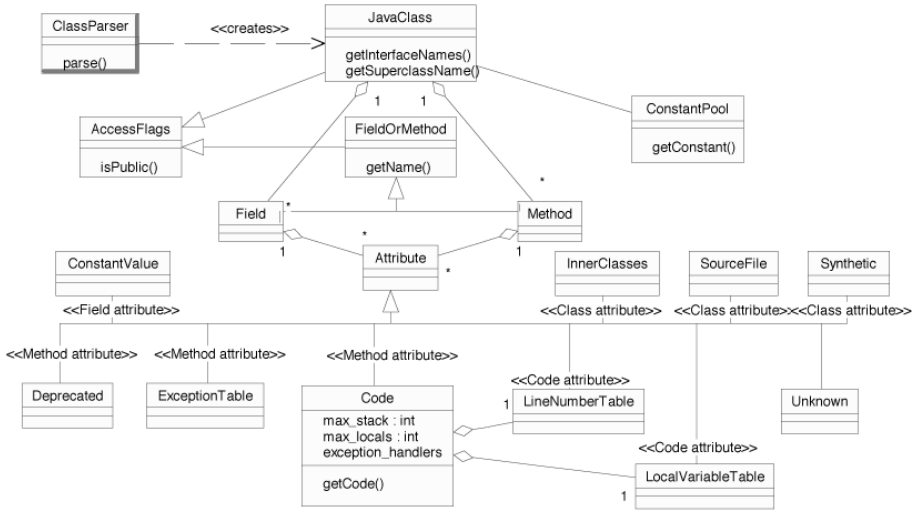

Figure 2.4.: UML diagram for the BCEL API [16].

BCEL supports the Visitor design pattern. The API is available under the terms of the Apache Software License.

2.5.2. ASM

The name ASM [9] is a reference to the `__asm__` keyword in C. ASM is designed to generate and transform Java classes at runtime, but also offline. This library provides two APIs:

- core API: event based representation
- tree API: object based representation

event based model: A class is represented with a sequence of events and each event represents an element of the class.

object based model: A class is represented with a tree of objects and each object represents a part of the class and has references to its constituents.

Of course, both of them have advantages, as well as disadvantages. The event based API is faster and requires less memory. However, there is only one element available at any given time. With the object based API the whole class is available in memory.

However, both APIs maintain no information about the class hierarchy. They can only manage one class at a time.

ASM [10] uses the Visitor design pattern without explicitly representing the visited tree with objects. This approach helps to achieve the goal of being as small and as fast as possible. The size and the runtime performance are compared to that of BCEL and SERP (similar to BCEL). The result was that ASM is much smaller and faster than these tools.

2.5.3. Javassist

Javassist [12, 14] (Java programming assistant) is a class library for editing Java Bytecode at compile time or load time. Besides, new classes can be defined at runtime. The difference to most other tools is the source-level abstraction Javassist provides. This enables programmers to manipulate Java bytecode without detailed knowledge of bytecode instructions and the structure of a `class` file. Javassist instead takes source text and compiles it into bytecode before inserting it into a `class` file.

The three main application areas of Javassist are [12]:

Aspect Oriented Programming (AOP): Javassist allows to introduce new methods into a class and to insert before/after/around advices at the caller as well as at the callee sides.

Runtime Reflection: Javassist enables to use a metaobject to control method calls on base-level objects.

Remote method invocation: Javassist can be used to call a method on a remote object running on a web server. However, no stub compiler (such as `rmic`) is needed, because the stub code is dynamically produced by Javassist.

2.6. Summary

Spectrum-based debugging is an often used approach, also because of its simplicity and effectiveness. It is a statistical approach, which uses similarity coefficients to rank program entities according to their possibility of being faulty. To realize spectrum-based debugging of Java bytecode programs, the information about the source line numbers of the statements is necessary. This is also available in the debugging information of a `class` file. The bytecode of the `class` file can be manipulated to gain the execution trace of a program. There are a lot of libraries to manipulate Java bytecode. Each Java bytecode manipulation tool has its advantages and disadvantages. The one used in this project is Javassist. It is easy to use, because of the source-level abstraction it provides. The next sections describe how spectrum-based debugging and Java bytecode manipulation are combined to a new approach.

3. Software Analysis and Design

This chapter deals with the analysis and the design of the software to implement. Section 3.1 covers the software requirements and Section 3.2 the software design.

3.1. Software Requirements

This section lists the defined requirements of the software to be implemented.

1. The framework should search the fault of a given program under test by using spectrum-based debugging.
2. Therefore, it manipulates a given `class` file of the program under test to extract runtime information.
3. To perform spectrum-based debugging a test suite is needed. The framework should take JUnit-tests, which test the program under test.
4. The aim of the framework is to output the possibility of being faulty of each program entity of the manipulated `class` file.

3.2. Software Design

This section describes the software design. It explains the selection of the programming language and the manipulation tool. Additionally, UML diagrams are provided.

3.2.1. Programming Language and Tools

The programming language Java was chosen. It is an object oriented programming language and thus the software written in Java is easily expandable and changeable. This ensures to enhance the life-cycle of the software. Besides, JUnit tests should be run and therefore, Java is necessary. Another motive for using Java is that Java bytecode should be manipulated. There are a lot of bytecode manipulation libraries written in Java, which can be integrated into Java programs.

The chosen Java bytecode manipulation tool is Javassist. It is easy to use, because of the source-level abstraction it provides. This means, Javassist takes source text and compiles it before inserting it into the `class` file. The advantage is that the

programmer needs not to know bytecode instructions and the structure of the `class` file. Therefore, Java bytecode manipulation is less fault-prone.

3.2.2. UML Diagrams

The following UML diagrams illustrate the structure and the functionality of the software to implement. Figure 3.1 shows the class diagram. To ensure expandability

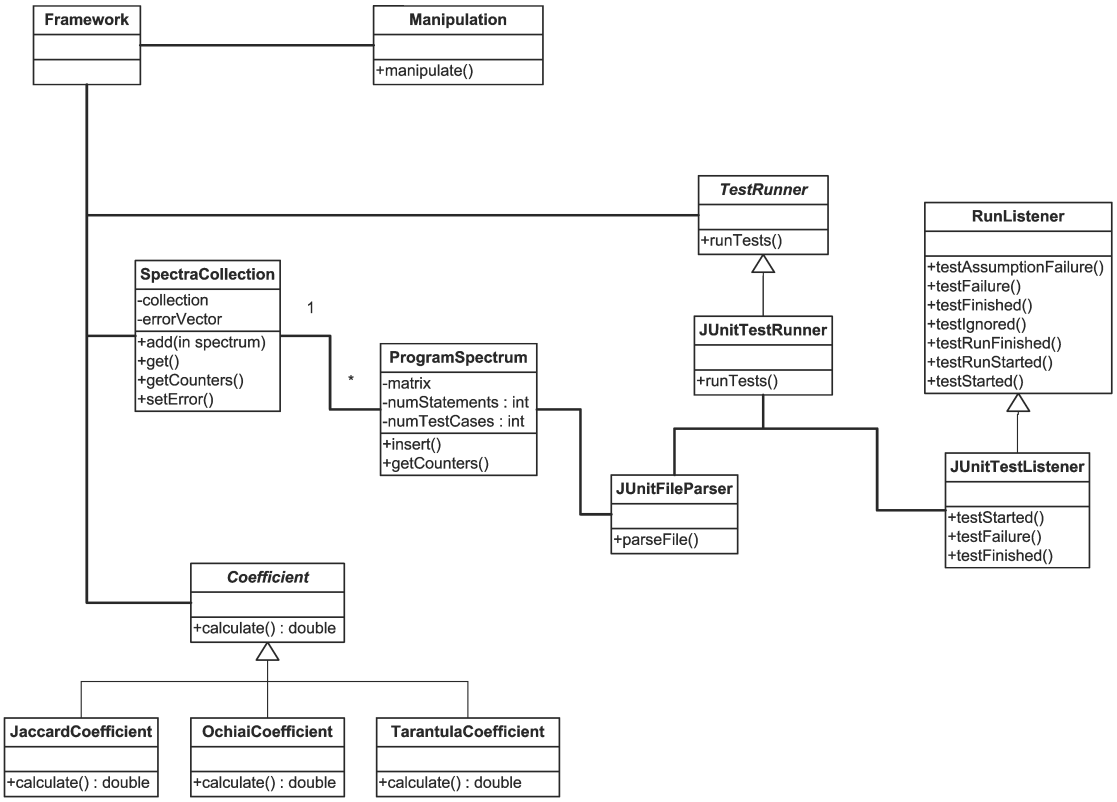

Figure 3.1.: class diagram

there are two interfaces: *`TestRunner`* and *`Coefficient`*. Primarily `JUnitTestRunner` implements *`TestRunner`* to be able to execute JUnit tests. The coefficients, which will be implemented, are the `OchiaiCoefficient`, `JaccardCoefficient` and the `TarantulaCoefficient`.

The sequence of the program to implement is presented in Figure 3.2. At first, the *.class files have to be manipulated. Then, the JUnit tests are run to get the execution trace. The `JUnitTestListener` has to be added to react to the events of the running JUnit tests (start, finish and fail). The created logfile needs to be parsed. For each *.class file a `ProgramSpectrum` object is created and added to the `SpectraCollection`. When a statement should be marked in the matrix as

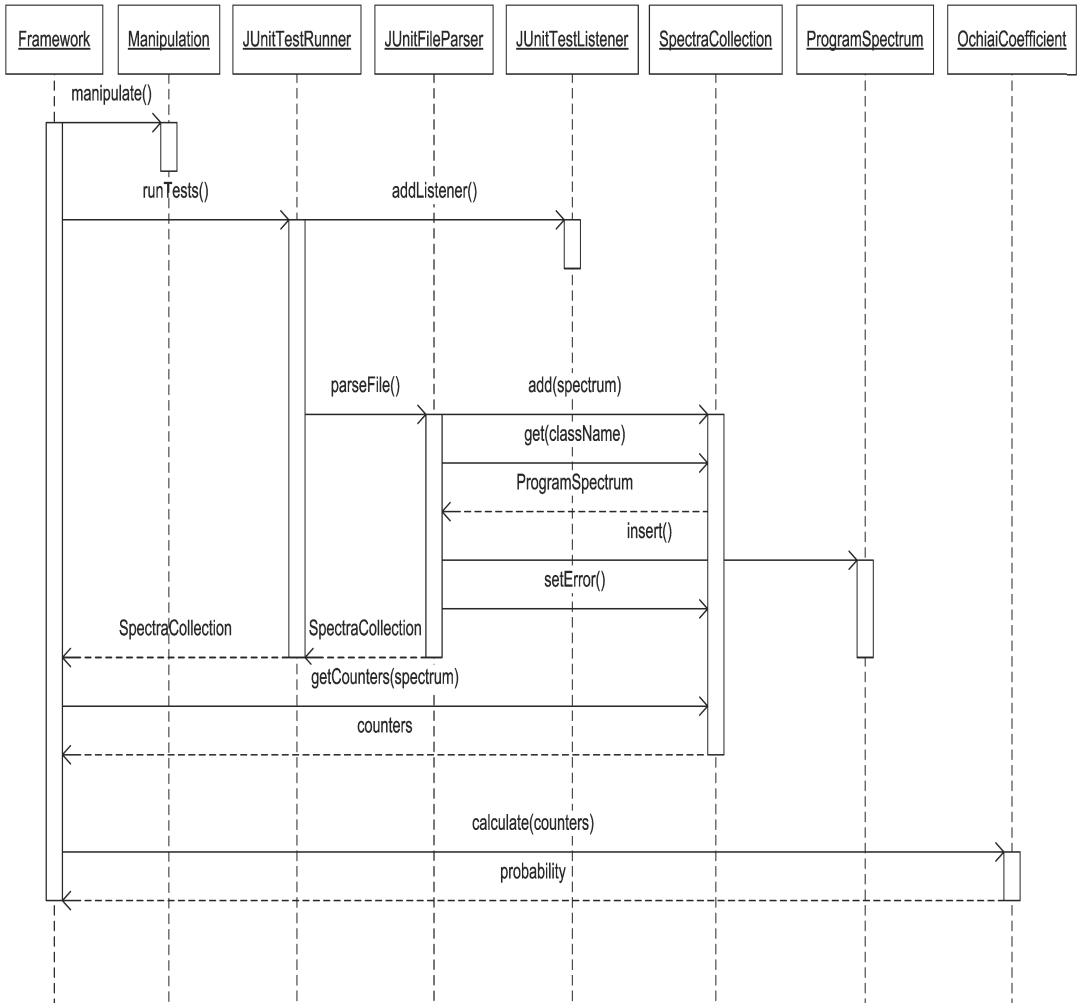

Figure 3.2.: sequence diagram

visited or not, the appropriate `ProgramSpectrum` object is get and a 0 or 1 is inserted into the matrix at the adequate position. If a failure occurred during a test run, the error will be set in the error vector. The complete `SpectraCollection` is then returned to the `Framework`. The `Framework` calls for each `ProgramSpectrum` object in the `SpectraCollection` the method `getCounters()` of the `SpectraCollection`. Finally, the possibility of being faulty is calculated for each statement of the represented class with the help of a similarity coefficient. In this case the `OchiaiCoefficient` is used.

4. Implementation

This chapter explains how the framework was implemented. Section 4.1 describes each single class with its methods. The used libraries are listed in Section 4.2. Section 4.3 shows the output of the framework and the continue processing. Finally, Section 4.4 summarizes the program flow.

It has been decided to insert standard output messages into the given Java bytecode with the help of Javassist. Later, when this program is executed, the standard output messages are diverted to a file. This approach enables to execute only parts of the program. There is no need to take care of any logging object (Has it been created? Has it been initialized? Is it known by the invoked program entity? ...). Tests which only invoke a method can thus be executed without any problem.

The framework takes the following parameters:

	description	obligatory?	
-cp	⟨ classpath ⟩	-	(1)
-c	⟨ class file to manipulate ⟩	√	
	⟨ name of the methods to manipulate ⟩	-	(2)
-l	⟨ level [m | s] ⟩	-	(3)
-lf	⟨ logfile name ⟩	√	(4)
-jp	⟨ path to JUnit test(s) ⟩	-	(5)
-j	⟨ JUnit class ⟩	√	(6)
-sc	⟨ similarity coefficient ⟩	-	(7)

Table 4.1.: The parameters taken by the framework.

(1) `-cp` indicates a pathlist separated with semicolons (in Windows) or colons (in Linux). It is not obligatory, or to be more precise, the path has to be specified if the class file cannot be found in the working directory.
(2) `-c` declares the name of a class file including packages, which has to be manipulated. This information is obligatory. Additionally, one can limit the methods to manipulate. The only thing to keep in mind is that if there are more methods with the same name, but different parameters, only one of these methods is manipulated. However, it cannot be guaranteed which one. It is possible to declare more than one class file to manipulate, each of them introduced with `-c`.
(3) `-l` allows setting the level, considered by the program. It can either be set to

method-level (`m`) or to statement-level (`s`). By default the statement-level is considered.
(4) Since the program writes the runtime-behavior of the program under test into a file, a logfile-name is needed. It is expected that the parameter `-lf` specifies it.
(5) `-jp` enables to declare the path to the JUnit test(s). Similarly to the classpath parameter, the path has to be specified if the JUnit test classes cannot be found in the working directory.
(6) `-j` declares the name of a JUnit class file including packages. This information is also obligatory. It is possible to declare more than one JUnit class file, each of them introduced with `-j`.
(7) `-sc` specifies the similarity coefficient. The supported values are: `ochiai`, `jaccard` or `tarantula`. By default the Ochiai coefficient is set.
In the following a few examples of calling the framework are listed.

```
-cp C:\java\testprogram; -c source.Framework -lf C:\java\testprogram\
output\result.log -jp C:\java\testprogram -j junit.FrameworkTest -sc
ochiai

-c test_src.Test -c test_src.Test2 doTest -lf log.txt -j
junit.FirstTest -sc jaccard

-c src.Class1 -c src.Class2 method1 method2 -lf output.log -j
src.junit.Test1 -j src.junit.Test2

-c Class -l m -lf output\log.txt -jp C:\junitTests -j junitTest
-sc tarantula
```

4.1. Description of Classes and Methods

The following section lists the implemented classes and their methods. Each method is described in detail.

4.1.1. Framework

`Framework` is the main class. It controls the sequence of the program. First of all, the committed arguments are parsed and saved. These arguments have to be delivered from the user when executing the program. The arguments are described in detail in Table 4.1.

The next step is to manipulate the class files. `Framework` expects the number of lines to be returned from `Manipulation` class. The number of lines is then given to a `TestRunner`. It runs the tests and writes the execution trace into a file. Additionally, the `JUnitTestRunner` calls the `JUnitFileParser`, which parses the file and creates a `SpectraCollection`. This `SpectraCollection` object is returned

to `Framework`. The `Framework` gets the counters of each `ProgramSpectrum` object from the `SpectraCollection`. The counters of one statement are given to the `Coefficient`, which returns the possibility of the considered statement. Finally, the possibility of being faulty of each statement for each manipulated class file is displayed on the console.

4.1.2. Manipulation

Java Bytecode Manipulation is carried out with the help of the Javassist library. See also Section 4.2.1. Standard output messages are inserted to collect runtime information when executing the tests later.

`protected HashMap<String, Integer> manipulate():` is called from the `Framework`. For each class to manipulate either `manipulateMethods()` or `manipulateAllMethods()` is called, depending on whether specific methods are given as arguments. It returns a mapping of class name and the number of source lines of the class or the number of methods to manipulate, depending on whether the statement level or method level is considered. This information is used to define the size of the program spectra.

`private void manipulateMethods(CtClass c, ArrayList<String> L):` assembles strings with class name and method name for all methods given in `L`. It calls `insertInformation()` for each method to manipulate. If the method level is considered, the number of methods to manipulate is saved.

`private void manipulateAllMethods(CtClass ctclass):` assembles strings with class name and method name for all methods of the `ctclass` object. It calls `insertInformation()` for each method to manipulate. If the method level is considered, the number of methods to manipulate is saved.

