Masterarbeit

Mutationsbasierte Fehlerkorrektur auf Basis von objektorientierten Sprachen

Martin Schmeisser, BSc

Graz, 2011

Institute for Software Technology
Graz University of Technology

Univ.-Prof. Dipl.-Ing. Dr. techn. Franz Wotawa
Dipl.-Ing. Birgit Hofer

Abstract

Some estimate that testing and correcting software takes up to 40 % of the total effort during software development. Thus it's not astonishing that people like to to detect, find and correct faults in software automatically. The focus in this work lies in automatically correcting software faults.

Hence the question we like to answer is: "Is it possible to automatically correct software, if we detect a fault in it". To come to the point, if I could answer that with a simple "Yes", I wouldn't have continued to write this text; instead I would have sold the idea and would be a very rich man by now.

We tackle the challenge to automatically correct software faults with the idea of the evolutionary theory that is implemented in genetic programming. To make it even possible to know that a software has faults, we relay on test cases that have to be available for the program. The program becomes mutated over and over and the test cases are executed on that mutated program. Depending on the positively executed test cases the program gets a rating and is stored in the population. Programs in this population also become mutated, rated and stored in the population repeatedly. If the population is getting to big, programs with bad ratings are removed. If one of this programs passes all test cases, it is a solution for the program and can be considered as a correction for the fault.

With this method we obtained for nearly 50 % of the tested programs, that actually produced failed test cases, solutions. During further evaluation we noticed that the results where very stable; if once a solution was found, it was possible to reproduce this over and over again.

Therefore it is quite possible to correct programs with faults with the approach of genetic programming. Also a lot of restrictions and limitations occurred that needs to be tackled in future works.

Kurzfassung

Schätzungen gehen davon aus, dass Testen und Korrigieren von Software 40 % der Gesamtkosten bei der Softwareentwicklung verursachen. Der Wunsch, diesen Prozess zu automatisieren und Fehler automatisch zu erkennen, zu finden und zu korrigieren, liegt daher sehr nahe. Auf dem automatischen Korrigieren liegt der Fokus dieser Arbeit.

Es stellt sich daher die Frage: “Ist es möglich eine Software automatisch zu korrigieren, wenn bei dieser Fehler erkannt worden sind?” Um es gleich vorweg zunehmen, könnte ich diese Frage mit einem einfachen “Ja” beantworten, würde ich diesen Text nicht schreiben, sondern hätte meine Idee verkauft und wäre ein sehr, sehr reicher Mann.

Um Fehler automatisch zu korrigieren, greifen wir auf die Idee der Evolutionstheorie zurück, die in der genetischen Programmierung umgesetzt worden ist. Um überhaupt erst feststellen zu können, dass das Programm fehlerhaft ist, wird auf die Testfälle zugegriffen, die für das Programm verfügbar sein müssen. Das Programm wird wiederkehrend mutiert und die Testfälle werden erneut ausgeführt. Abhängig von der Anzahl der positiv ausgeführten Testfälle bekommt das Programm eine Bewertung und wird in einer Population gespeichert. Programme aus dieser Population werden nun wiederkehrend mutiert, bewertet und in die Population eingefügt. Wird die Population zu groß, werden schlechtere Individuen aus ihr entfernt. Werden für ein Individuum alle Testfälle positiv ausgeführt, wurde eine Lösung gefunden, die eine Korrektur des Programms sein kann.

Mit dieser Methode konnten für knapp 50 % der getesteten fehlerhaften Programme, bei denen Fehler auch erkannt wurden, Lösungen gefunden werden. Im Zuge der Evaluierung stellte sich heraus, dass die Ergebnisse sehr stabil waren; wurde eine Lösung gefunden, so wurde sie auch bei wiederholten Evaluierungsläufen immer wieder sehr zuverlässig gefunden.

Es stellte sich heraus, dass fehlerhafte Programme mit dem gewählten Ansatz der genetischen Programmierung korrigiert werden können. Es wurden aber auch viele Einschränkungen und Limitierungen aufgedeckt, auf die in Zukunft eingegangen werden muss.

Danksagung

Am Ende und doch wieder am Anfang.

Als ich mit der Master Arbeit begann, dachte ich mir: Man beschäftigt sich intensiv über ein halbes Jahr mit einer einzigen von vielen Fachrichtungen des Studiums und sucht sich dort auch nur einen Teilaspekt heraus - danach wird man zu diesem Thema nahezu alles wissen und verstehen. Doch leider musste ich, wie schon der berühmte Philosoph Sokrates 2.500 Jahre vor meiner Zeit, feststellen: Scio me nihil scire (Ich weiß, dass ich nichts weiß).