`private void insertInformation(String info, CtBehavior behav):` If only method level is considered, the string is inserted as standard output message at the beginning of the method. On the other hand, if the statement-level is considered, the string extended with the line number is added to each statement as standard output message. A line, which only contains a bracket is not considered as statement by Javassist and thus not manipulated. Besides, the maximal source line number is detected while manipulating the `*.class` file, if the statement level is considered.

4.1.3. TestRunner

The *`TestRunner`* interface is implemented by `JUnitTestRunner`. Additionally, the implementation of `TcasTestRunner` was introduced to use a test program for which no JUnit tests exist. The following methods have to be implemented by any implementation of the *`TestRunner`* interface:

`SpectraCollection runTests(HashMap<String, Integer> lineNumbers)` : should run the saved tests and return a `SpectraCollection` object. The given mapping of class name and the number of source lines of the class or the number of methods to manipulate is used to define the size of program spectra.

`void setPath(String path):` saves the given path to the test classes.

`void addClass(String className):` adds a test class to run.

`void setFilename(String filename):` saves the log file name.

JUnitTestListener:

`JUnitTestListener` extends `org.junit.runner.notification.RunListener`. The methods are called while a JUnit test class is executed. The implemented methods of `JUnitTestListener` are listed below. They allow noticing when atomic tests start, fail and finish. All of them write standard output messages, which are diverted into the logfile.

`public void testStarted(Description description):` writes "START:" and method name if an atomic test case is started.

`public void testFailure(Failure failure):` writes "FAILURE OCCURED!" and the failure description if an atomic test case failed.

`public void testFinished(Description description):` writes "FINISHED!" if an atomic test case finished.

JUnitFileParser:

The `JUnitFileParser` is responsible for reading out the information of the logfile and saving it in a `SpectraCollection` object. The method

```
public SpectraCollection parseFile(String file, Result[] result,
            HashMap<String, Integer> lineNumbers):
```

takes the logfile name, the results of the JUnit tests and a mapping of class name and the number of source lines in the class. From the size of the results one can conclude the number of test cases to create a `SpectraCollection` object. For each class a `ProgramSpectrum` object is created, which is added to the collection. The class name and the number of source lines are taken from the map. As described above, each test case is initiated with "START:", followed by all statements or methods visited when the test case was executed. Each visited statement has to be marked in the matrix of the `ProgramSpectrum` object. Therefore `collection.get(className).insert(1, visitedStatement, testCase)` is called. If "FAILURE OCCURED!" indicates a failure, the error vector has to be updated with `collection.setError(1, testCase)`.

4.1.4. ProgramSpectrum

One `ProgramSpectrum` object saves the runtime information of one class file in a matrix. The constructor expects the class name and furthermore the number of statements and the number of test cases to determine the size of the matrix. Each matrix value is initialized with 0.

`protected void insert(int val, int statement, int testcase):` is used to insert a binary digit (0,1) into the matrix. The value 0 represents that the given statement was not performed when executing the given test case. On the contrary, the value 1 represents that the given statement was performed.

`protected int[] getCounters(int[] errorVector, int statement):` returns the counters of a statement as int array. The returned array is composed as follows:
$count[0] = a_{00}$ increases when statement was not executed and test case was not erroneous
$count[1] = a_{01}$ increases when statement was not executed and test case was erroneous
$count[2] = a_{10}$ increases when statement was executed and test case was not erroneous
$count[3] = a_{11}$ increases when statement was executed and test case was erroneous

Additionally, it provides some methods to get information about the spectrum, also outside the object:

`public String getClassName():` returns the class name of the class file represented by the `ProgramSpectrum` object.

`public int getNumStatements():` returns the number of statements of the class file represented by the `ProgramSpectrum` object.

`public int getNumTestCases():` returns the number of test cases used to test the class files.

4.1.5. SpectraCollection

The `SpectraCollection` class is a collection of `ProgramSpectrum` objects. Besides, it saves the error vector. Therefore, the number of test cases has to be given to the constructor. Each value of the error vector is initialized with 0.

`public void add(ProgramSpectrum spectrum):` adds a `ProgramSpectrum` object.

`protected void setError(int val, int testcase):` allows to fill the error vector with binary digits (0,1). The value represents if the given test case was faulty (1) or not (0).

Two methods are provided to get a specific `ProgramSpectrum` object. Besides, there are two methods to get the counters of a specific `ProgramSpectrum` object. One takes the class name of the class file represented by the `ProgramSpectrum` object. The other one takes the index of the collection, where the `ProgramSpectrum` object is saved.

`public ProgramSpectrum get(String className):` takes the class name as String and returns the appropriate `ProgramSpectrum` object.

`public ProgramSpectrum get(int index):` chooses the `ProgramSpectrum` object with the given index.

`protected int[][] getCounters(String className):` takes the class name as String. It returns the counters of each statement of the class represented by the `ProgramSpectrum` object.

`protected int[][] getCounters(int index):` takes the index to choose the `ProgramSpectrum` object in the collection. It returns the counters of each statement of the class represented by the `ProgramSpectrum` object.

`private int[][] counters(ProgramSpectrum spectrum):` is called by `getCounters(String className)` as well as by `getCounters(int index)`. The method calls `getCounters(int[] errorVector, int statement)` of `ProgramSpectrum` for each statement of the class represented by the `ProgramSpectrum` object. Finally, it returns the counters of each statement.

`public int size():` returns the size of the collection. In other words, it returns the number of `ProgramSpectrum` objects stored.

`public int getNumTestCases():` returns the number of test cases used to test the class files.

4.1.6. Coefficient

The *`Coefficient`* interface is implemented by `OchiaiCoefficient`, `JaccardCoefficient` and `TarantulaCoefficient`. The interface allows swapping these implementations. All coefficients implement the method:

```
public double calculate(int[] counters)
```

The method takes an array as argument, which contains the four counters described in Section 2.2.2 (a_{11}, a_{10}, a_{01}, a_{00}). If any denominator is 0, the method returns the possibility of 0.0 instead of NaN (Not a Number). Further information about the coefficients can be found in Section 2.2.3. The formulas are here again written down:

- Jaccard Coefficient: $s = \frac{a_{11}}{a_{11}+a_{01}+a_{10}}$
- Ochiai Coefficient: $s = \frac{a_{11}}{\sqrt{(a_{11}+a_{01})*(a_{11}+a_{10})}}$
- Tarantula Coefficient: $s = \frac{\frac{a_{11}}{a_{11}+a_{01}}}{\frac{a_{11}}{a_{11}+a_{01}}+\frac{a_{10}}{a_{10}+a_{00}}}$

4.2. Used Libraries

This section refers to the used libraries. Javassist was used to manipulate class files and the JUnit library is necessary to enable running JUnit tests.

4.2.1. Javassist

The javaDocs of Javassist can be found at [13]. The used classes of the Javassist library are described briefly. For further information see the javaDocs.

javassist.ClassPool: is a container of `CtClass` objects. If `get(String classname)` is called, the locations represented by `ClassPath` are searched to find the appropriate class file. A `CtClass` object, representing the class file is created and returned.

javassist.CtBehavior: is the abstract super class of `CtMethod` and `CtConstructor`. This means, a `CtBehavior` object represents a method or a constructor.

javassist.CtClass: a `CtClass` object represents a Java class.

javassist.CtMethod: a `CtMethod` object represents a method.

4.2.2. JUnit

A javaDoc of JUnit 4.8 can be found at [8]. The used classes of the JUnit library are described briefly. For further information see the javaDocs.

org.junit.runner.JUnitCore: is used for running JUnit tests. It supports JUnit 4 and JUnit 3.8.x as well as mixtures.

org.junit.runner.Result: collects and summarizes information from the running JUnit tests.

org.junit.runner.Description: describes a JUnit test, either a test which is to be run or which has been run.

org.junit.runner.notification.Failure: consists of the `Description` of the failed JUnit test and the thrown exception.

org.junit.runner.notification.RunListener: provides methods to react to events that occur during a test run. Therefore, the appropriate methods of `RunListener` have to be overridden. The `JUnitTestListener` of the framework overrides the following methods of `RunListener`:

- `testFailure(Failure failure)`: atomic test case failed
- `testFinished(Description description)`: atomic test case finished
- `testStarted(Description description)`: atomic test case is to be started

4.3. Framework Output

This section describes which results are provided and how these results can help to find the faulty statement.

As already mentioned, the framework supplies the possibility of being faulty for each statement of each considered class file. This information is written on the console in collaboration with some runtime information. Therefore, the framework is split into subprocesses. The runtime is given for each single subprocess.

- initialization
- manipulation
- run tests (including parsing logfile)
- calculating possibility values

The possibility values are written in the following format.

```
class: <class name>
line: x possibility: y
line: z possibility: 0.0
```

Each possibility value is a value between 0.0 and 1.0. The value 1.0 means the highest possibility of being faulty. Thus, the statements with the highest possibility value should be regarded first to find the fault. If the statements with the highest possibility do not contain the fault, one can regard the statements with the next smaller possibility. The aim, of course, is to find the fault with considering as few statements as possible.

4.4. Summarization of the program flow

This section summarizes the most important steps of the framework to get a better understanding. Therefore, Figure 4.1 visualizes a simplified program flow.

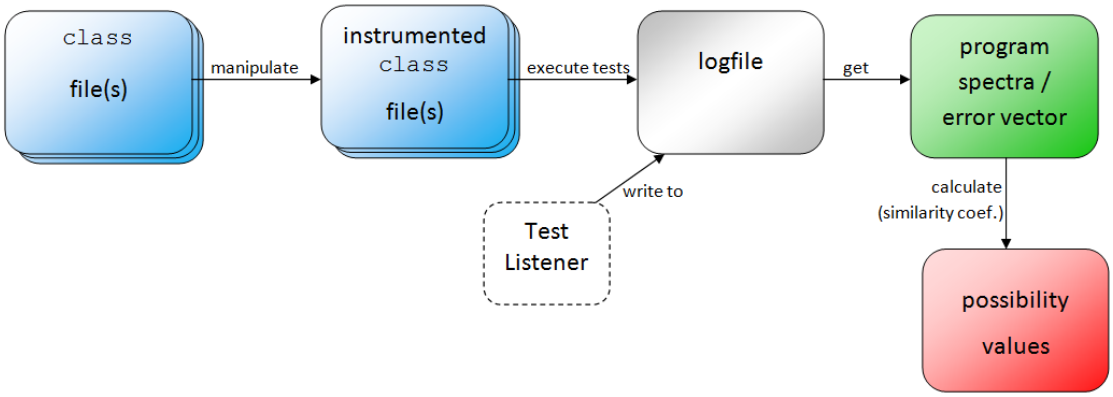

Figure 4.1.: Visualization of the program flow.

The framework gets one or more `class` files. To manipulate a class file means to add standard output messages to each source code statement. Each `class` file is replaced by its manipulated `class` file. After manipulating the Java bytecode of the `class` files the tests are executed. When a statement is visited, the appropriate standard output message is generated. The standard output is redirected to the logfile. That way the execution trace is saved. The test listener is responsible for logging when an atomic test case starts, when a failure occurs and when a test case has finished. The information written to the logfile is parsed and stored in program spectra objects and an error vector. The basic idea of program spectra is described in Section 2.2.1. The program spectra provide the information for the similarity coefficients (see Section 2.2.3) to calculate the possibility values for each line of source code. These values are written on the console as described in Section 4.3.

5. Empirical Evaluation

The empirical evaluation is carried out at statement level. The ranking is calculated with all three provided similarity coefficients:

- Jaccard coefficient
- Ochiai coefficient
- Tarantula coefficient

5.1. Test Environment

The used notebook has a 2 GHz Processor, 2 GB RAM and a 64-bit operating system. The installed operating system is Windows 7 Professional, additionally Ubuntu 10.04 is emulated in VMware Player.

To execute the framework a few libraries are needed. Therefore, Java standard edition (SE) version 1.6, JUnit version 4.8.2 and Javassist version 3.11.0 are installed.

5.2. Simple Test Program: Factorial

A really simple testprogram has been written, which calculates the factorial of a given integer. Line 15 contains the fault. In the following, the assumptions of the program are listed:

- The factorial of a negative integer is 0.
- The factorial of 0 is 1.
- The factorial of 1 is 1.
- The factorial of any higher integer is the product of this integer and all smaller positive integers.

To provide the possibility to reconstruct the whole example, the source code of `Factorial.java` and `FactorialTest.java` is written down.

```
1 package test_src;
2
3 public class Factorial {
```

```
4
5     public int calculate(int x)
6     {
7         if (x < 0)
8           return 0;
9         if (x == 0)
10          return 1;
11
12        int res = x;
13
14        for (int i = x-1; i > 1; i--)
15          res *= x;             //correct: res *= i;
16
17        return res;
18    }
19 }
```

The appropriate JUnit test (JUnit version: 4.8.2) checks if the program returns the expected results. Therefore, a test was written, which takes multiple pairs of parameters. The first parameter is the input for the program and the second one the expected output.

```
1  package test_src.junit;
2
3  import static org.junit.Assert.assertEquals;
4
5  import java.util.Arrays;
6  import java.util.Collection;
7
8  import org.junit.Test;
9  import org.junit.runner.RunWith;
10 import org.junit.runners.Parameterized;
11 import org.junit.runners.Parameterized.Parameters;
12
13 import test_src.Factorial;
14
15 @RunWith(Parameterized.class)
16 public class FactorialTest
17 {
18    private int input;
19    private int res;
20
21    public FactorialTest(int input, int res)
22    {
23       this.input = input;
24       this.res = res;
25    }
```

```
26
27      @Test
28      public void testFactorial()
29      {
30          Factorial fac = new Factorial();
31          int ret = fac.calculate(input);
32          assertEquals(res, ret);
33      }
34
35      @Parameters
36      public static Collection<Object[]> createTestInput()
37      {
38          return Arrays.asList(new Object[][]{
39                  {-10, 0},
40                  {-2, 0},
41                  {0, 1},
42                  {1, 1},
43                  {2, 2},
44                  {3, 6},
45                  {4, 24},
46                  {5, 120},
47                  {6, 720},
48                  {8, 40320},
49                  {10, 3628800},
50                  {12, 479001600}});
51      }
52  }
```

To manipulate the class `Factorial` in the package `test_src` and then run the test `FactorialTest` in the package `test_src.junit`, the program has to be called with the following arguments. Since these packages are in the same folder as the `src` package of the framework, no paths have to be declared.

```
-c test_src.Factorial -j test_src.junit.FactorialTest -lf log.txt -sc
                              <coefficient>
```

While executing `FactorialTest` a logfile is produced, which logs the visited source lines of `Factorial.java` for each test case. An excerpt is shown below. The `START: ...`, `FINISHED!` and `FAILURE OCCURED! ...` lines are outputs from `JUnitTestListener`. The other output lines result from the manipulation of the class file.

```
START: testFactorial[0]
-----------------------------------------------
test_src.Factorial method: calculate() 1 line: 7 visited!
test_src.Factorial method: calculate() 1 line: 8 visited!
FINISHED!
START: testFactorial[1]
```

```
------------------------------------------------
test_src.Factorial method: calculate() 1 line: 7 visited!
test_src.Factorial method: calculate() 1 line: 8 visited!
FINISHED!
START: testFactorial[2]
------------------------------------------------
test_src.Factorial method: calculate() 1 line: 7 visited!
test_src.Factorial method: calculate() 1 line: 9 visited!
test_src.Factorial method: calculate() 1 line: 10 visited!
FINISHED!
START: testFactorial[3]
------------------------------------------------
test_src.Factorial method: calculate() 1 line: 7 visited!
test_src.Factorial method: calculate() 1 line: 9 visited!
test_src.Factorial method: calculate() 1 line: 12 visited!
test_src.Factorial method: calculate() 1 line: 14 visited!
test_src.Factorial method: calculate() 1 line: 17 visited!
FINISHED!
START: testFactorial[4]
------------------------------------------------
test_src.Factorial method: calculate() 1 line: 7 visited!
test_src.Factorial method: calculate() 1 line: 9 visited!
test_src.Factorial method: calculate() 1 line: 12 visited!
test_src.Factorial method: calculate() 1 line: 14 visited!
test_src.Factorial method: calculate() 1 line: 17 visited!
FINISHED!
START: testFactorial[5]
------------------------------------------------
test_src.Factorial method: calculate() 1 line: 7 visited!
test_src.Factorial method: calculate() 1 line: 9 visited!
test_src.Factorial method: calculate() 1 line: 12 visited!
test_src.Factorial method: calculate() 1 line: 14 visited!
test_src.Factorial method: calculate() 1 line: 15 visited!
test_src.Factorial method: calculate() 1 line: 17 visited!
FAILURE OCCURED! testFactorial[5](test_src.junit.FactorialTest): expected:<6>
but was:<9>
FINISHED!
START: testFactorial[6]
------------------------------------------------
test_src.Factorial method: calculate() 1 line: 7 visited!
test_src.Factorial method: calculate() 1 line: 9 visited!
test_src.Factorial method: calculate() 1 line: 12 visited!
test_src.Factorial method: calculate() 1 line: 14 visited!
test_src.Factorial method: calculate() 1 line: 15 visited!
test_src.Factorial method: calculate() 1 line: 15 visited!
test_src.Factorial method: calculate() 1 line: 17 visited!
FAILURE OCCURED! testFactorial[6](test_src.junit.FactorialTest): expected:<24>
but was:<64>
FINISHED!
START: testFactorial[7]
------------------------------------------------
test_src.Factorial method: calculate() 1 line: 7 visited!
```

```
test_src.Factorial method: calculate() 1 line: 9 visited!
test_src.Factorial method: calculate() 1 line: 12 visited!
test_src.Factorial method: calculate() 1 line: 14 visited!
test_src.Factorial method: calculate() 1 line: 15 visited!
test_src.Factorial method: calculate() 1 line: 15 visited!
test_src.Factorial method: calculate() 1 line: 15 visited!
test_src.Factorial method: calculate() 1 line: 17 visited!
FAILURE OCCURED! testFactorial[7](test_src.junit.FactorialTest): expected:<120>
but was:<625>
FINISHED!
[snip...]
7 failure(s) of 12 run(s).
```

The resulting possibility of being faulty is written on the console for each line of code. The following output is displayed when the Ochiai coefficient is applied.

```
starttime: 1289295784493
start manipulation: 1289295784509
start run tests: 1289295784712
start calculate pos.: 1289295784899
--------------------results:--------------------
class: test_src.Factorial
line: 1 possibility: 0.0
line: 2 possibility: 0.0
line: 3 possibility: 0.7637626158259734
line: 4 possibility: 0.0
line: 5 possibility: 0.0
line: 6 possibility: 0.0
line: 7 possibility: 0.7637626158259734
line: 8 possibility: 0.0
line: 9 possibility: 0.8366600265340756
line: 10 possibility: 0.0
line: 11 possibility: 0.0
line: 12 possibility: 0.8819171036881969
line: 13 possibility: 0.0
line: 14 possibility: 0.8819171036881969
line: 15 possibility: 1.0
line: 16 possibility: 0.0
line: 17 possibility: 0.8819171036881969
------------------------------------------------
endtime: 1289295784899
```

It can be seen, that line 15, which really contains the fault, has a possibility of being faulty of 100 %. The start and end time are displayed in milliseconds. The total runtime was, hence, 406 ms. Most of the time is consumed with manipulating the class file.