Soweit ich mich zurückerinnern kann, habe ich mich auf den Tag gefreut, an dem ich endlich mein Studium abschließen werde und `fertig mit dem Lernen` sein werde. Zumindest 50 % davon habe ich erreicht. Für die Wissenschaft trifft wohl das zu, was ein Baumarkt schon sehr lange mit seinem Slogan propagiert: "Es gibt immer was zu tun".

An dieser Stelle möchte ich den Menschen danken, die mich in meinem Leben bisher begleitet und unterstützt haben; besonders während des Studium und beim Schreiben der Master Arbeit, aber auch schon vorher. So wäre ich nie an den Punkt gekommen, an dem ich jetzt stehe, wenn ich nicht schon in jungen Jahren soviel Unterstützung bekommen hätte, sei es durch das Technik Lego, den ersten Computer oder einfach die Zeit, die mit mir verbracht wurde. Diese Unterstützung ist nie weniger geworden und ich durfte auf sie während meines gesamten Studiums immer zurückgreifen. Dafür bin ich meinen Eltern Christine und Hubert Schmeisser unendlich dankbar.

Im Gegensatz zum Bachelor Studium, bei dem ich die Mindeststudienzeit doch ein wenig überschritten hatte, konnte ich das Master Studium in der dafür vorgesehenen Mindeststudienzeit plus Toleranzsemester abschließen. Das wäre ohne meine Partnerin Aniko Enders niemals möglich gewesen. Nicht nur wegen der moralischen Unterstützung, die enorm war, sondern auch weil sie mir so viele Dinge abgenommen hat und ich mich daher voll dem Studium widmen konnte.

Dem Studium bin ich immer berufsbegleitend nachgegangen. Das wäre nicht möglich gewesen, wenn mir mein Arbeitgeber, die Energie Steiermark AG, kein so großes Verständnis und die notwendige Flexibilität über die ganzen Jahre hinweg entgegengebracht hätte. Stellvertretend für alle meine Kollegen möchte ich meinem Vorgesetzten Hrn. DI. Heimo Windisch ganz besonders dafür danken.

Last but not least gilt mein Dank Hrn. Univ.-Prof. Dipl.-Ing. Dr. techn. Franz Wotawa und Fr. Dipl.-Ing. Birgit Hofer am Institut für Softwaretechnologie der TU Graz. Die Zusammenarbeit bestand schon seit dem Master Projekt und führte über das Diplomanden Seminar bis hin zur Master Arbeit. Ohne deren fortwährende fachliche Unterstützung wäre es nicht möglich gewesen diese Arbeit abzuschließen.

Martin Schmeisser
Graz, 2011

Statutory Declaration

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

Graz, ____________________ ____________________

Place, Date Signature

Eidesstattliche Erklärung

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

Graz, am ____________________ ____________________

Ort, Datum Unterschrift

Inhaltsverzeichnis

Tabellenverzeichnis ix

Abbildungsverzeichnis x

Auflistungsverzeichnis xi

1. Einführung 1
- 1.1. Ausgangssituation . . . 2
- 1.2. Motivation . . . 2
- 1.3. Herausforderungen . . . 2
  - 1.3.1. Der Suchraum . . . 2
  - 1.3.2. Die minimale Lösung . . . 2
  - 1.3.3. Auswahl der Testfälle . . . 2
- 1.4. Gefahren . . . 3

2. Genetische Programmierung 4
- 2.1. Individuum . . . 4
- 2.2. Population . . . 4
- 2.3. Mutation . . . 5
- 2.4. Crossover . . . 5
- 2.5. Fitnessfunktion . . . 5
- 2.6. Selektion . . . 6
- 2.7. Minimale Lösung . . . 6

3. Verwandte Arbeiten 7
- 3.1. Automatisches Lokalisieren von Fehlern . . . 7
  - 3.1.1. Modell-basierter Ansatz . . . 7
  - 3.1.2. Spektrum-basierter Ansatz . . . 7
- 3.2. Automatisches Korrigieren von Fehlern . . . 8
  - 3.2.1. Fitnessfunktion . . . 8
  - 3.2.2. Selektion der Testfälle . . . 9
  - 3.2.3. Mutation . . . 9
  - 3.2.4. Crossover . . . 9
  - 3.2.5. Parametrierung . . . 10
  - 3.2.6. Ergebnisse . . . 10
- 3.3. Einfluss anderer Arbeiten auf die Implementierung der JCC Engine . . . 10

4. Automatische Fehlerkorrektur 12
- 4.1. Zielsetzung . . . 12

4.2. Algorithmus 13
4.3. Demonstration anhand eines Code-Beispiels 15

5. Implementierung **20**
5.1. Programmiersprache 20
5.2. Entwicklungsumgebung 20
5.3. Bibliotheken 20
5.3.1. AST Manipulation 21
5.3.2. Eclipse JDT 21
5.3.3. Apache log4 21
5.4. Spezielle Anforderungen an die Implementierung 21
5.4.1. Lose Koppelung der Komponenten 21
5.4.2. Threadsicherheit und parallele Verarbeitung 21
5.4.3. Ausgaben auf System.err 22
5.5. Einschränkungen aus der Implementierung 22
5.5.1. Aufrufe von System.exit() 22
5.5.2. Eindeutiger Name für Testfälle 22
5.5.3. Compile Fehler im fehlerhaften Programm 23
5.5.4. Endlosschleife im fehlerhaften Programm 23
5.6. Anforderungen an das zu untersuchende Programm 23
5.6.1. Programm 23
5.6.2. Testfälle 23
5.6.3. Fehlerlokalisierung 23

6. Implementierung der Komponenten **25**
6.1. Program Engine 25
6.1.1. ProgramFactory 25
6.1.2. IProgramResources 26
6.1.3. IProgramBase 26
6.1.4. IProgram 26
6.1.5. IProgramChangedSourceCode 26
6.2. Mutation Engine 27
6.2.1. IMutationEngine 27
6.2.2. IMutationOperator 28
6.2.3. IMutationInfo 29
6.3. Compile Engine 29
6.3.1. ICompiler 29
6.3.2. IProgramByteCode 30
6.4. Test Engine 30
6.4.1. ITestEngine 30
6.4.2. ITestSuite 31
6.4.3. ITestSuiteSet 31
6.4.4. ITest 32
6.4.5. ITestResult 32
6.4.6. ITestResults 32
6.5. Fitness Function 32
6.5.1. IFitnessFunction 32
6.6. Population 33
6.6.1. IPopulation 33
6.6.2. IIndividuum 35
6.6.3. IPopulationSetting 36
6.7. Execution Engine 36
6.7.1. IExecutionEngine 36
6.7.2. ITestSuiteSetEngine 39

6.7.3. Abbruchkriterien . . . 39
6.7.4. Reduzierung zur Final Repair . . . 39
6.7.5. IPerformance . . . 39
6.8. Configuration . . . 40

7. Bedienung und Steuerung **41**
7.1. Konfiguration . . . 41
7.1.1. Programm . . . 41
7.1.2. Abbruchkriterien . . . 42
7.1.3. Mutation . . . 42
7.1.4. Test Set . . . 42
7.1.5. Fitnesswert . . . 43
7.1.6. Population . . . 43
7.2. JCC GUI . . . 44
7.2.1. Informationsbereich . . . 45
7.2.2. Population . . . 46
7.2.3. Generationen . . . 46
7.2.4. Programm Änderungen . . . 47
7.2.5. TestSuiteSets Einstellungen . . . 47
7.2.6. Performance Entwicklung . . . 47
7.2.7. Logging Einstellungen . . . 47

8. Evaluierungsaufbau **50**
8.1. Verwendete Hardware . . . 50
8.1.1. Lokaler Rechner: Cyria . . . 50
8.1.2. Virtuelle Cloud Server: JiffyBox 01 - 50 . . . 50
8.1.3. Java VM Parameter . . . 51
8.2. Evaluierungsdatenbank . . . 51
8.2.1. Tabellen für die Parameter . . . 52
8.2.2. Tabellen für die Programme und Programmfehler . . . 53
8.2.3. Tabellen für die Evaluierungsläufe . . . 53
8.2.4. Tabellen der Ergebnisse . . . 55
8.3. Tools zur Evaluierung . . . 55
8.4. Programme zur Evaluierung . . . 56
8.5. Definition für korrigiertes Programm . . . 57
8.6. Bestimmung der Fehlerwahrscheinlichkeiten . . . 57

9. Evaluierungsergebnisse **59**
9.1. Korrigierbarkeit der Programmfehler . . . 59
9.1.1. Fragestellung . . . 59
9.1.2. Versuchsaufbau . . . 60
9.1.3. Ergebnis . . . 61
9.2. Parametrierung der Mutationsoperatoren . . . 67
9.2.1. Fragestellung . . . 67
9.2.2. Erwartetes Ergebnis . . . 67
9.2.3. Versuchsaufbau . . . 67
9.2.4. Ergebnis . . . 68
9.3. Parametrierung der Population . . . 70
9.3.1. Fragestellung . . . 70
9.3.2. Versuchsaufbau . . . 71
9.3.3. Ergebnis . . . 71
9.4. Parametrierung der Fitnessfunktion . . . 73
9.4.1. Fragestellung . . . 73
9.4.2. Erwartetes Ergebnis . . . 73