The following output is displayed when the Jaccard coefficient is applied.

```
starttime: 1289296498288
start manipulation: 1289296498304
start run tests: 1289296498522
```

```
start calculate pos.: 1289296498694
--------------------results:--------------------
class: test_src.Factorial
line: 1 possibility: 0.0
line: 2 possibility: 0.0
line: 3 possibility: 0.5833333333333334
line: 4 possibility: 0.0
line: 5 possibility: 0.0
line: 6 possibility: 0.0
line: 7 possibility: 0.5833333333333334
line: 8 possibility: 0.0
line: 9 possibility: 0.7
line: 10 possibility: 0.0
line: 11 possibility: 0.0
line: 12 possibility: 0.7777777777777778
line: 13 possibility: 0.0
line: 14 possibility: 0.7777777777777778
line: 15 possibility: 1.0
line: 16 possibility: 0.0
line: 17 possibility: 0.7777777777777778
------------------------------------------------
endtime: 1289296498709
```

Applying the Jaccard coefficient, line 15 has also a possibility of 100 %. The total runtime was 421 ms.

The following output is displayed when the Tarantula coefficient is applied.

```
starttime: 1289296691198
start manipulation: 1289296691214
start run tests: 1289296691432
start calculate pos.: 1289296691635
--------------------results:--------------------
class: test_src.Factorial
line: 1 possibility: 0.0
line: 2 possibility: 0.0
line: 3 possibility: 0.5
line: 4 possibility: 0.0
line: 5 possibility: 0.0
line: 6 possibility: 0.0
line: 7 possibility: 0.5
line: 8 possibility: 0.0
line: 9 possibility: 0.625
line: 10 possibility: 0.0
line: 11 possibility: 0.0
line: 12 possibility: 0.7142857142857143
line: 13 possibility: 0.0
line: 14 possibility: 0.7142857142857143
line: 15 possibility: 1.0
line: 16 possibility: 0.0
line: 17 possibility: 0.7142857142857143
------------------------------------------------
endtime: 1289296691635
```

Applying the Tarantula coefficient, line 15 has also a possibility of 100 %. The total runtime was 437 ms.

What all coefficients have in common are the lines rated with 0 % error possibility. Besides, line 3 and line 7 have the lowest error rate, except the lines with 0 %. Since line 15 is only invoked when a failure occurs and never invoked when no failure occurs, it is 100 % equal to the error vector. This equality is detected by all three similarity coefficients.

5.3. TCAS

TCAS is a testprogram, with 41 different versions from the Software-artifact Infrastructure Repository (SIR), which can be found at http://sir.unl.edu/. The Java implementation of TCAS is introduced in [26]. Each version has another fault injected. The `TestRunner` was manually set to `TcasTestRunner`, instead of `JUnitTestRunner`, because no JUnit tests are provided. In the following, the results of all versions with a single fault are shown. The rejected versions, which have more than one fault injected, are: 10, 11, 15, 31, 32, 40.

The testprograms are executed with the help of a batch file:

```
@ECHO off
ECHO
ECHO batch file for tcas

set CLASSPATH=%CLASSPATH%;C:\Users\luser\MASTERARBEIT\svn\tools\junit-4.8.2.jar;
C:\Users\luser\MASTERARBEIT\svn\tools\javassist-3.11.0\javassist.jar;.

ECHO tcas_v01
del src\tcas*.class
javac src\*.java
java src.Framework -c src.tcas_v01 -j src.tcas_v01 -lf output\logtcas_v01.txt
-sc ochiai > output\logtcas_v01_ochiai.txt
del src\tcas*.class
javac src\*.java
java src.Framework -c src.tcas_v01 -j src.tcas_v01 -lf output\logtcas_v01.txt
-sc jaccard > output\logtcas_v01_jaccard.txt
del src\tcas*.class
javac src\*.java
java src.Framework -c src.tcas_v01 -j src.tcas_v01 -lf output\logtcas_v01.txt
-sc tarantula > output\logtcas_v01_tarantula.txt
[snip...]
```

This is only an excerpt of the batch file for the first version. The calls are the same for every single version.

5.3.1. Analysis

To analyze TCAS a matlab file was written. It reads the output created by the batch file and prints the information in diagrams.

```
close all;
clear all;

%% read data

number_of_versions = 41;
faultlinenum = {60, 48, 105, 64, 103, ...
                89, 20, 20, 75, [89, 94], ...
                [89, 94, 103], 103, 5, 6, [7, 103], ...
                20, 20, 20, 20, 57, ...
                57, 57, 75, 75, 82, ...
                103, 103, 48, 48, 48, ...
                [61, 66, 113], [79, 84, 114], 20, 109, 48, ...
                39, 43, 20, 82, [60, 111], ...
                64};
maxpos_jac = [];
maxpos_och = [];
maxpos_tar = [];
fault_pos_jac = [];
fault_pos_och = [];
fault_pos_tar = [];
versionnumber = [];
rank_jac = zeros(1,number_of_versions);
rank_och = zeros(1,number_of_versions);
rank_tar = zeros(1,number_of_versions);

for i=1:number_of_versions
fprintf('tcas_v%02d:\n',i);
%read jaccard
filename = sprintf('logtcas_v%02d_jaccard.txt',i);
[lines, pos_jac] = textread(filename, ...
'line: %d possibility: %f', 'headerlines', 6);
lines = lines(1:end-2);
pos_jac = pos_jac(1:end-2);

%read ochiai
[snip...]

%read tarantula
[snip...]

    % print the possibilities of being faulty for each statement and mark the
    % faulty statement with a bar
    [snip...]

close all;

%consider versions with a single fault injected
if (length(faultlinenum{i}) == 1)
    %get maximum possibility
maxpos_jac = [maxpos_jac, max(pos_jac)];
maxpos_och = [maxpos_och, max(pos_och)];
maxpos_tar = [maxpos_tar, max(pos_tar)];

%store possibility of faulty statement
fault_pos_jac = [fault_pos_jac, pos_jac(faultlinenum{i}(1))];
fault_pos_och = [fault_pos_och, pos_och(faultlinenum{i}(1))];
fault_pos_tar = [fault_pos_tar, pos_tar(faultlinenum{i}(1))];

versionnumber = [versionnumber, i];

%sort possibilities and get indices of all values equal to the
%fault possibility and calculate rank
[sorted, order] = sort(pos_jac);
index = find(sorted == fault_pos_jac(end));
```

```
rank_jac(i) = length(lines) - (index(length(index))-1);

%do the same for ochiai and tarantula
[snip...]
end
end

%print fault possibility compared to maximum possibility for each version
[snip...]
```

5.3.2. tcas_v01

This section looks at the first version of TCAS more precisely. The injected fault is in line 60. Only lines with a possibility different to 0.0 % are given for all provided similarity coefficients.
Ochiai coefficient:

```
starttime: 1289398454280
start manipulation: 1289398454305
start run tests: 1289398454560
start calculate pos.: 1289398456516
--------------------results:--------------------
class: src.tcas_v01
line: 20 possibility: 0.0873704056661038
line: 43 possibility: 0.3816490459683328
line: 48 possibility: 0.3378775335809819
line: 57 possibility: 0.3378775335809819
line: 58 possibility: 0.3378775335809819
line: 60 possibility: 0.36959118899518817
line: 64 possibility: 0.09659161779186924
line: 66 possibility: 0.3378775335809819
line: 75 possibility: 0.3378775335809819
line: 76 possibility: 0.3378775335809819
line: 78 possibility: 0.36959118899518817
line: 82 possibility: 0.09659161779186924
line: 84 possibility: 0.3378775335809819
line: 89 possibility: 0.337260018181911
line: 94 possibility: 0.3259854570353207
line: 103 possibility: 0.28674101656942175
line: 104 possibility: 0.28674101656942175
line: 105 possibility: 0.28674101656942175
line: 107 possibility: 0.28674101656942175
line: 109 possibility: 0.28674101656942175
line: 111 possibility: 0.3378775335809819
line: 112 possibility: 0.3378775335809819
line: 113 possibility: 0.3378775335809819
line: 118 possibility: 0.3378775335809819
line: 119 possibility: 0.45837113170323407
line: 120 possibility: 0.09679098150818871
line: 121 possibility: 0.015951580680567408
line: 123 possibility: 0.10032639104077389
line: 126 possibility: 0.28674101656942175
```

```
------------------------------------------------
endtime: 1289398456531
```

Jaccard coefficient:

```
starttime: 1289398705588
start manipulation: 1289398705619
start run tests: 1289398705868
start calculate pos.: 1289398707793
--------------------results:--------------------
class: src.tcas_v01
line: 20 possibility: 0.007633587786259542
line: 43 possibility: 0.17636986301369864
line: 48 possibility: 0.12866817155756208
line: 57 possibility: 0.12866817155756208
line: 58 possibility: 0.12866817155756208
line: 60 possibility: 0.1796875
line: 64 possibility: 0.04356435643564356
line: 66 possibility: 0.12866817155756208
line: 75 possibility: 0.12866817155756208
line: 76 possibility: 0.12866817155756208
line: 78 possibility: 0.1796875
line: 82 possibility: 0.04356435643564356
line: 84 possibility: 0.12866817155756208
line: 89 possibility: 0.1492063492063492
line: 94 possibility: 0.14566284779050737
line: 103 possibility: 0.08338720103425985
line: 104 possibility: 0.08338720103425985
line: 105 possibility: 0.08338720103425985
line: 107 possibility: 0.08338720103425985
line: 109 possibility: 0.08338720103425985
line: 111 possibility: 0.12866817155756208
line: 112 possibility: 0.12866817155756208
line: 113 possibility: 0.12866817155756208
line: 118 possibility: 0.12866817155756208
line: 119 possibility: 0.2727272727272727
line: 120 possibility: 0.03868194842406877
line: 121 possibility: 0.008032128514056224
line: 123 possibility: 0.04310344827586207
line: 126 possibility: 0.08338720103425985
------------------------------------------------
endtime: 1289398707819
```

Tarantula coefficient:

```
starttime: 1289398949181
start manipulation: 1289398949210
start run tests: 1289398949538
start calculate pos.: 1289398952535
--------------------results:--------------------
class: src.tcas_v01
```

```
line: 20 possibility: 1.0
line: 43 possibility: 0.7152228578009828
line: 48 possibility: 0.6251653320094139
line: 57 possibility: 0.6251653320094139
line: 58 possibility: 0.6251653320094139
line: 60 possibility: 0.7273144480563827
line: 64 possibility: 0.39384866630375615
line: 66 possibility: 0.6251653320094139
line: 75 possibility: 0.6251653320094139
line: 76 possibility: 0.6251653320094139
line: 78 possibility: 0.7273144480563827
line: 82 possibility: 0.39384866630375615
line: 84 possibility: 0.6251653320094139
line: 89 possibility: 0.6754060589809671
line: 94 possibility: 0.67192415854912
line: 103 possibility: 0.5015678782455886
line: 104 possibility: 0.5015678782455886
line: 105 possibility: 0.5015678782455886
line: 107 possibility: 0.5015678782455886
line: 109 possibility: 0.5015678782455886
line: 111 possibility: 0.6251653320094139
line: 112 possibility: 0.6251653320094139
line: 113 possibility: 0.6251653320094139
line: 118 possibility: 0.6251653320094139
line: 119 possibility: 0.8363772862865989
line: 120 possibility: 0.34468794664125774
line: 121 possibility: 0.15769398430688752
line: 123 possibility: 0.3808135250647409
line: 126 possibility: 0.5015678782455886
------------------------------------------------
endtime: 1289398952552
```

Having a look at the runtime, it can be seen that not manipulating extends the runtime, but running the test. There are 1578 test cases to run, instead of 12 of the simple testprogram. The Ochiai and the Tarantula coefficient rank the faulty statement third. The Jaccard coefficient ranks it even second. The next section summarizes all versions of TCAS.

5.3.3. Summarization of TCAS

The appendix A contains for each version, which has a single fault, one diagram. The diagram shows the possibility of being faulty for each statement, calculated with Jaccard coefficient, Ochiai coefficient and Tarantula coefficient. Figures 5.1, 5.2 and 5.3 show the possibility of the faulty statement compared to the maximum.

Table 5.1 shows for each considered version the calculated rank of the faulty statement. The ranks of all three similarity coefficients are declared.

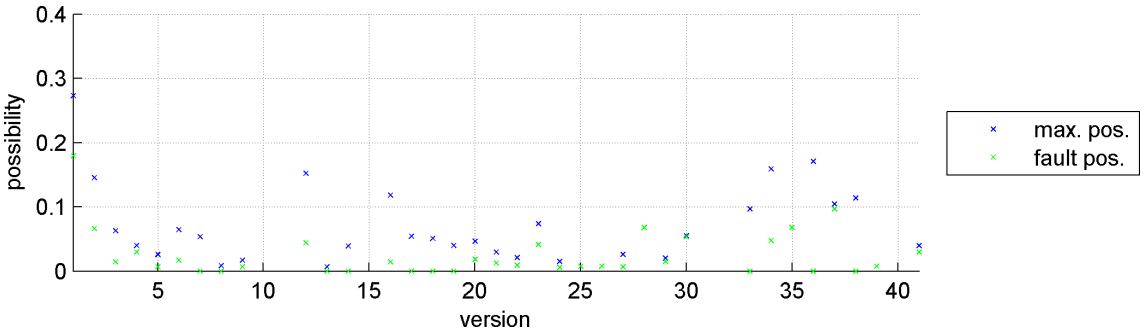

Figure 5.1.: The possibility of the faulty statement compared to the maximum possibility calculated by the Jaccard coefficient.

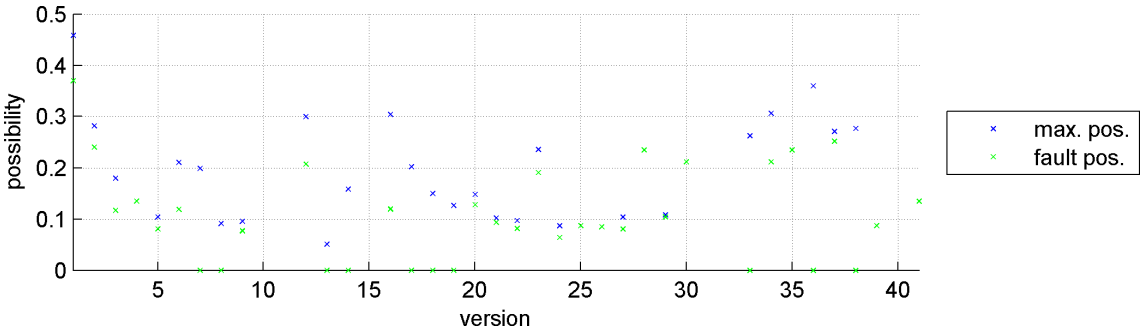

Figure 5.2.: The possibility of the faulty statement compared to the maximum possibility calculated by the Ochiai coefficient.

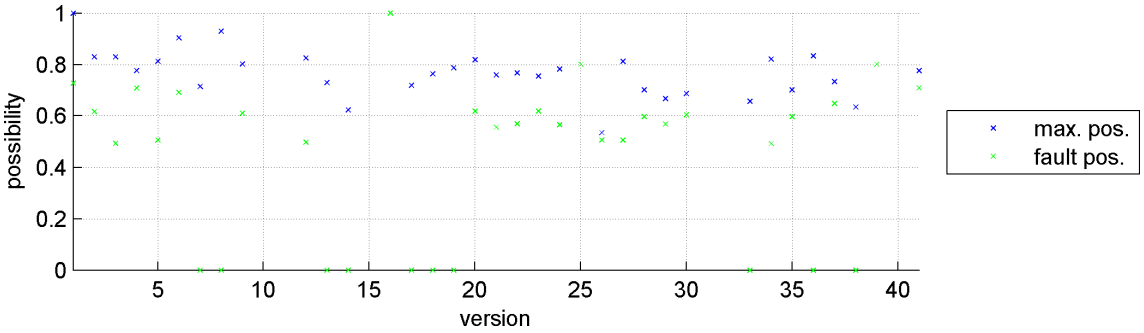

Figure 5.3.: The possibility of the faulty statement compared to the maximum possibility calculated by the Tarantula coefficient.