9.4.3. Versuchsaufbau 74
9.4.4. Ergebnis 75
9.5. Parametrierung der TestSuiteSets 75
9.5.1. Fragestellung 75
9.5.2. Versuchsaufbau 76
9.5.3. Ergebnis 76
9.6. Evaluierung der Performance 77
9.6.1. Fragestellung 77
9.6.2. Erwartetes Ergebnis 78
9.6.3. Versuchsaufbau 78
9.6.4. Ergebnis 79

10. Erkenntnisse **82**
10.1. Erkenntnisse aus dem Ergebnis 82
10.2. Erkenntnisse aus dem Evaluierungsprozess 83
10.3. Erkenntnisse aus der Implementierung 83

11. Ausblick **84**
11.1. Erweitern der Mutationsoperatoren 84
11.2. Evaluierungen mit komplexeren Fehlern durchführen 84
11.3. Einbinden der JCC Engine in eine IDE 84
11.4. Verbesserungen bei der Implementierung 85
11.4.1. Entfernen von Einschränkungen 85
11.4.2. Verbesserter Umgang mit Timeouts bei Testfällen 85
11.4.3. Verbessern der Performance 85

Literaturverzeichnis **87**

Tabellenverzeichnis

8.1. Java Virtual Machine Parameter 51
8.2. Übersicht der zu evaluierenden Programme 56

9.1. JTOPAS_v1 Fehlerübersicht 61
9.2. JTOPAS_v2 Fehlerübersicht 62
9.3. JTOPAS_v3 Fehlerübersicht 62
9.4. TCAS Fehlerübersicht 63
9.5. ATMS Fehlerübersicht 63
9.6. REFLECTION-VISITOR Fehlerübersicht 64
9.7. Durchschnittswerte und Standardabweichungen für korrigierbare Programmfehler 65
9.8. Testanzahl bis die erste Lösung je Programmfehler gefunden wurde 66
9.9. Evaluierungsergebnisse der Mutationsparameter 70
9.10. Evaluierungsergebnisse der Populationsparameter 73
9.11. Evaluierungsergebnisse der Fitnesswertparameter 75
9.12. Evaluierungsergebnisse der TestSetParameter 77
9.13. Verfügbare Leistungsstufen der JiffyBox 78

Abbildungsverzeichnis

4.1. Pseudocode für die automatische Fehlerkorrektur eines Programmes . . . 13
4.2. Pseudocode zur Berechnung des Fitnesswertes . . . 14
4.3. Übersicht über den Programmablauf . . . 15

6.1. Klassendiagramm für Program Engine . . . 25
6.2. Klassendiagramm für Program Factory . . . 26
6.3. Klassendiagramm für die Mutation Engine . . . 27
6.4. Klassendiagramm für die Compile Engine . . . 29
6.5. Klassendiagramm für die Test Engine . . . 31
6.6. Klassendiagramm für die Fitness Funktion . . . 33
6.7. Klassendiagramm für die Population . . . 34
6.8. Ablauf bei der Erstellung eines neuen Individuums . . . 34
6.9. Ablauf beim Hinzufügen eines neuen Individuums zur Population . . . 35
6.10. Klassendiagramm für die Execution Engine . . . 37
6.11. Ablauf innerhalb der Execution Engine . . . 38
6.12. Klassendiagramm für die Parameter Konfiguration . . . 40

7.1. Hauptfenster der GUI . . . 44
7.2. GUI für die TestSuiteSets . . . 48
7.3. GUI für die Performance Entwicklung . . . 48
7.4. GUI für das Logging . . . 49

8.1. Evaluierungstabelle für die Parametrierung der Mutationsoperatoren . . . 53
8.2. Evaluierungstabelle für die Programme . . . 53
8.3. Evaluierungstabelle für die Fehler der Programme . . . 54
8.4. Evaluierungstabelle für die Evaluierungsläufe . . . 54
8.5. Evaluierungstabelle für den Status der Evaluierungsläufe . . . 54

9.1. Testanzahl bis die erste Lösung gefunden wurde . . . 67
9.2. Verhalten der Performance in Abhängigkeit der Threads . . . 80
9.3. Verhalten der Performance in Abhängigkeit der Evaluierungsumgebung . . . 81

Auflistungsverzeichnis

4.1. Fehlerhafter Programmabschnitt von JTOPASv1 Fehler Nr. 2 16
4.2. Testfall testWrappedException. Assert in Zeile 12 schlug fehl. 16
4.3. Testfall testNestedExceptions. Assert in Zeile 15 schlug fehl. 16
4.4. Testfall testMessageFormatting. Null Pointer Exception in Zeile 15 geworfen. 17
4.5. Programmabschnitt nach DELETE Operator 17
4.6. Programmabschnitt nach weiterem DELETE Operator 17
4.7. Programmabschnitt nach INSERT Operator 18
4.8. Programmabschnitt nach BACKCROSSING Operator 18
4.9. Programmabschnitt nach FINAL REPAIR 19

7.1. Konfigurationsfile mit Fehlerwahrscheinlichkeiten 42

8.1. Script zur einfachen Steuerung der JiffyBox API 52
8.2. Script zum Ausführen der Evaluierung. 56
8.3. Cronjob zum Starten des Evaluierungsscripts. 56

Kapitel 1

Einführung

> *"Knowing that a program has a bug is good,*
> *knowing its location is better,*
> *but a fix is best." [16]*

Sehr viele Arbeiten und Bücher beschäftigen sich mit dem Testen von Software [21, 5, 22]. Viele Arbeiten beschäftigen sich mit der Lokalisierung bzw. Eingrenzung von Fehlern [1, 4, 20, 17, 3]. Und immer mehr Arbeiten beschäftigten sich mit einer der Königsklassen im Bereich der Softwareentwicklung bzw. Softwareinstandhaltung: dem automatischen Korrigieren von Fehlern in Software [8, 6, 26, 14, 25, 13, 24, 11].

Auf den ersten Blick scheint es unmöglich eine fehlerhafte Software automatisch, ohne den Eingriff von Außen, zu korrigieren. Auf den zweiten Blick bleibt es zwar noch immer so, es eröffnen sich aber Möglichkeiten, Programme unter gewissen Umständen automatisch zu korrigieren. Die Gemeinsamkeit in allen Arbeiten ist, dass kein Ansatz oder keine Lösung alle Fehler in einer Software automatisch korrigieren kann. Je nach Branche, wird für das Testen und die Instandhaltung von Software 40 % der Entwicklungszeit [23] und folglich auch der Kosten aufgewendet. Somit hätten auch Automatismen, die nur einen Teil abdecken, hohes Einsparungspotenzial.

Die betrachteten Lösungen zu diesem Thema liefern gute Ansätze, jedoch war von den drei näher betrachteten Ansätzen nur einer dabei, der eine vollständige Implementierung vorzuweisen hatte, die auch auf Programme aus der Praxis anwendbar gewesen wäre. Das führte letztendlich zu dieser Arbeit mit dem Ziel, eine vollständige Implementierung zum automatischen Korrigieren von Programmen zu erstellen. Das Ergebnis ist die `Java Code Correction Engine (JCC Engine)`, die auf Basis der `Genetischen Programmierung` für fehlerhafte Java Programme automatisiert eine Lösung sucht.

In dieser Einführung werden die Ausgangssituation, die Motivation, die Herausforderungen, aber auch die Gefahren bei dieser Arbeit beschrieben. In Kapitel 2 wird auf die Idee, die hinter der `Genetischen Programmierung` steht, näher eingegangen. In Kapitel 3 werden verwandte Arbeiten zu diesem Thema untersucht und es wird geprüft, ob Ansätze daraus für die `JCC Engine` verwendet werden können. Im darauffolgenden Kapitel 4 wird der gewählte Ansatz beschrieben und anhand eines Beispiels demonstriert. Die Implementierung des Algorithmus hinter diesem Ansatz wird in den nächsten beiden Kapiteln 5 und 6 vorgestellt. In Kapitel 7 werden die Konfigurationsparameter und die Möglichkeiten zur Bedienung der `JCC Engine` beschrieben. Dabei wird auch die `JCC GUI` vorgestellt, mit der die Funktionalität der `JCC Engine` visualisiert werden kann. Mit der Evaluierung und Präsentation der Ergebnisse beschäftigen sich die nächsten beiden Kapitel 8 und 9. Schließlich liefern die Kapitel 10 und 11 eine Zusammenfassung der Erkenntnisse und geben einen Ausblick für weitere mögliche Arbeiten und Verbesserungen.

1.1. Ausgangssituation

Als Ausgangssituation dienen Programme, die Fehler aufweisen. Zu diesen Programmen muss es eine Art `Oracle` geben. Das Oracle liefert für übergebene Eingangsparameter die korrekten Ausgabewerte des Programms, mit dem die Korrektheit oder das Fehlverhalten des Programms festgestellt werden kann. Automatische `Testfälle` sind in der Softwareentwicklung häufig für ein Programm verfügbar und können als Form eines `Oracle` betrachtet und verwendet werden.

1.2. Motivation

Testen und Korrigieren von Software ist sehr zeitintensiv und damit auch kostenintensiv. Ziel ist es, diesen Prozess weitestgehend zu automatisieren. Das soll möglichst geschehen, ohne zusätzlichen Aufwand während oder nach der Softwareentwicklung zu verursachen.