Version Number	Jaccard	Ochiai	Tarantula
01	2	3	3
02	6	5	6
03	19	16	20
04	2	1	2
05	18	2	18
06	6	7	6
07	28	28	28
08	25	25	25
09	8	6	8
12	17	16	18
13	25	25	25
14	29	29	29
16	28	28	1
17	28	28	28
18	29	29	29
19	28	28	28
20	5	3	5
21	6	9	6
22	6	10	6
23	7	5	9
24	7	11	7
25	1	1	1
26	4	1	4
27	18	2	18
28	2	1	6
29	6	7	7
30	2	1	6
33	29	29	29
34	17	17	18
35	2	1	6
36	29	29	29
37	3	14	2
38	29	29	29
39	1	1	1
41	2	1	2
average rank	13.54	12.80	13.29

Table 5.1.: The rank of the faulty statement calculated by three different similarity coefficients for each considered version.

Looking at Table 5.1 version 16 attracts attention. The Tarantula coefficient ranks the faulty statement first. The other coefficients rank it much worse. The faulty statement is line 20, which initializes a static array. This means, line 20 is called only once, when the first test case is executed. Looking at the similarity coefficients it can be seen that the Tarantula coefficient calculates a possibility of 100 %, if the faulty statement is only visited when a test case is executed where a failure occurs, even if it is only called once. No matter if there are test cases where a failure occurs and where the faulty statement is not visited. In contrary, the other coefficients take that into consideration. The following formulas can demonstrate that.

Tarantula coefficient: $\frac{\frac{a_{11}}{a_{11}+a_{01}}}{\frac{a_{11}}{a_{11}+a_{01}}+\frac{a_{10}}{a_{10}+a_{00}}} = \frac{\frac{1}{1+x}}{\frac{1}{1+x}+0}$

Jaccard coefficient: $\frac{a_{11}}{a_{11}+a_{01}+a_{10}} = \frac{1}{1+x+0}$

Ochiai coefficient: $\frac{a_{11}}{\sqrt{(a_{11}+a_{01})*(a_{11}+a_{10})}} = \frac{1}{\sqrt{(1+x)*(1+0)}}$

In the case, the first test case is not erroneous, line 20 is rated with 0.0 %, because it is never visited when an erroneous test case is executed. This happens in version 7, 8, 17, 18, 19, 33 and 38.

Executing version 13, 14 and 36 the faulty statement (line: 5/6/39) is also rated with 0.0 %. Line 5, 6 and 39 define static member variables and are never marked as visited when a test case detects a fault.

When the average rank is compared, the Ochiai coefficient supplies the best results. The average rank of the Tarantula coefficient is falsified by version 16, because the faulty statement is ranked first "by accident", as described above.

5.4. Initialization of Variables

To illustrate the point in time when a variable is initialized, the `Factorial.java` file is extended.

```
1  package test_src;
2
3  public class Factorial {
4
5      public static final int id = 0123;
6      public static String className = "class_name";
7      public double number = 0.0;
8      private short num = 1;
9
10     public int calculate(int x)
11     { // [snip...]
```

The corresponding logfile shows that the `static` variable is only initialized once, when the first test case is executed. A static variable belongs to the class and not

to the individual objects of this class. The `static final` variable is never marked as visited. In contrast, the member variables of the instances are initialized every time a new object is created. This means for this case, they are initialized every time a new test case is started. The output looks the same, regardless of executing the framework in Windows or Ubuntu.

```
START: testFactorial[0]
------------------------------------------------
test_src.Factorial method: <clinit>() 0 line: 6 visited!
test_src.Factorial method: <clinit>() 0 line: 3 visited!
test_src.Factorial method: Factorial() 1 line: 3 visited!
test_src.Factorial method: Factorial() 1 line: 7 visited!
test_src.Factorial method: Factorial() 1 line: 8 visited!
test_src.Factorial method: calculate() 2 line: 12 visited!
test_src.Factorial method: calculate() 2 line: 13 visited!
FINISHED!
START: testFactorial[1]
------------------------------------------------
test_src.Factorial method: Factorial() 1 line: 3 visited!
test_src.Factorial method: Factorial() 1 line: 7 visited!
test_src.Factorial method: Factorial() 1 line: 8 visited!
test_src.Factorial method: calculate() 2 line: 12 visited!
test_src.Factorial method: calculate() 2 line: 13 visited!
FINISHED!
START: testFactorial[2]
------------------------------------------------
test_src.Factorial method: Factorial() 1 line: 3 visited!
test_src.Factorial method: Factorial() 1 line: 7 visited!
test_src.Factorial method: Factorial() 1 line: 8 visited!
test_src.Factorial method: calculate() 2 line: 12 visited!
test_src.Factorial method: calculate() 2 line: 14 visited!
test_src.Factorial method: calculate() 2 line: 15 visited!
FINISHED!
START: testFactorial[3]
------------------------------------------------
test_src.Factorial method: Factorial() 1 line: 3 visited!
test_src.Factorial method: Factorial() 1 line: 7 visited!
test_src.Factorial method: Factorial() 1 line: 8 visited!
test_src.Factorial method: calculate() 2 line: 12 visited!
test_src.Factorial method: calculate() 2 line: 14 visited!
test_src.Factorial method: calculate() 2 line: 17 visited!
test_src.Factorial method: calculate() 2 line: 19 visited!
test_src.Factorial method: calculate() 2 line: 22 visited!
FINISHED!
[snip...]
```

5.5. JTopas - Java tokenizer and parser tools

JTopas is a testprogram taken from the Software-artifact Infrastructure Repository (SIR), which can be found at http://sir.unl.edu/. It provides `shell (*.sh)` scripts

to install a specific version (with or without errors) of the program. Therefore, it has to be executed on a Unix system.

Trying to find the injected faults of jtopas-0.4 a few problems appear. First of all, some faults to inject are faulty initializations of `static final` variables. As shown in Section 5.4 the initialization of `static final` variables are never marked as visited. Therefore, it is not possible to find the fault with spectrum-based debugging.

When `AbstractTokenizer` should be manipulated, a `CannotCompileException by javassist.bytecode.BadBytecode: conflict: int and de.susebox.java.util.TokenizerProperty` is thrown. It seems to be a bug of Javassist 3.11. One could think about why to manipulate abstract methods, but `AbstractTokenizer` does also contain implemented methods. The framework only manipulates methods if there is a method body. Using Javassist 3.4 also the class `AbstractTokenizer` can be manipulated without any exception.

Besides, some faults are injected by deleting a statement. The problem is, that a deleted statement cannot be ranked or rather it does not appear in the analysis.

To demonstrate how spectrum-based debugging behaves if a fault is injected, which affects multiple lines, the definition of a boolean variable in `if` and `else` block was changed. Thus, the definition is faulty in `if` as well as in `else` block. The parameters given to the framework are listed below. As described above, the framework is executed in Ubuntu emulated in VMware Player.

```
-cp /home/user/masterarbeit/svn/Testdaten/jtopas/source/ -c
de.susebox.java.io.ExtIOException -lf log_v1extioexception.txt -jp
/home/user/masterarbeit/svn/Testdaten/jtopas/source/junit/ -j
de.susebox.TestExceptions -sc <coefficient>
```

Statement 47 and 49 are faulty. If a test case is executed where a failure occurs, either line 47 or line 49 is visited, never both. Other statements, which are executed each time a failure occurs, are ranked much higher. The results of all three similarity coefficients are shown above. Only source lines with a possibility different to 0.0 % are listed.

Ochiai coefficient:

```
starttime: 1290977043774
start manipulation: 1290977043825
start run tests: 1290977044010
start calculate pos.: 1290977044104
--------------------results:--------------------
class: de.susebox.java.io.ExtIOException

line: 15 possibility: 0.5773502691896258
line: 19 possibility: 1.0
line: 23 possibility: 0.5773502691896258
line: 31 possibility: 0.5773502691896258
line: 32 possibility: 0.5773502691896258
line: 39 possibility: 0.5773502691896258
line: 40 possibility: 0.5773502691896258
line: 43 possibility: 1.0
line: 46 possibility: 1.0
```

```
line: 47 possibility: 0.5773502691896258
line: 49 possibility: 0.8164965809277261
line: 51 possibility: 1.0
line: 52 possibility: 1.0
line: 53 possibility: 1.0
line: 56 possibility: 0.5773502691896258
line: 59 possibility: 1.0
line: 61 possibility: 1.0
line: 63 possibility: 1.0
------------------------------------------------
endtime: 1290977044115
```

Jaccard coefficient:

```
starttime: 1290977132815
start manipulation: 1290977132846
start run tests: 1290977133050
start calculate pos.: 1290977133165
--------------------results:--------------------
class: de.susebox.java.io.ExtIOException

line: 15 possibility: 0.3333333333333333
line: 19 possibility: 1.0
line: 23 possibility: 0.3333333333333333
line: 31 possibility: 0.3333333333333333
line: 32 possibility: 0.3333333333333333
line: 39 possibility: 0.3333333333333333
line: 40 possibility: 0.3333333333333333
line: 43 possibility: 1.0
line: 46 possibility: 1.0
line: 47 possibility: 0.3333333333333333
line: 49 possibility: 0.6666666666666666
line: 51 possibility: 1.0
line: 52 possibility: 1.0
line: 53 possibility: 1.0
line: 56 possibility: 0.3333333333333333
line: 59 possibility: 1.0
line: 61 possibility: 1.0
line: 63 possibility: 1.0
------------------------------------------------
endtime: 1290977133176
```

Tarantula coefficient:

```
starttime: 1290977191691
start manipulation: 1290977191725
start run tests: 1290977191932
start calculate pos: 1290977192054
--------------------results:--------------------
class: de.susebox.java.io.ExtIOException

line: 15 possibility: 1.0
line: 19 possibility: 1.0
```

```
line: 23 possibility: 1.0
line: 31 possibility: 1.0
line: 32 possibility: 1.0
line: 39 possibility: 1.0
line: 40 possibility: 1.0
line: 43 possibility: 1.0
line: 46 possibility: 1.0
line: 47 possibility: 1.0
line: 49 possibility: 1.0
line: 51 possibility: 1.0
line: 52 possibility: 1.0
line: 53 possibility: 1.0
line: 56 possibility: 1.0
line: 59 possibility: 1.0
line: 61 possibility: 1.0
line: 63 possibility: 1.0
------------------------------------------------
endtime: 1290977192067
```

The reason why the Tarantula coefficient calculates either a possibility of 0.0 % or a possibility of 100 % is that all test cases, which call the `ExtIOException`, detect a fault. As already discussed, applying the Tarantula coefficient a statement is rated with 100 % if it is only visited when a failure occurs. The other coefficients rate a statement with 100 % only if it is visited every time a failure occurs and never if no fault is detected.

In addition, a newer version of JTopas was tested, jtopas-0.5.1. The first injected fault to `AbstractTokenizer` cannot be found. One test class creates a logfile, which is too big (> 400 MB). Executing the other test classes no failure occurs. The faulty statement is never executed. This means that no statement has a possibility of being faulty higher than 0.0 %. This case shows how important it is to have test cases, which cover the whole source code.

The parameters below effect the manipulation of `AbstractTokenizer` and `InputStremTokenizer`, which extends `AbstractTokenizer`. Besides, the three test classes, which supply results that can be evaluated, are executed.

```
-cp /home/user/masterarbeit/svn/Testdaten/jtopas/source/ -c
de.susebox.java.util.AbstractTokenizer -c
de.susebox.java.util.InputStreamTokenizer -lf log_v2tokenizer.txt -jp
/home/user/masterarbeit/svn/Testdaten/jtopas/source/junit/ -j
de.susebox.java.util.TestDifficultSituations -j
de.susebox.java.util.TestTokenizerProperties -j
de.susebox.java.util.TestTextAccess -sc <coefficient>
```

Injecting another fault, the test classes even deliver appropriate results to find the fault. The faulty statement is in line 655 of `AbstractTokenizer`. Below, the results of the three coefficients can be compared. This time only the faulty statement and lines with a higher or equal possibility are listed. To get an impression of the size of the class files also the last source line is given.

Ochiai coefficient:

```
starttime: 1291018716182
start manipulation: 1291018716624
start run tests: 1291018717524
start calculate pos: 1291018718860
--------------------results:--------------------
class: de.susebox.java.util.AbstractTokenizer
...
line: 655 possibility: 1.0
line: 656 possibility: 1.0
...
line: 673 possibility: 1.0
...
line: 1419 possibility: 0.2773500981126146
class: de.susebox.java.util.InputStreamTokenizer
...
line: 85 possibility: 0.2773500981126146
------------------------------------------------
endtime: 1291018718927
```

Jaccard coefficient:

```
starttime: 1291019133810
start manipulation: 1291019133862
start run tests: 1291019134287
start calculate pos: 1291019135114
--------------------results:--------------------
class: de.susebox.java.util.AbstractTokenizer
...
line: 655 possibility: 1.0
line: 656 possibility: 1.0
...
line: 673 possibility: 1.0
...
line: 1419 possibility: 0.07692307692307693
class: de.susebox.java.util.InputStreamTokenizer
...
line: 85 possibility: 0.07692307692307693
------------------------------------------------
endtime: 1291019135218
```

Tarantula coefficient:

```
starttime: 1291019251981
start manipulation: 1291019252114
start run tests: 1291019252579
start calculate pos: 1291019253522
--------------------results:--------------------
class: de.susebox.java.util.AbstractTokenizer
...
line: 655 possibility: 1.0
line: 656 possibility: 1.0
...
line: 673 possibility: 1.0
```

```
...
line: 1419 possibility: 0.5
class: de.susebox.java.util.InputStreamTokenizer
...
line: 85 possibility: 0.5
-----------------------------------------------
endtime: 1291019253744
```

No matter which similarity coefficient is used, the faulty statement is found. It even has a possibility of 100 %. Besides, the other lines with a possibility of 100 % are associated with the faulty statement. Line 655 is a `if` statement. The condition to be smaller was changed to smaller or equal. The following line 656 is executed if the condition is fulfilled. Line 673 calls the method containing the faulty `if` statement. The three test classes contain together 13 different test cases. One of them leads to a failure. The three lines rated with 100 % are only visited when a fault is detected.

When the `Token` class is tested a file with 86.7 MB is created, although only 9 test cases are executed. This shows how fast a file with 1332726 lines is created. Besides, no test case detects a fault. In the following the runtime is listed.

```
starttime: 1291035109104
start manipulation: 1291035109183
start run tests: 1291035109422
start calculate pos: 1291035150159
endtime: 1291035150174
```

The manipulation process needs 239 ms, running the tests lasts 40 s in the VMware Player. The whole runtime amounts 41 s. For comparison only, the `TestToken-Properties` is executed without starting the manipulation process. The resulting runtime is listed in the following.

```
starttime: 1291056971890
start manipulation: 1291056972023
start run tests: 1291056972028
endtime: 1291056975038
```

Running the test if no class file has been manipulated takes about 3 seconds. This is about a tenth of the runtime compared to the run with manipulation. However, thinking about how many lines are written to the file if the `Token` class was manipulated this difference is understandable.

5.6. Summary

This section summarizes the results of the empirical evaluation. Table 5.2 shows for each program under test the number of lines of code (LoC), as well as the ranks of the faulty statements and the number of test cases. The ranks are stated for all three supported similarity coefficients (Jaccard, Ochiai, Tarantula). The number in brackets specifies the number of statements with the same rank as the faulty statement (including the faulty one). The test cases are divided into positive and negative test cases. A positive test case is a test, where no failure occurs during execution. Thus, a negative test case is a test, where a failure occurs during execution. Table 5.3 summarizes for each program under test the runtime information as well as the size of the logfile, generated when executing the tests. The runtime information is averaged over the programs executed with the three different similarity coefficients. The column "runtime" lists the whole runtime of the framework, carrying out manipulation, running tests and calculating possibility values. The next column shows the time needed for manipulation. The runtime of executing the tests is displayed for the case using the original `class` files as well as using the manipulated ones. Note that the runtime of executing the tests includes parsing the logfile if the `class` files have been manipulated. The Factorial example in the first line shows that the runtime does also depend on the current system load. The execution time of the tests without a manipulated `class` file takes longer than using the manipulated one.

program	LoC	rank_J	rank_O	rank_T	pos. TCs	neg. TCs
Factorial	19	1 (1)	1 (1)	1 (1)	5	7
tcas_v01	128	2 (2)	3 (2)	3 (2)	1447	131
tcas_v02	128	6 (11)	5 (11)	6 (11)	1511	67
tcas_v03	128	19 (5)	16 (5)	20 (5)	1555	23
tcas_v04	128	2 (2)	1 (2)	2 (2)	1558	20
tcas_v05	128	18 (5)	2 (5)	18 (5)	1568	10
tcas_v06	128	6 (1)	7 (1)	6 (1)	1566	12
tcas_v07	128	27 (98)	27 (98)	27 (98)	1542	36
tcas_v08	128	24 (101)	24 (101)	24 (101)	1577	1
tcas_v09	128	8 (11)	6 (11)	8 (11)	1571	7
tcas_v12	128	17 (5)	16 (5)	18 (5)	1508	70
tcas_v13	128	24 (101)	24 (101)	24 (101)	1574	4
tcas_v14	128	28 (97)	28 (97)	28 (97)	1528	50
tcas_v16	128	27 (1)	27 (1)	1 (1)	1508	70
tcas_v17	128	27 (98)	27 (98)	27 (98)	1543	35
tcas_v18	128	28 (97)	28 (97)	28 (97)	1549	29
tcas_v19	128	27 (98)	27 (98)	27 (98)	1559	19
tcas_v20	128	5 (11)	3 (11)	5 (11)	1560	18
tcas_v21	128	6 (11)	8 (11)	6 (11)	1562	16
tcas_v22	128	6 (11)	9 (11)	6 (11)	1567	11
tcas_v23	128	7 (11)	5 (11)	9 (11)	1537	41
tcas_v24	128	7 (11)	10 (11)	7 (11)	1571	7
tcas_v25	128	1 (2)	1 (2)	1 (2)	1575	3
tcas_v26	128	4 (5)	1 (5)	4 (5)	1567	11
tcas_v27	128	18 (5)	2 (5)	18 (5)	1568	10
tcas_v28	128	2 (11)	1 (11)	6 (11)	1503	75
tcas_v29	128	6 (11)	6 (11)	7 (11)	1560	18
tcas_v30	128	2 (11)	1 (11)	6 (11)	1521	57
tcas_v33	128	28 (97)	28 (97)	28 (97)	1489	89
tcas_v34	128	17 (5)	17 (5)	18 (5)	1501	77
tcas_v35	128	2 (11)	1 (11)	6 (11)	1503	75
tcas_v36	128	28 (97)	28 (97)	28 (97)	1458	120
tcas_v37	128	3 (1)	14 (1)	2 (1)	1486	92
tcas_v38	128	28 (97)	28 (97)	28 (97)	1459	119
tcas_v39	128	1 (2)	1 (2)	1 (2)	1575	3
tcas_v41	128	2 (2)	1 (2)	2 (2)	1558	20
jtopas 0.5.1 (Tokenizer)	1420+86	1 (3)	1 (3)	1 (3)	12	1
jtopas 0.5.1 (Token)	251	-	-	-	9	0