Eine Möglichkeit dafür ist die `Genetische Programmierung (GP)`, inspiriert von der biologischen Evolutionstheorie. Von einem fehlerhaften Programm werden Mutationen und Kreuzungen erzeugt und durch die Fitnessfunktion bewertet. Entsprechend dieser Bewertung werden die Programme wieder durch Mutationen und Kreuzungen weiter verarbeitet oder ausgeschieden. Im Idealfall lässt sich nach einigen Durchgängen eine Version des Programms ermitteln, in der der Fehler korrigiert worden ist.

1.3. Herausforderungen

1.3.1. Der Suchraum

Das größte Problem bei der `Genetischen Programmierung` ist, dass sich ein unendlich großer Suchraum für die Lösung ergibt; zum einen gibt es unendlich viele Stellen an denen das Programm defekt sein kann, zum anderen sind die Möglichkeiten diese Stellen zu verändern ebenso vielfältig.

1.3.2. Die minimale Lösung

Ebenso wie der Suchraum ist auch der Raum für die Lösungen unendlich groß, auch wenn er eine Teilmenge des Suchraums ist. Wird nun eine Lösung gefunden, so muss diese nicht zwingend die minimale Lösung sein. Wie eine minimale Lösung unter anderem bestimmt werden kann, sehen wir in Kapitel 6.7.4.

1.3.3. Auswahl der Testfälle

Um die Fitness (siehe Kapitel 2.5) eines Individuums zu bewerten, werden Tests durchgeführt. Stehen viele Testfälle zur Verfügung, wie es bei großen Projekten meist der Fall ist, so müssen geeignete Testfälle gewählt werden um die Fitness optimal bewerten zu können. Die Herausforderung dieser Auswahl besteht darin, zwischen zwei Anforderungen, die sich gegenseitig negativ beeinflussen, einen Kompromiss zu finden:

- Das Ausführen von Testfällen beansprucht einen überwiegenden Teil der Zeit, daher sollte die Anzahl möglichst gering sein.
- In den Testfällen ist die Spezifikation des Programms hinterlegt, daher sollte die Anzahl möglichst groß sein.

1.4. Gefahren

Idealerweise sollten die Mutationen und vor allem die Testläufe in einer Sandbox bzw. virtuellen Umgebung erfolgen. Der Code wird im Prinzip willkürlich mutiert und man hat keinen Einfluss darauf, welche Ergebnisse produziert werden. So könnte es z.B. passieren, dass ein Mutant plötzlich das gesamte Dateisystem löscht oder in den Speicherbereich eines anderen Programms schreibt, sofern dies nicht vom Betriebssystem abgefangen wird.

Kapitel 2

Genetische Programmierung

Die `Genetische Programmierung (GP)` gehört zur Klasse der `Evolutionären Algorithmen`. Diese haben als Vorbild die biologische Evolution, in der ausgehend von einem `Individuum` durch Veränderung eine `Population` von möglichen Lösungen erstellt wird. Die so entstandene Population wird bewertet und selektiert (siehe Kapitel 2.5 bzw. 2.6). Die verbleibenden Individuen werden weiter verändert und sind die Grundlage für die nächste Population.
Je nach Implementierung terminiert der Algorithmus, wenn (auch mehrere Ereignisse möglich)

- eine oder mehrere Lösungen gefunden worden sind.
- ein Individuum mit einem festgelegten Fitnesswert gefunden wurde.
- der Suchraum erschöpft ist (würde bedeuten, dass es keine Lösung gibt).
- ein Timeout erreicht worden ist.
- eine festgelegte Anzahl von Individuen erstellt worden ist.
- ...

2.1. Individuum

Unter `Individuum` bei der `Genetischen Programmierung` versteht man eine konkrete Ausprägung des Programms. Das zu Beginn fehlerhafte Programm, die hoffentlich gefundene Lösung und jede Variante des Programms, die dazwischen durch `Mutation` oder `Crossover` gebildet wird, sind als ein solches Individuum zu betrachten.

Als `offspring` bezeichnet man ein Individuum, das aus einem anderen Individuum, dem `parent`, durch `Mutation` oder `Crossover` erzeugt wird.

2.2. Population

Die `Population` bei der `Genetischen Programmierung` ist die Summe aller `Individuen`, die vor bzw. nach dem Durchführen von `Mutation` und `Crossover` vorhanden sind.

2.3. Mutation

Bei einer `Mutation` wird ein `Individuum` durch einen `Mutationsoperator` verändert und ein weiteres Individuum erstellt. Nach den bisher in der Literatur gefundenen `Mutationsoperatoren` würden diese in 4 Gruppen eingeteilt werden:

- **Delete** - Ein Teil des Programms wird entfernt
- **Insert** - Ein Teil des Programms wird an einer anderen Stelle eingesetzt
- **Swap** - Ein Teil des Programms wird mit einem Teil des Programms an einer anderen Stelle ersetzt
- **Change** - Ein Teil des Programms wird nach bestimmten Regeln verändert (z.B. Integer Werte werden um 1 erhöht oder gesenkt, Vergleichsoperatoren werden ausgetauscht, etc.)