Table 5.2.: Summary of test data (1).

program	runtime	manipulation time [ms]	exec. tests (with man.) [ms]	exec. tests (without man.) [ms]	logfile size
Factorial	421	213	187	203	7 KB
tcas_v01	2189	255	1883	718	1661 KB
tcas_v02	2137	245	1841	671	1657 KB
tcas_v03	2210	271	1888	609	1792 KB
tcas_v04	2158	240	1852	666	1664 KB
tcas_v05	2179	266	1862	807	1746 KB
tcas_v06	2128	255	1820	671	1662 KB
tcas_v07	2158	255	1852	718	1658 KB
tcas_v08	2205	245	1893	666	1661 KB
tcas_v09	2257	245	1961	718	1659 KB
tcas_v12	2439	317	2070	682	2061 KB
tcas_v13	2190	281	1857	692	1695 KB
tcas_v14	2111	302	1753	775	1431 KB
tcas_v16	2226	261	1903	697	1656 KB
tcas_v17	2220	271	1898	759	1658 KB
tcas_v18	2205	276	1878	645	1658 KB
tcas_v19	2340	302	1987	702	1659 KB
tcas_v20	2237	266	1914	671	1663 KB
tcas_v21	2257	256	1945	645	1607 KB
tcas_v22	2278	265	1945	676	1599 KB
tcas_v23	2132	271	1815	713	1597 KB
tcas_v24	2221	271	1898	713	1606 KB
tcas_v25	2257	276	1935	630	1660 KB
tcas_v26	2351	276	2028	728	1719 KB
tcas_v27	2247	287	1914	760	1746 KB
tcas_v28	2226	271	1903	707	1648 KB
tcas_v29	2221	271	1893	729	1655 KB
tcas_v30	2184	271	1857	697	1654 KB
tcas_v33	2283	281	1940	811	1668 KB
tcas_v34	2387	276	2059	526	2092 KB
tcas_v35	2138	244	1836	676	1648 KB
tcas_v36	2169	250	1872	780	1660 KB
tcas_v37	2142	245	1852	676	1669 KB
tcas_v38	2168	234	1878	754	1553 KB
tcas_v39	2148	250	1852	739	1660 KB
tcas_v41	2216	260	1893	588	1644 KB
jtopas 0.5.1 (Tokenizer)	1972	596	1035	692	2.3 MB
jtopas 0.5.1 (Token)	41070	239	40737	3010	86.7 MB

Table 5.3.: Summary of test data (2).

6. Conclusion and Future Work

This master's thesis presents a new approach which combines spectrum-based debugging and Java bytecode manipulation. The following sections provide the conclusions of the project and make some suggestions for future work.

6.1. Spectrum-based Debugging

A detected challenge of spectrum-based debugging is to find a faulty initialization of a `static` variable. The problem is that such a statement is only marked as visited once, when executing the first test case. If the first test case detects no fault, the initialization will even have a possibility of being faulty of 0.0 %. The initialization of a `static final` variable is never marked as visited. Therefore, the initialization of a `static final` variable does always have a possibility of being faulty of 0.0 %, even if it is faulty.

The empirical evaluation approved that the Ochiai coefficient performs best, compared to the two other tested. The Tarantula coefficient gives the most insufficient results. This affirms the conclusions of former works.

The most important thing to get satisfactory results of spectrum-based debugging is that the test cases cover the whole source code. A statement can only be evaluated as faulty if it is processed.

6.2. Java Bytecode Manipulation

Java bytecode manipulation is a good approach to get runtime-information of the program under test needed to create program spectra. Manipulating one `class` file usually takes about 300 - 500 ms (up to 1500 lines source code) with the used notebook. Besides, the manipulation runtime could be decreased by using another Java bytecode manipulation tool, which takes already Java bytecode and does not need to compile source code before inserting it.

However, there is also a problem which occurs, when the execution trace is written into a file with the help of Java bytecode manipulation. The file size gets rather fast extremely big.

6.3. Future Work

Analyzing Java bytecode enables to analyze all programming languages, which can be transformed into Java bytecode. In the future one could thus treat not only programs written in Java, but also programs written in other programming languages. Besides, another Java bytecode manipulation tool can be used to decrease the runtime of the manipulation process. Furthermore, one can think about more efficient ways to save the execution trace.

Future work could include extending this approach. Spectrum-based debugging can be combined with model-based techniques. To ensure that most of the source code is covered with test cases, the test cases could be generated automatically.

A. TCAS

A.1. Diagrams

This section shows some diagrams, which illustrate the probabilities of being faulty for each statement. Only versions which have a single fault are considered. The x-axis represents the line (statement) numbers of the source code and the y-axis represents the possibility. The possibility value is between 0 and 1.

It can be seen, that the Tarantula coefficient results in the highest possibility values and the Jaccard coefficient in the lowest. The Ochiai coefficient lies in the middle of both.

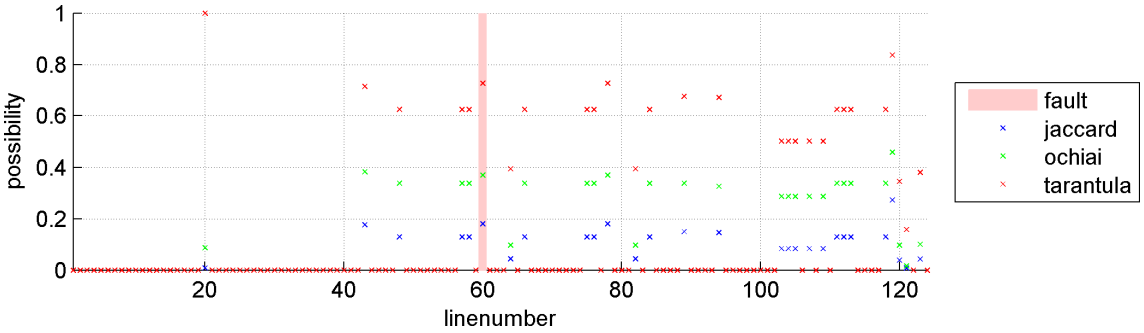

Figure A.1.: The possibility of being faulty for each statement of version 01.

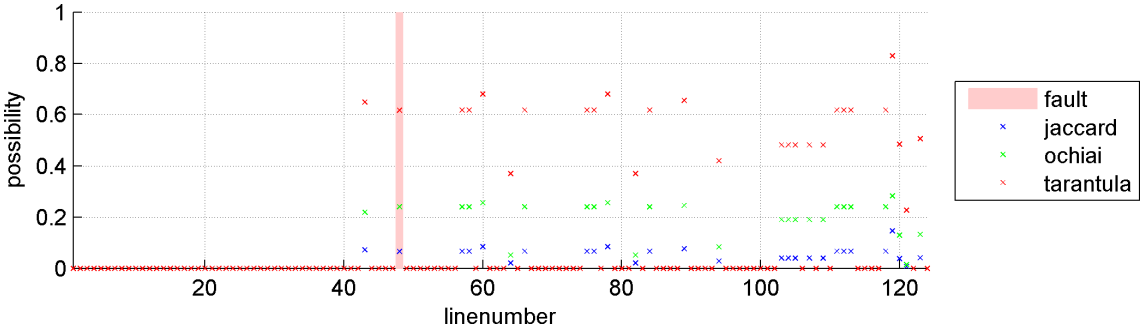

Figure A.2.: The possibility of being faulty for each statement of version 02.

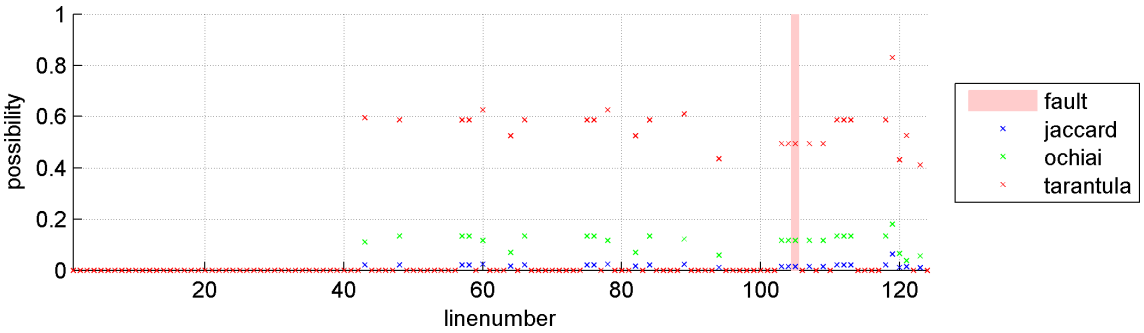

Figure A.3.: The possibility of being faulty for each statement of version 03.

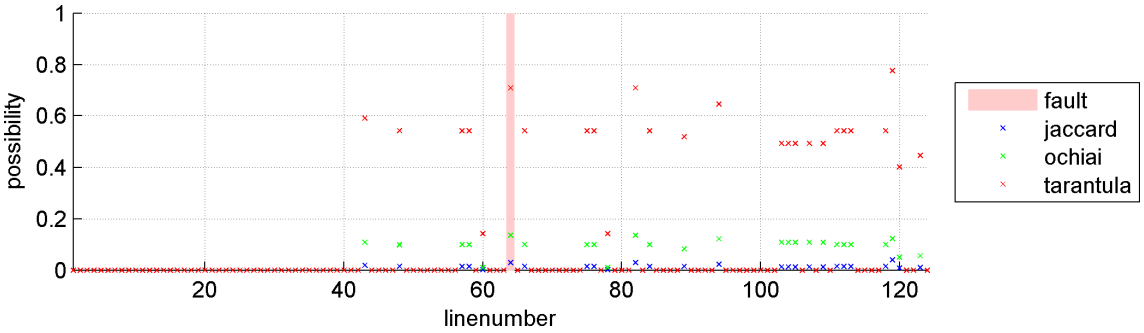

Figure A.4.: The possibility of being faulty for each statement of version 04.

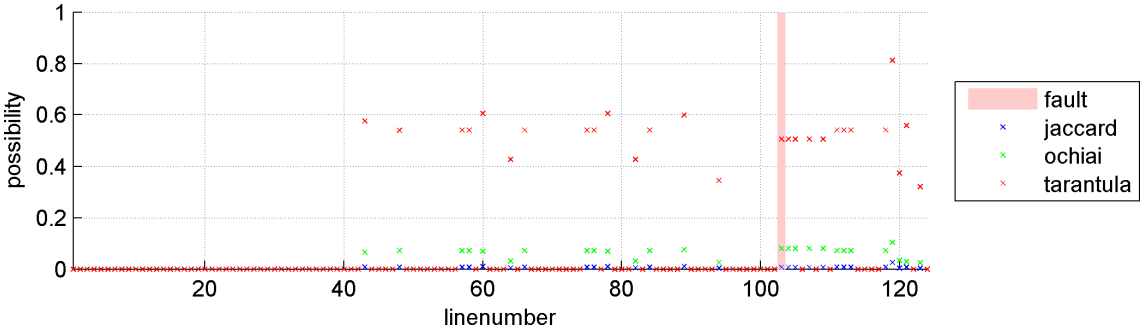

Figure A.5.: The possibility of being faulty for each statement of version 05.

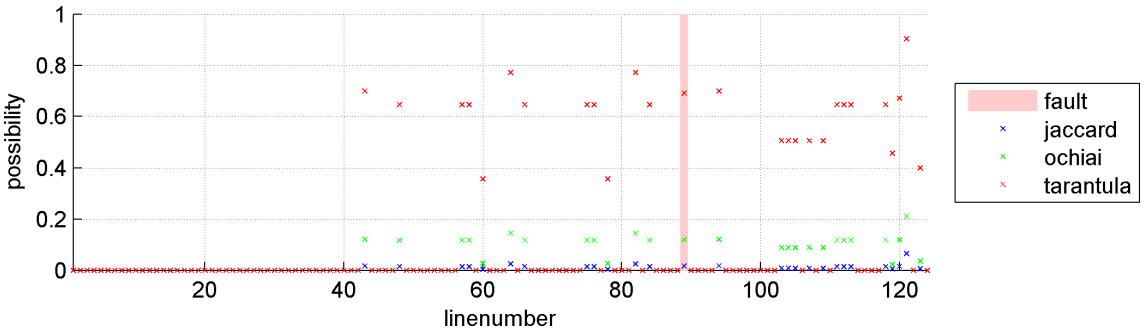

Figure A.6.: The possibility of being faulty for each statement of version 06.

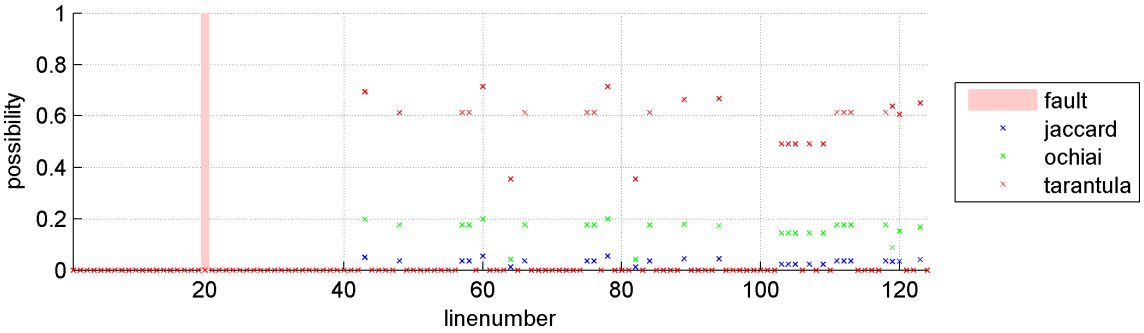

Figure A.7.: The possibility of being faulty for each statement of version 07.

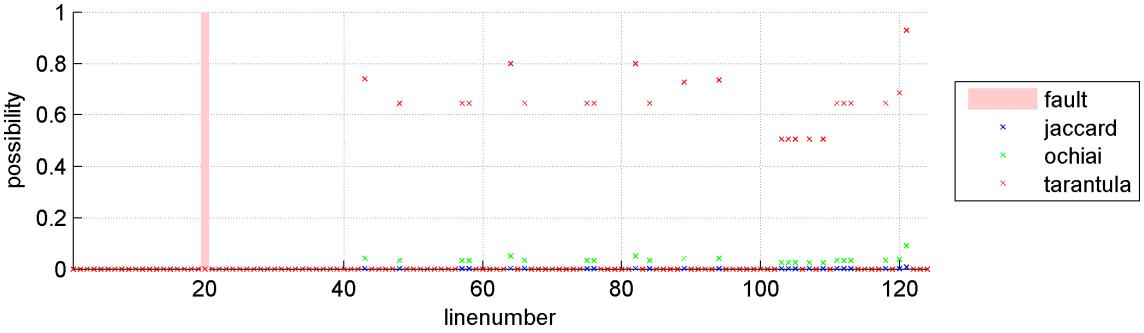

Figure A.8.: The possibility of being faulty for each statement of version 08.

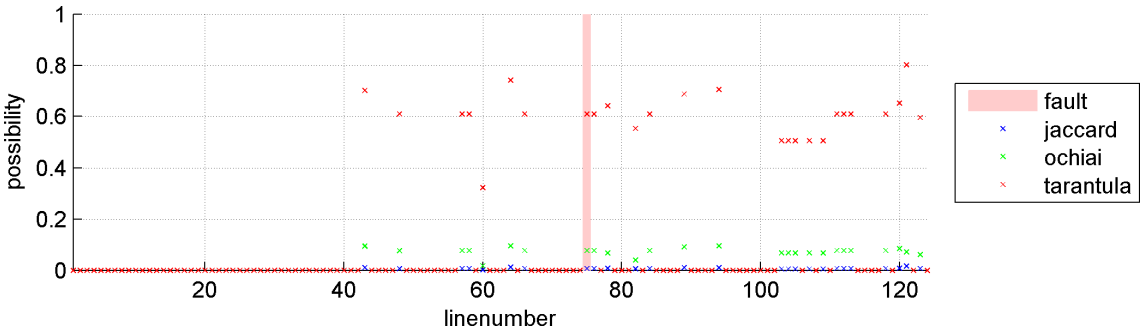

Figure A.9.: The possibility of being faulty for each statement of version 09.

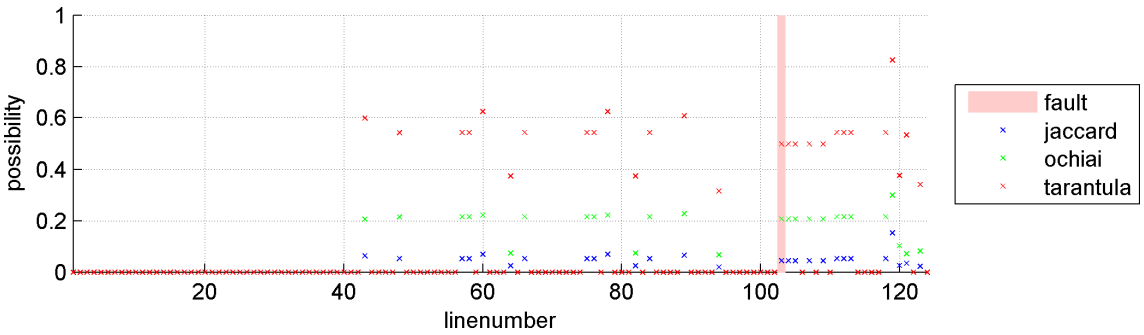

Figure A.10.: The possibility of being faulty for each statement of version 12.