Wir unterscheiden zwischen `first-order` und `higher-order` Mutanten:

- **first-order Mutanten**: Es wird immer nur das Ausgangsprogramm mutiert; die daraus entstehenden Mutanten werden nicht weiter mutiert.
- **higher-order Mutanten**: Mutanten, die durch Mutation des Ausgangsprogramms entstehen, werden weiter mutiert.

2.4. Crossover

Beim `Crossover` werden zwei `Individuen` miteinander gekreuzt, indem Teile vom ersten und zweiten Individuum zu einem neuen Individuum zusammengefügt werden.
In ähnlichen Arbeiten zu dem Thema (siehe Kapitel 3) wurden zwei unterschiedliche Varianten des `Crossover` verwendet:

- **Crossover innerhalb einer Generation**: Innerhalb einer Generation werden zwei Individuen miteinander gekreuzt.
- **Crossing Back**: Ein Individuum der jeweiligen Generation wird mit dem ursprünglichen Programm gekreuzt.

2.5. Fitnessfunktion

Eine `Fitnessfunktion` weist einem `Individuum` in einer `Population` einen Wert zu, der der Qualität des jeweiligen Individuum entspricht. Der Wertebereich kann dabei beliebig definiert werden, es scheint aber zweckmäßig Werte zwischen 0 (Individuum repräsentiert die Lösung überhaupt nicht) und 1 (Individuum ist eine Lösung bzw. ist die Lösung) zu wählen. Im Anwendungsgebiet für die Fehlersuche fällt die Repräsentation der Werte schon recht schwer. 0 wird meist für nicht kompilierbare Programme verwendet und 1 wenn das Programm fehlerfrei ist (alle Testfälle liefern das gewünschte Ergebnis). Die Werte dazwischen zuzuweisen gestaltet sich schon um einiges schwerer. Dabei handelt es sich um Programme, welche fehlerhaft sind, aber vielleicht schon einen Teil des Programms korrigiert haben; aber auch solche, die das Programm noch weiter verschlechtert haben.

Die Fitnessfunktion hat 2 Aufgaben:

- Sie dient als Terminierungs-Kriterium für die Suche. Wenn die Fitness einer Programmvariante 100% erreicht, dann ist die Lösung gefunden, die das Programm korrigiert, sodass alle Testfälle positiv ausgeführt werden können.
- Sie dient als Entscheidungskriterium, welche Programmvarianten einer Generation in die nächste übernommen werden. Es wird immer nur die Hälfte einer Generation weiter verarbeitet. Jede Variante wird mit der Fitnessfunktion bewertet und der höher bewertete Anteil wird in die nächste Generation übernommen.

Im einfachsten Fall werden den positiv und negativ absolvierten Testfällen Werte zugewiesen und diese aufsummiert; optional kann dieser Wert auf [0 - 1] normiert werden. Die Fitnessfunktion liefert dann das entsprechend gewichtete Verhältnis von positiven zu negativen Testfällen.
Aus Erfahrung weiß man, dass bei einem Programm eine minimale Änderung dazu führen kann, dass ein Programm keinen einzigen Testfall mehr positiv absolvieren kann und umgekehrt. Daher ist diese Implementierung der Fitnessfunktion, die Testfälle nur auf positiven oder negativen Ausgang prüft, nicht immer zielführend. Die Implementierung eines fortschrittlicheren Ansatzes findet man im Kapitel 3.

2.6. Selektion

Nach dem Generieren und Bewerten der neuen Generation (siehe Kapitel 2.3, 2.4 und 2.5) erfolgt die `Selektion` der Population. Dabei sollen die schlecht bewerteten Individuen ausgeschieden und die verbleibenden weiterverarbeitet werden.
Möglichkeiten zur Selektion sind:

- **Festlegung eines Schwellwertes** - Dabei werden jene Individuen, die eine parametrierte Bewertung nicht erreichen, ausgeschieden. Dabei gibt es zwei Varianten:
  - **Absolut** - Der Schwellwert ist fest parametriert. Jedes Individuum, das darunter ist, wird ausgeschieden.
  - **Relativ** - Anstelle des Schwellwertes ist ein Anteil angegeben, wie viele Individuen ausgeschieden werden sollen; entsprechend dieses Anteils werden die am schlechtesten bewerteten ausgeschieden.
- **Tournamen Selektion** - Es wird eine Anzahl an Individuen vorgegeben, die zufällig ausgewählt werden; der am besten von diesen bewertete verbleibt, die restlichen werden ausgeschieden.