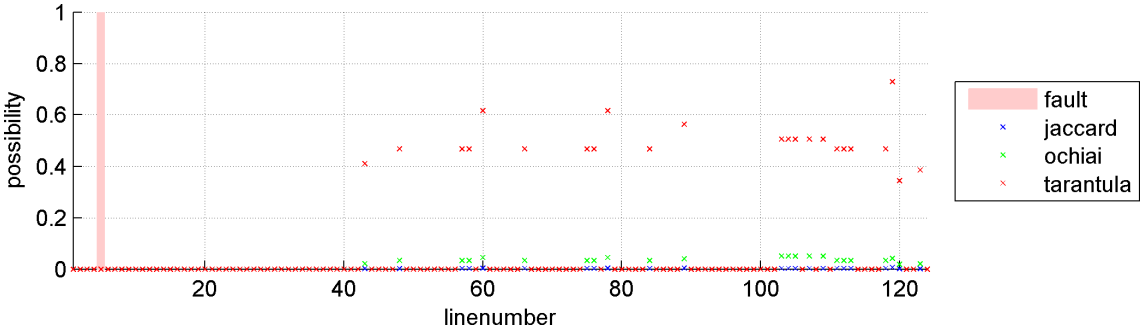

Figure A.11.: The possibility of being faulty for each statement of version 13.

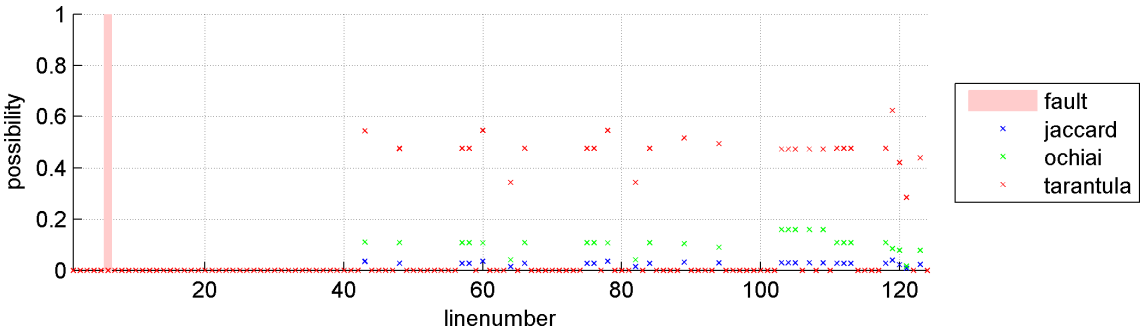

Figure A.12.: The possibility of being faulty for each statement of version 14.

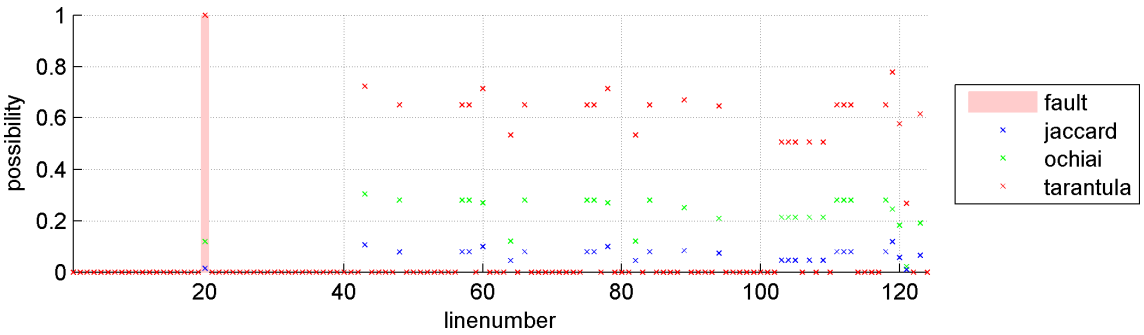

Figure A.13.: The possibility of being faulty for each statement of version 16.

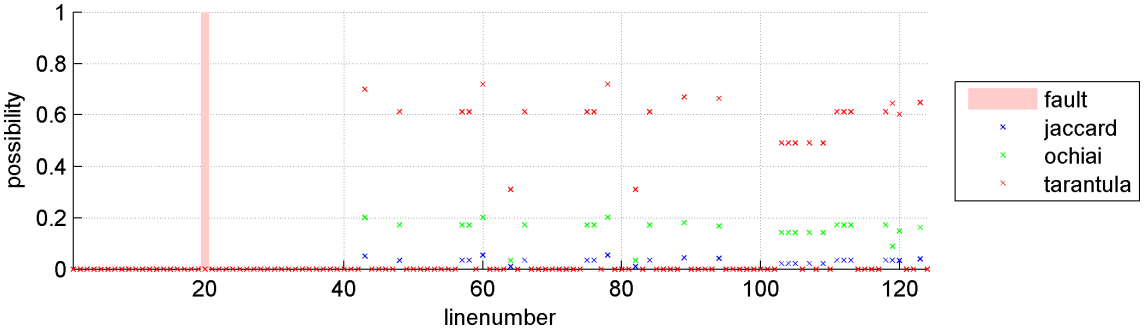

Figure A.14.: The possibility of being faulty for each statement of version 17.

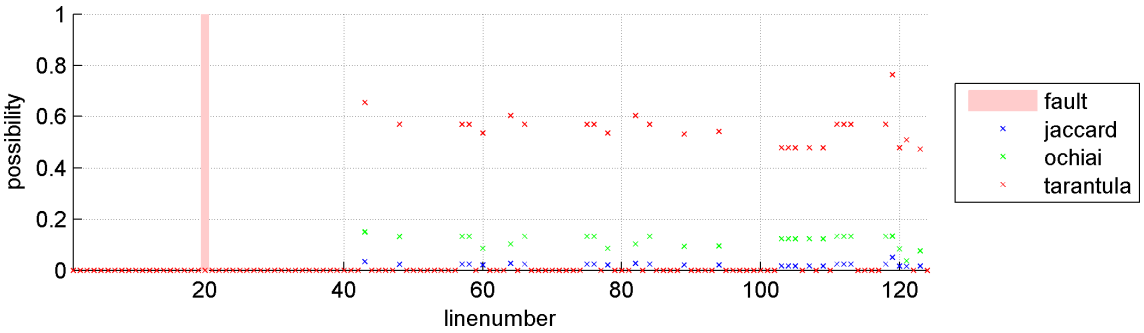

Figure A.15.: The possibility of being faulty for each statement of version 18.

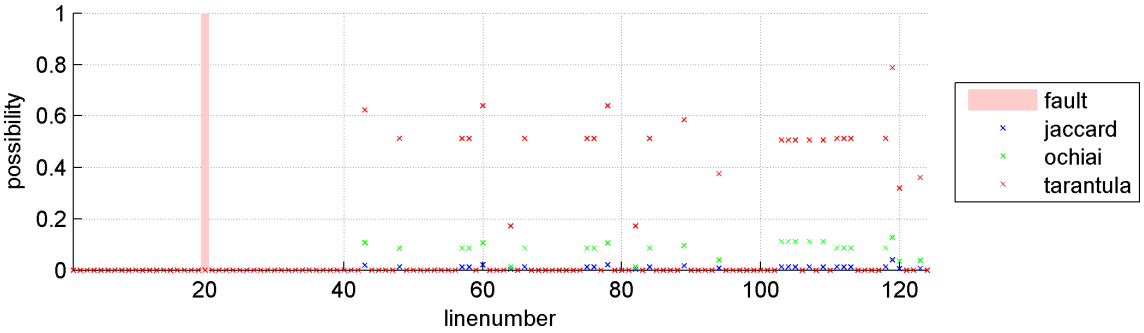

Figure A.16.: The possibility of being faulty for each statement of version 19.

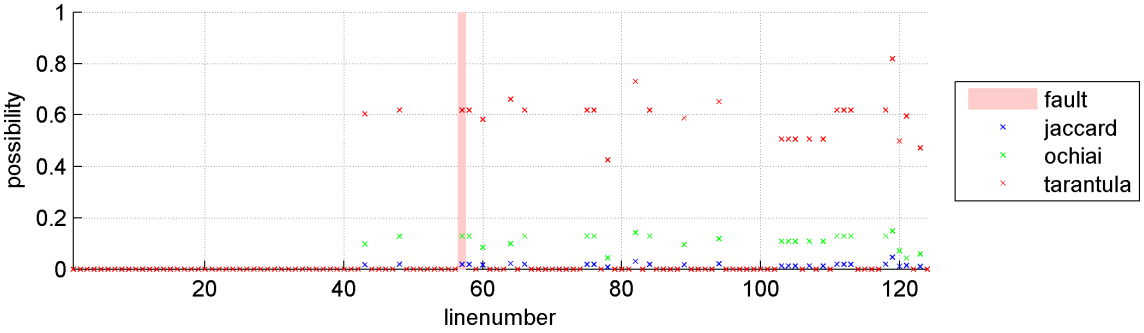

Figure A.17.: The possibility of being faulty for each statement of version 20.

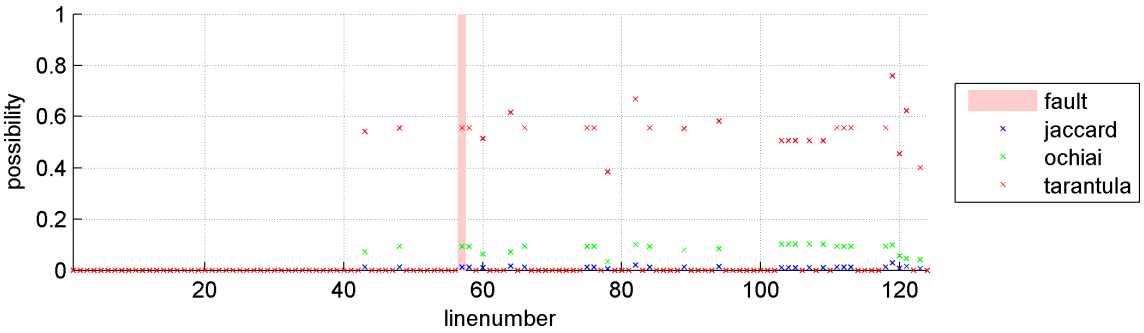

Figure A.18.: The possibility of being faulty for each statement of version 21.

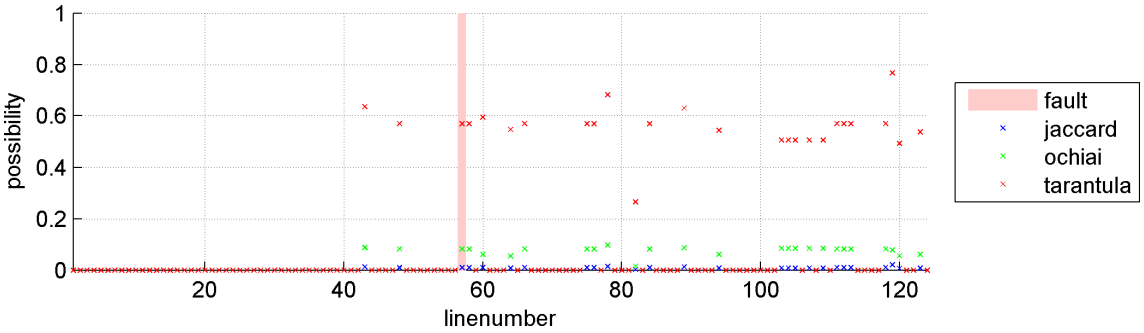

Figure A.19.: The possibility of being faulty for each statement of version 22.

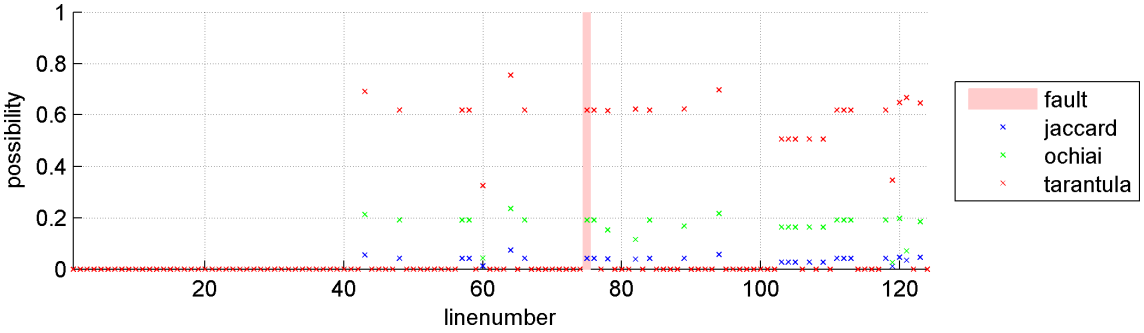

Figure A.20.: The possibility of being faulty for each statement of version 23.

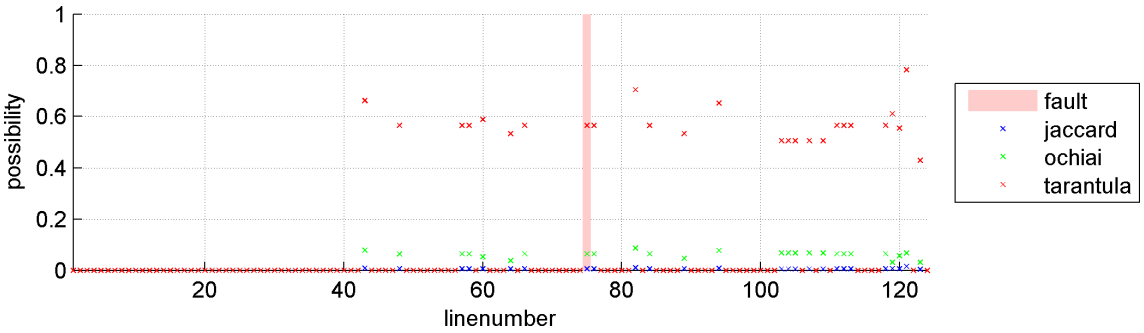

Figure A.21.: The possibility of being faulty for each statement of version 24.

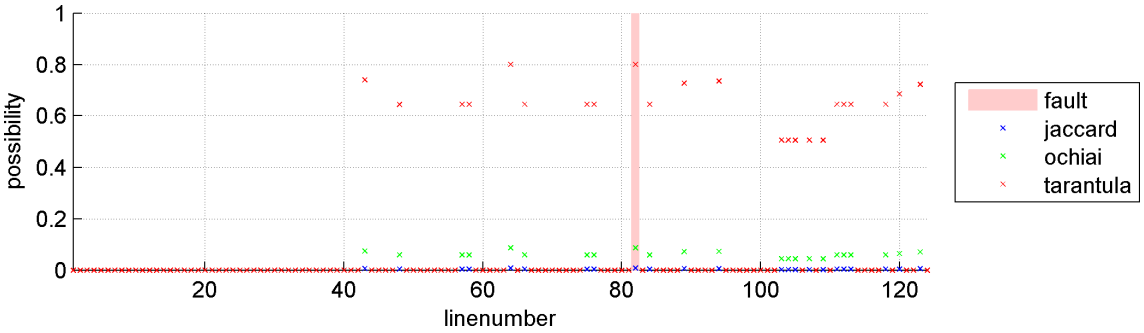

Figure A.22.: The possibility of being faulty for each statement of version 25.

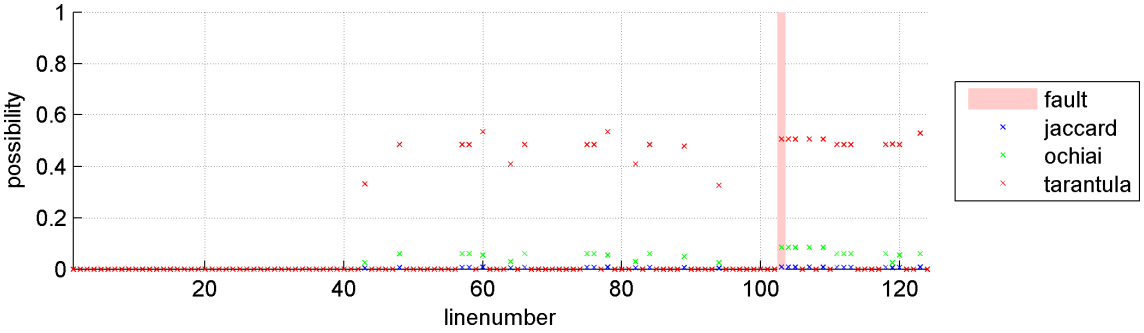

Figure A.23.: The possibility of being faulty for each statement of version 26.

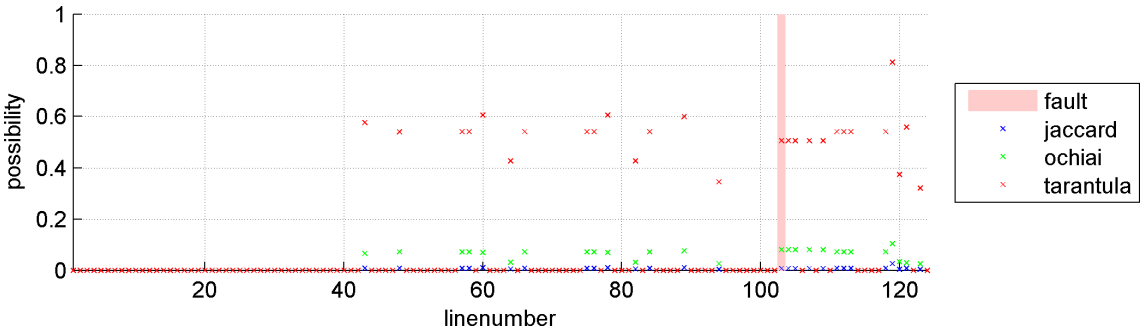

Figure A.24.: The possibility of being faulty for each statement of version 27.

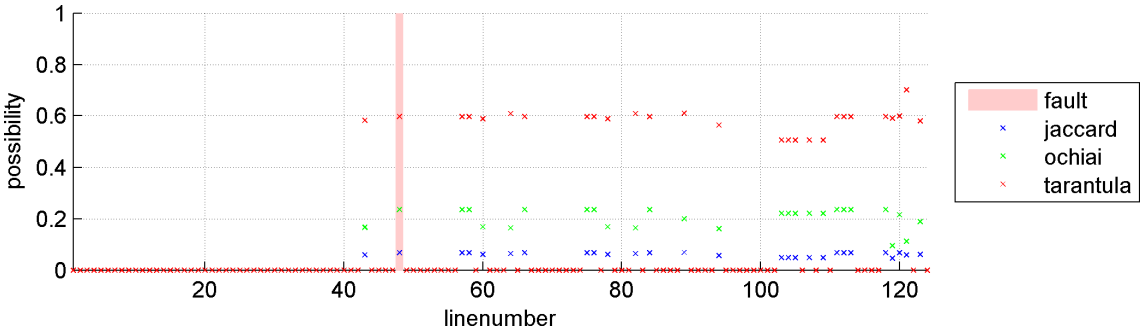

Figure A.25.: The possibility of being faulty for each statement of version 28.

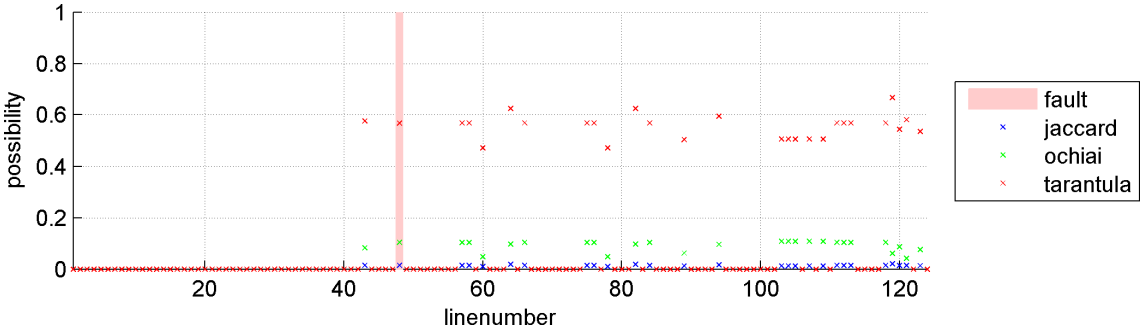

Figure A.26.: The possibility of being faulty for each statement of version 29.

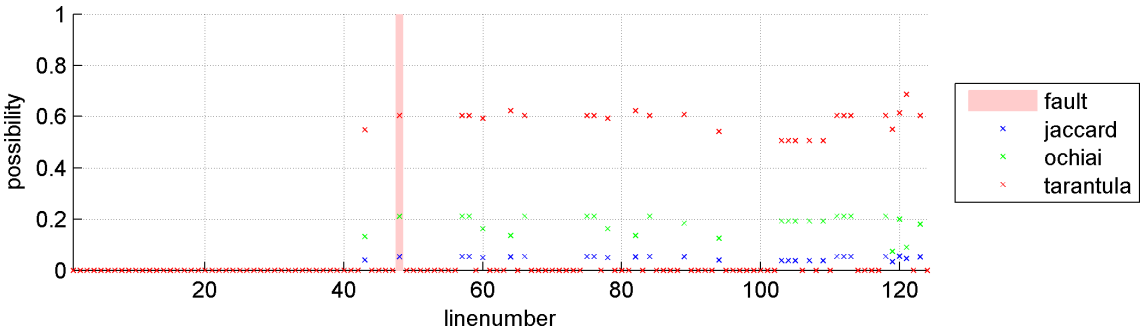

Figure A.27.: The possibility of being faulty for each statement of version 30.

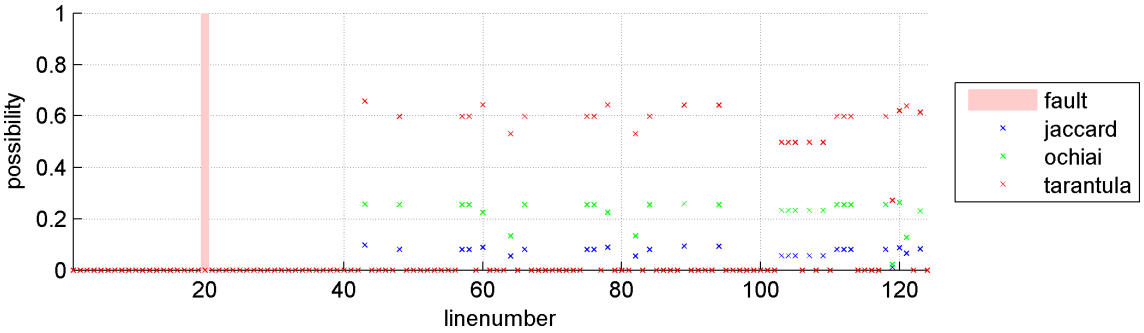

Figure A.28.: The possibility of being faulty for each statement of version 33.

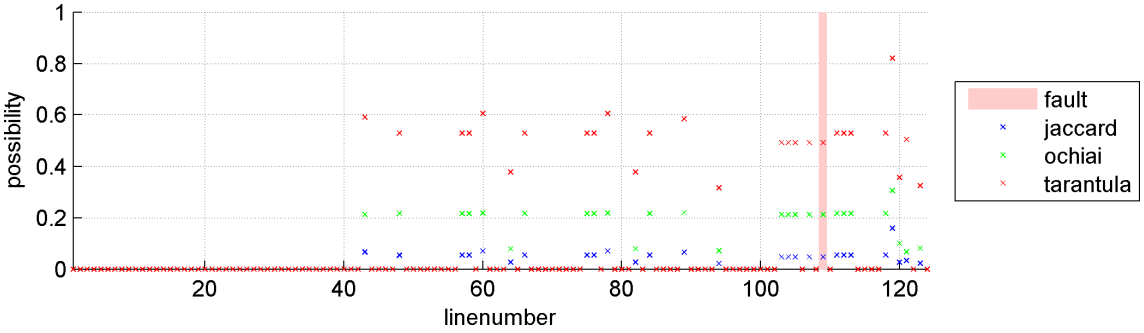

Figure A.29.: The possibility of being faulty for each statement of version 34.

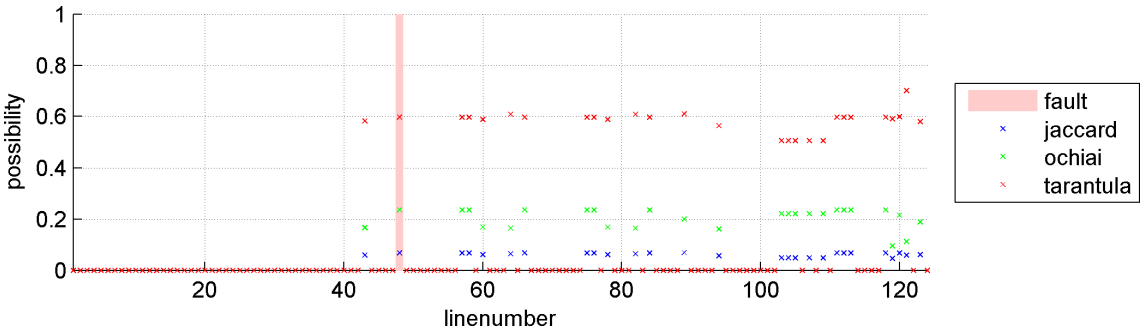

Figure A.30.: The possibility of being faulty for each statement of version 35.

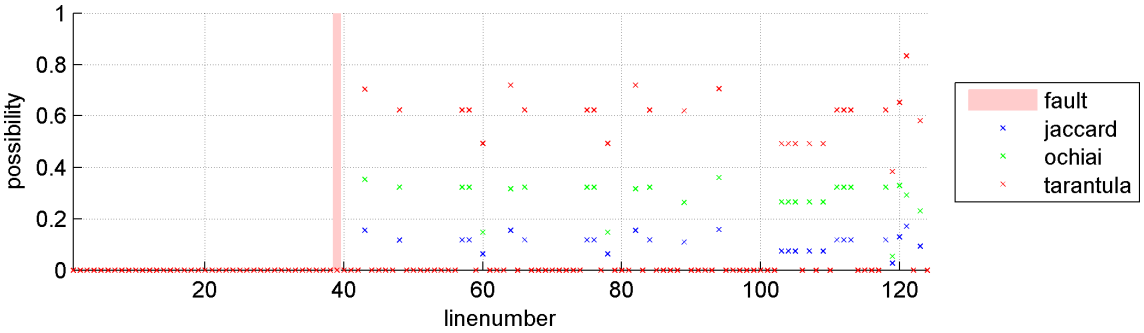

Figure A.31.: The possibility of being faulty for each statement of version 36.

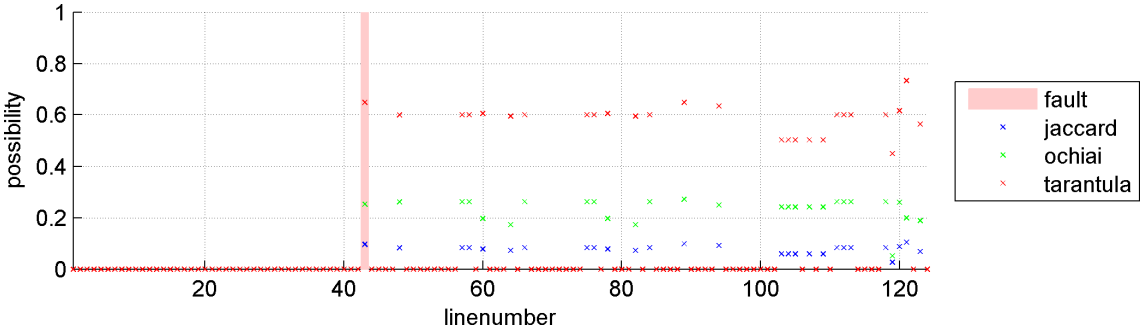

Figure A.32.: The possibility of being faulty for each statement of version 37.

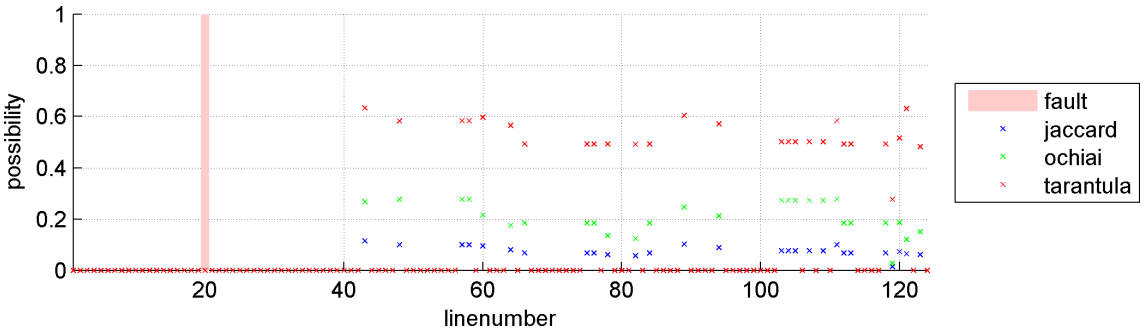

Figure A.33.: The possibility of being faulty for each statement of version 38.

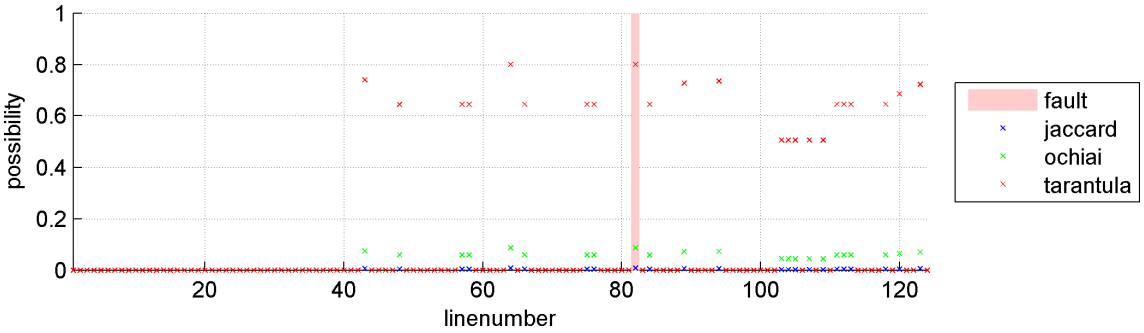

Figure A.34.: The possibility of being faulty for each statement of version 39.

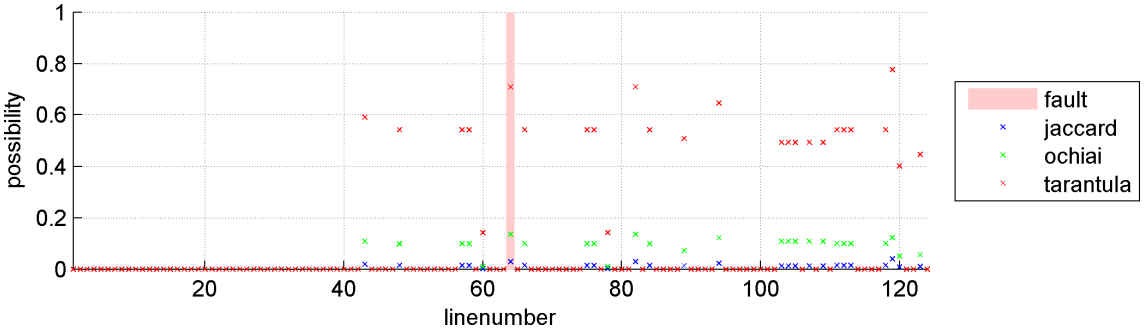

Figure A.35.: The possibility of being faulty for each statement of version 41.

A.2. Source

A.2.1. Source Code

This section supplies the original source code of TCAS. The Java implementation of TCAS is introduced in [26]. The uninitialized member variables are set for each test case by the test runner.

```
class tcas {

    public static final int OLEV       = 600; /* in feets/minute */
    public static final int MAXALTDIFF = 600; /* max altitude difference in feet */
    public static final int MINSEP     = 300;       /* min separation in feet */
    public static final int NOZCROSS   = 100; /* in feet */


    static int Cur_Vertical_Sep;
    static boolean High_Confidence;
    static boolean Two_of_Three_Reports_Valid;

    static int Own_Tracked_Alt;
    static int Own_Tracked_Alt_Rate;
    static int Other_Tracked_Alt;

    static int Alt_Layer_Value; /* 0, 1, 2, 3 */
    static int Positive_RA_Alt_Thresh[] = {400,500,640,740};

    static int Up_Separation;
    static int Down_Separation;

    /* state variables */
    static int Other_RAC; /* NO_INTENT, DO_NOT_CLIMB, DO_NOT_DESCEND */
    public static final int NO_INTENT = 0;
    public static final int DO_NOT_CLIMB = 1;
    public static final int DO_NOT_DESCEND = 2;

    static int Other_Capability; /* TCAS_TA, OTHER */
    public static final int TCAS_TA = 1;
    public static final int OTHER = 2;

    static boolean Climb_Inhibit; /* true/false */

    public static final int UNRESOLVED = 0;
    public static final int UPWARD_RA = 1;
    public static final int DOWNWARD_RA = 2;

    static int ALIM ()
    {
        return Positive_RA_Alt_Thresh[Alt_Layer_Value];
    }

    static int Inhibit_Biased_Climb ()
    {
        return (Climb_Inhibit ? Up_Separation + NOZCROSS : Up_Separation);
    }

    static boolean Non_Crossing_Biased_Climb()
    {
        boolean upward_preferred;
        int upward_crossing_situation;
        boolean result;

        upward_preferred = Inhibit_Biased_Climb() > Down_Separation;
        if (upward_preferred)
            {
                result = !(Own_Below_Threat()) || ((Own_Below_Threat()) && (!(Down_Separation >= ALIM())));
            }
        else
            {
                result = Own_Above_Threat() && (Cur_Vertical_Sep >= MINSEP) && (Up_Separation >= ALIM());
            }
        return result;
    }

    static boolean Non_Crossing_Biased_Descend()
    {
        boolean upward_preferred;
        int upward_crossing_situation;
        boolean result;

        upward_preferred = Inhibit_Biased_Climb() > Down_Separation;
        if (upward_preferred)
```

```
            {
                result = Own_Below_Threat() && (Cur_Vertical_Sep >= MINSEP) && (Down_Separation >= ALIM());
            }
        else
            {
                result = !(Own_Above_Threat()) || ((Own_Above_Threat()) && (Up_Separation >= ALIM()));
            }
        return result;
    }

    static boolean Own_Below_Threat()
    {
        return (Own_Tracked_Alt < Other_Tracked_Alt);
    }

    static boolean Own_Above_Threat()
    {
        return (Other_Tracked_Alt < Own_Tracked_Alt);
    }

    static int alt_sep_test()
    {
        boolean enabled, tcas_equipped, intent_not_known;
        boolean need_upward_RA, need_downward_RA;
        int alt_sep;

        enabled = High_Confidence && (Own_Tracked_Alt_Rate <= OLEV) && (Cur_Vertical_Sep > MAXALTDIFF);
        tcas_equipped = Other_Capability == TCAS_TA;
        intent_not_known = Two_of_Three_Reports_Valid && Other_RAC == NO_INTENT;

        alt_sep = UNRESOLVED;

        if (enabled && ((tcas_equipped && intent_not_known) || !tcas_equipped))
            {
                need_upward_RA = Non_Crossing_Biased_Climb() && Own_Below_Threat();
                need_downward_RA = Non_Crossing_Biased_Descend() && Own_Above_Threat();
                if (need_upward_RA && need_downward_RA)
                    /* unreachable: requires Own_Below_Threat and Own_Above_Threat
                       to both be true - that requires Own_Tracked_Alt < Other_Tracked_Alt
                       and Other_Tracked_Alt < Own_Tracked_Alt, which isn't possible */
                    alt_sep = UNRESOLVED;
                else if (need_upward_RA)
                    alt_sep = UPWARD_RA;
                else if (need_downward_RA)
                    alt_sep = DOWNWARD_RA;
                else
                    alt_sep = UNRESOLVED;
            }
        return alt_sep;
    }
}
```

A.2.2. Injected Faults

To show which faults have been injected to the different versions the difference file is inserted below.

```
1c1
< class tcas {
---
> class tcas_v01 {
58c58
<                 result = !(Own_Below_Threat()) || ((Own_Below_Threat()) && (!(Down_Separation >= ALIM())));
---
>                 result = !(Own_Below_Threat()) || ((Own_Below_Threat()) && (!(Down_Separation > ALIM())));
1c1
< class tcas {
---
> class tcas_v02 {
46c46
<         return (Climb_Inhibit ? Up_Separation + NOZCROSS : Up_Separation);
---
>         return (Climb_Inhibit ? Up_Separation + MINSEP : Up_Separation);
1c1
< class tcas {
---
> class tcas_v03 {
103c103
<         intent_not_known = Two_of_Three_Reports_Valid && Other_RAC == NO_INTENT;
---
>         intent_not_known = Two_of_Three_Reports_Valid || Other_RAC == NO_INTENT;
1c1
< class tcas {
```

```
---
> class tcas_v04 {
62c62
<                 result = Own_Above_Threat() && (Cur_Vertical_Sep >= MINSEP) && (Up_Separation >= ALIM());
---
>                 result = Own_Above_Threat() && (Cur_Vertical_Sep >= MINSEP) || (Up_Separation >= ALIM());
1c1
< class tcas {
---
> class tcas_v05 {
101c101
<         enabled = High_Confidence && (Own_Tracked_Alt_Rate <= OLEV) && (Cur_Vertical_Sep > MAXALTDIFF);
---
>         enabled = High_Confidence && (Own_Tracked_Alt_Rate <= OLEV);
1c1
< class tcas {
---
> class tcas_v06 {
87c87
<         return (Own_Tracked_Alt < Other_Tracked_Alt);
---
>         return (Own_Tracked_Alt <= Other_Tracked_Alt);
1c1
< class tcas {
---
> class tcas_v07 {
18c18
<     static int Positive_RA_Alt_Thresh[] = {400,500,640,740};
---
>     static int Positive_RA_Alt_Thresh[] = {400,550,640,740};
1c1
< class tcas {
---
> class tcas_v08 {
18c18
<     static int Positive_RA_Alt_Thresh[] = {400,500,640,740};
---
>     static int Positive_RA_Alt_Thresh[] = {400,500,640,700};
1c1
< class tcas {
---
> class tcas_v09 {
73c73
<         upward_preferred = Inhibit_Biased_Climb() > Down_Separation;
---
>         upward_preferred = Inhibit_Biased_Climb() >= Down_Separation;
1c1
< class tcas {
---
> class tcas_v10 {
87c87
<         return (Own_Tracked_Alt < Other_Tracked_Alt);
---
>         return (Own_Tracked_Alt <= Other_Tracked_Alt);
92c92
<         return (Other_Tracked_Alt < Own_Tracked_Alt);
---
>         return (Other_Tracked_Alt <= Own_Tracked_Alt);
1c1
< class tcas {
---
> class tcas_v11 {
87c87
<         return (Own_Tracked_Alt < Other_Tracked_Alt);
---
>         return (Own_Tracked_Alt <= Other_Tracked_Alt);
92c92
<         return (Other_Tracked_Alt < Own_Tracked_Alt);
---
>         return (Other_Tracked_Alt <= Own_Tracked_Alt);
101c101
<         enabled = High_Confidence && (Own_Tracked_Alt_Rate <= OLEV) && (Cur_Vertical_Sep > MAXALTDIFF);
---
>         enabled = High_Confidence || (Own_Tracked_Alt_Rate <= OLEV) && (Cur_Vertical_Sep > MAXALTDIFF);
1c1
< class tcas {
---
> class tcas_v12 {
101c101
<         enabled = High_Confidence && (Own_Tracked_Alt_Rate <= OLEV) && (Cur_Vertical_Sep > MAXALTDIFF);
---
>         enabled = High_Confidence || (Own_Tracked_Alt_Rate <= OLEV) && (Cur_Vertical_Sep > MAXALTDIFF);
1c1
< class tcas {
---
> class tcas_v13 {
3c3
<     public static final int OLEV       = 600; /* in feets/minute */
```

```
---
>     public static final int OLEV       = 600+100; /* in feets/minute */
1c1
< class tcas {
---
> class tcas_v14 {
4c4
<     public static final int MAXALTDIFF = 600; /* max altitude difference in feet */
---
>     public static final int MAXALTDIFF = 600+50; /* max altitude difference in feet */
1c1
< class tcas {
---
> class tcas_v15 {
5c5
<     public static final int MINSEP     = 300;       /* min separation in feet */
---
>     public static final int MINSEP     = 300+350;       /* min separation in feet */
101c101
<         enabled = High_Confidence && (Own_Tracked_Alt_Rate <= OLEV) && (Cur_Vertical_Sep > MAXALTDIFF);
---
>         enabled = High_Confidence && (Own_Tracked_Alt_Rate <= OLEV);
1c1
< class tcas {
---
> class tcas_v16 {
18c18
<     static int Positive_RA_Alt_Thresh[] = {400,500,640,740};
---
>     static int Positive_RA_Alt_Thresh[] = {400+1,500,640,740};
1c1
< class tcas {
---
> class tcas_v17 {
18c18
<     static int Positive_RA_Alt_Thresh[] = {400,500,640,740};
---
>     static int Positive_RA_Alt_Thresh[] = {400,500+1,640,740};
1c1
< class tcas {
---
> class tcas_v18 {
18c18
<     static int Positive_RA_Alt_Thresh[] = {400,500,640,740};
---
>     static int Positive_RA_Alt_Thresh[] = {400,500,640+50,740};
1c1
< class tcas {
---
> class tcas_v19 {
18c18
<     static int Positive_RA_Alt_Thresh[] = {400,500,640,740};
---
>     static int Positive_RA_Alt_Thresh[] = {400,500,640,740+20};
1c1
< class tcas {
---
> class tcas_v20 {
55c55
<         upward_preferred = Inhibit_Biased_Climb() > Down_Separation;
---
>         upward_preferred = Inhibit_Biased_Climb() >= Down_Separation;
1c1
< class tcas {
---
> class tcas_v21 {
55c55
<         upward_preferred = Inhibit_Biased_Climb() > Down_Separation;
---
>         upward_preferred = (Up_Separation + NOZCROSS) > Down_Separation;
1c1
< class tcas {
---
> class tcas_v22 {
55c55
<         upward_preferred = Inhibit_Biased_Climb() > Down_Separation;
---
>         upward_preferred = Up_Separation > Down_Separation;
1c1
< class tcas {
---
> class tcas_v23 {
73c73
<         upward_preferred = Inhibit_Biased_Climb() > Down_Separation;
---
>         upward_preferred = (Up_Separation + NOZCROSS) > Down_Separation;
1c1
< class tcas {
```

```
---
> class tcas_v24 {
73c73
<         upward_preferred = Inhibit_Biased_Climb() > Down_Separation;
---
>         upward_preferred = Up_Separation > Down_Separation;
1c1
< class tcas {
---
> class tcas_v25 {
80c80
<                 result = !(Own_Above_Threat()) || ((Own_Above_Threat()) && (Up_Separation >= ALIM()));
---
>                 result = !(Own_Above_Threat()) || ((Own_Above_Threat()) && (Up_Separation > ALIM()));
1c1
< class tcas {
---
> class tcas_v26 {
101c101
<         enabled = High_Confidence && (Own_Tracked_Alt_Rate <= OLEV) && (Cur_Vertical_Sep > MAXALTDIFF);
---
>         enabled = High_Confidence && (Cur_Vertical_Sep > MAXALTDIFF);
1c1
< class tcas {
---
> class tcas_v27 {
101c101
<         enabled = High_Confidence && (Own_Tracked_Alt_Rate <= OLEV) && (Cur_Vertical_Sep > MAXALTDIFF);
---
>         enabled = High_Confidence && (Own_Tracked_Alt_Rate <= OLEV);
1c1
< class tcas {
---
> class tcas_v28 {
46c46
<         return (Climb_Inhibit ? Up_Separation + NOZCROSS : Up_Separation);
---
>         return ((Climb_Inhibit == false) ? Up_Separation + NOZCROSS : Up_Separation);
1c1
< class tcas {
---
> class tcas_v29 {
46c46
<         return (Climb_Inhibit ? Up_Separation + NOZCROSS : Up_Separation);
---
>         return (Up_Separation);
1c1
< class tcas {
---
> class tcas_v30 {
46c46
<         return (Climb_Inhibit ? Up_Separation + NOZCROSS : Up_Separation);
---
>         return Up_Separation + NOZCROSS;
1c1
< class tcas {
---
> class tcas_v31 {
58a59
>                 result = result && (Own_Tracked_Alt <= Other_Tracked_Alt);
62a64
>                 result = result && (Own_Tracked_Alt < Other_Tracked_Alt);
109c111
<                 need_upward_RA = Non_Crossing_Biased_Climb() && Own_Below_Threat();
---
>                 need_upward_RA = Non_Crossing_Biased_Climb();
1c1
< class tcas {
---
> class tcas_v32 {
76a77
>                 result = result & (Other_Tracked_Alt < Own_Tracked_Alt);
80a82
>                 result = result & (Other_Tracked_Alt <= Own_Tracked_Alt);
110c112
<                 need_downward_RA = Non_Crossing_Biased_Descend() && Own_Above_Threat();
---
>                 need_downward_RA = Non_Crossing_Biased_Descend();
1c1
< class tcas {
---
> class tcas_v33 {
18c18
<     static int Positive_RA_Alt_Thresh[] = {400,500,640,740};
---
>     static int Positive_RA_Alt_Thresh[] = {0,400,500,640,740}; /* not quite equivalent to C code fault */
1c1
< class tcas {
```

```
---
> class tcas_v34 {
107c107
<         if (enabled && ((tcas_equipped && intent_not_known) || !tcas_equipped))
---
>         if (enabled && tcas_equipped && intent_not_known || !tcas_equipped)
1c1
< class tcas {
---
> class tcas_v35 {
46c46
<         return (Climb_Inhibit ? Up_Separation + NOZCROSS : Up_Separation);
---
>         return (Climb_Inhibit ? Up_Separation : Up_Separation + NOZCROSS);
1c1
< class tcas {
---
> class tcas_v36 {
37c37
<     public static final int DOWNWARD_RA = 2;
---
>     public static final int DOWNWARD_RA = 1;
1c1
< class tcas {
---
> class tcas_v37 {
41c41
<         return Positive_RA_Alt_Thresh[Alt_Layer_Value];
---
>         return Positive_RA_Alt_Thresh[0];
1c1
< class tcas {
---
> class tcas_v38 {
18c18
<     static int Positive_RA_Alt_Thresh[] = {400,500,640,740};
---
>     static int Positive_RA_Alt_Thresh[] = {400,500,640}; /* not quite equivalent to C fault */
1c1
< class tcas {
---
> class tcas_v39 {
80c80
<                 result = !(Own_Above_Threat()) || ((Own_Above_Threat()) && (Up_Separation >= ALIM()));
---
>                 result = !(Own_Above_Threat()) || ((Own_Above_Threat()) && (Up_Separation > ALIM()));
1c1
< class tcas {
---
> class tcas_v40 {
58c58
<                 result = !(Own_Below_Threat()) || ((Own_Below_Threat()) && (!(Down_Separation >= ALIM())));
---
>                 result = ((Own_Below_Threat()) && (!(Down_Separation >= ALIM())));
109c109
<                 need_upward_RA = Non_Crossing_Biased_Climb() && Own_Below_Threat();
---
>                 need_upward_RA = Non_Crossing_Biased_Climb();
1c1
< class tcas {
---
> class tcas_v41 {
62c62
<                 result = Own_Above_Threat() && (Cur_Vertical_Sep >= MINSEP) && (Up_Separation >= ALIM());
---
>                 result = (Cur_Vertical_Sep >= MINSEP) && (Up_Separation >= ALIM());
```

B. Abbreviations

API	Application Programming Interface
JVM	Java Virtual Machine
LoC	Lines of Code
MBD	Model-based Diagnosis
SFL	Spectrum-based Fault Localization
UML	Unified Modeling Language

Bibliography

[1] Rui Abreu, Wolfgang Mayer, Markus Stumptner, and Arjan J. C. van Gemund. Refining spectrum-based fault localization rankings. In *SAC '09: Proceedings of the 2009 ACM symposium on Applied Computing*, pages 409–414, New York, NY, USA, 2009. ACM.

[2] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. An evaluation of similarity coefficients for software fault localization. In *PRDC '06: Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing*, pages 39–46, Washington, DC, USA, 2006. IEEE Computer Society.

[3] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. Spectrum-based multiple fault localization. In *ASE '09: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 88–99, Washington, DC, USA, 2009. IEEE Computer Society.

[4] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan J. C. van Gemund. A practical evaluation of spectrum-based fault localization. *J. Syst. Softw.*, 82(11):1780–1792, 2009.

[5] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. On the accuracy of spectrum-based fault localization. In *TAICPART-MUTATION '07: Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, pages 89–98, Washington, DC, USA, 2007. IEEE Computer Society.

[6] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost analysis of java bytecode. In *ESOP'07: Proceedings of the 16th European conference on Programming*, pages 157–172, Berlin, Heidelberg, 2007. Springer-Verlag.

[7] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *Dependable and Secure Computing, IEEE Transactions on*, 1(1):11 – 33, jan.-march 2004.

[8] Kent Beck. Junit api document. kentbeck.github.com/junit/javadoc/4.8/.

[9] Eric Bruneton. Asm 3.0 a java bytecode engineering library. download.forge.objectweb.org/asm/asm-guide.pdf.

[10] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. Asm: a code manipulation tool to implement adaptable systems, November 2002. Grenoble, France.

[11] Mike Y. Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 595–604, Washington, DC, USA, 2002. IEEE Computer Society.

[12] Shigeru Chiba. Javassist java bytecode manipulation made simple. www.jboss.org/javassist.

[13] Shigeru Chiba. Javassist javadocs. http://www.csg.is.titech.ac.jp/∼ chiba/-javassist/html.

[14] Shigeru Chiba and Muga Nishizawa. An easy-to-use toolkit for efficient java bytecode translators. In *GPCE '03: Proceedings of the 2nd international conference on Generative programming and component engineering*, pages 364–376, New York, NY, USA, 2003. Springer-Verlag New York, Inc.

[15] Valentin Dallmeier, Christian Lindig, and Andreas Zeller. Lightweight defect localization for java. In Andrew P. Black, editor, *ECOOP 2005 - Object-Oriented Programming*, volume 3586 of *Lecture Notes in Computer Science*, pages 528–550. Springer Berlin / Heidelberg, 2005.

[16] Apache Software Foundation. Bcel api. jakarta.apache.org/bcel/manual.html.

[17] Mary Jean Harrold, Gregg Rothermel, Rui Wu, and Liu Yi. An empirical investigation of program spectra. In *PASTE '98: Proceedings of the 1998 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 83–90, New York, NY, USA, 1998. ACM.

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

[19] James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 467–477, New York, NY, USA, 2002. ACM.

[20] Don Lance, Roland H. Untch, and Nancy J. Wahl. Bytecode-based java program analysis. In *ACM-SE 37: Proceedings of the 37th annual Southeast regional conference (CD-ROM)*, page 14, New York, NY, USA, 1999. ACM.

[21] Tim Lindholm and Frank Yellin. The java virtual machine specification second edition. http://java.sun.com/docs/books/jvms/second_edition/html /VMSpecTOC.doc.html, 1999.

[22] Tim Lindholm and Frank Yellin. Vm spec - compiling for the java virtual machine. http://java.sun.com/docs/books/jvms/second_edition/html/ Compiling.doc.html, 1999.

[23] Tim Lindholm and Frank Yellin. Vm spec - introduction. http://java.sun.com/docs/books/jvms/second_edition/html/Introduction. doc.html, 1999.

[24] Tim Lindholm and Frank Yellin. Vm spec - the class file format. http://java.sun.com/docs/books/jvms/second_edition/html/Class File.doc.html, 1999.

[25] Tim Lindholm and Frank Yellin. Vm spec - the structure of the java virtual machine. http://java.sun.com/docs/books/jvms/second_edition/html /Overview.doc.html, 1999.

[26] Wolfgang Mayer. *Static and Hybrid Analysis in Model-based Debugging*. PhD thesis, School of Computer and Information Science, University of South Australia, 2007.

[27] Andreia da Silva Meyer, Antonio Augusto Franco Garcia, Anete Pereira de Souza, and Claudio Lopes de Souza Jr. Comparison of similarity coefficients used for cluster analysis with dominant markers in maize (Zea mays L). *Genetics and Molecular Biology*, 27:83 – 91, 00 2004.

[28] Attila Szegedi and Tibor Gyimothy. Dynamic slicing of java bytecode programs. In *SCAM '05: Proceedings of the Fifth IEEE International Workshop on Source Code Analysis and Manipulation*, pages 35–44, Washington, DC, USA, 2005. IEEE Computer Society.

[29] Martin Weiglhofer, Gordon Fraser, and Franz Wotawa. Using spectrum-based fault localization for test case grouping. In *ASE '09: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 630–634, Washington, DC, USA, 2009. IEEE Computer Society.

[30] X. Xie, W. E. Wong, T. Y. Chen, and B. Xu. Spetrum-based fault localization without test oracles. Technical Report UTDCS-07-10, Department of Computer Science, University of Texas at Dallas, February 2010.