2.7. Minimale Lösung

Unter der `Minimalen Lösung` versteht man die optimale Lösung, die zum fehlerfreien Programm führt. In der Softwareentwicklung ist die optimale Lösung nicht eindeutig. Meist versteht man darunter die Lösung, die mit der geringsten Anzahl an Änderungsschritten am Ausgangsprogramm zu einem fehlerfreien Programm führt. Diese Messgröße kann programmtechnisch überprüft werden, um aus mehreren Lösungen die minimale Lösung zu finden.

Kapitel 3

Verwandte Arbeiten

Im Zuge des Master Seminars wurden drei bestehende Ansätze zum automatischen Korrigieren von Programmen untersucht und miteinander verglichen (siehe Kapitel 3.2).

Bevor aber Fehler korrigiert werden können, müssen diese erst gefunden werden. Ist es nicht möglich den Fehler genau zu lokalisieren, so muss zumindest der Bereich eingeschränkt werden, in dem sich der Fehler befindet. Die im Seminar betrachteten Arbeiten bedienen sich ebenfalls dieser Verfahren, um die Fehlerkorrektur besser anwenden zu können. Auch wenn das Lokalisieren bzw. Eingrenzen der fehlerhaften Bereiche nicht Teil dieser Arbeit ist, wird kurz auf einige Ansätze zu diesem Thema verwiesen (siehe Kapitel 3.1).

Die Arbeiten lieferten interessante Ansätze und Strategien, von denen einige in der `JCC Engine` umgesetzt wurden (siehe Kapitel 3.3).

3.1. Automatisches Lokalisieren von Fehlern

In den betrachteten Arbeiten werden zwei unterschiedliche Ansätze für das automatische Lokalisieren von Fehlern verwendet. Dass es nicht immer nur „entweder ... oder" geben muss, zeigt die Implementierung des `DEPUTO` [1] Frameworks, in dem beide Ansätze miteinander kombiniert werden.

3.1.1. Modell-basierter Ansatz

Bei einem Modell-basierten Ansatz wird zuerst aus der Software ein Modell gebildet, das eine Abstraktion der Software darstellt. Dieses Modell kann dynamisch aus dem Programmverhalten, ähnlich wie beim Spektrum-basierten Ansatz, erzeugt werden. In dem Modell sind die unterschiedlichen Komponenten der Software abgebildet. Jeder Komponente ist ein Gesundheitszustand zugewiesen; die Summe aller Gesundheitszustände ergibt den Gesundheitszustand des Gesamtmodells bzw. der Software. Gesucht wird nun eine Diagnose um das System zu reparieren [4, 20].

3.1.2. Spektrum-basierter Ansatz

Unter einem Spektrum versteht man eine Sammlung von Daten, die eine bestimmte Sicht auf das dynamische Verhalten einer Software liefern [15]. Diese Daten werden zur Laufzeit der Software ermittelt. Mit diesem Spektrum wird die Fehlerwahrscheinlichkeit eines Statements in der Software bestimmt. Implementierungen dazu sind unter anderem `Tarantula` [17], `Pinpoint` [9] und `Ample` [10].

Zur Reihung der möglichen Fehlerstellen wird ein Ähnlichkeitskoeffizient verwendet. Der `Ochiai` [3] Koeffizient liefert aus 8 untersuchten Koeffizienten das beste Ergebnis [2].

3.2. Automatisches Korrigieren von Fehlern

`Ansatz 1` war von Arcuri, dessen Arbeiten zum Thema `Automatic Bug Fixing (ABF)` mit `Genetischer Programmierung` vor den anderen veröffentlicht worden sind [8, 6]. Seine Implementierung war allerdings nicht direkt auf Programme aus der Praxis anwendbar, da der Algorithmus auf eine formale Spezifikation des Programms angewiesen war. Zusätzlich musste das fehlerhafte Programm in eine eigene Sprache übergeführt werden, um es zu verarbeiten.
Aktuell (Juni, 2011) gibt es eine neue Veröffentlichung [7] von ihm, in der er die Thematik weiter abhandelt und die Implementierung des Java Framework `Java Automatic Fault Fixer - JAFF`, aufbauend auf den ersten Arbeiten, vorstellt.

`Ansatz 2` war von einer Gruppe von Wissenschaftlern, die sich ebenfalls dieser Thematik angenommen haben und mehrere aufeinander aufbauende Arbeiten zu diesem Thema veröffentlicht haben. [26, 14, 25, 13, 24]. Im Gegensatz zu Arcuri versuchten sie, die automatische Fehlerkorrektur auf Programme anzuwenden ohne sie vorher in eine eigene Sprache überführen zu müssen. Die untersuchten Programme waren auch entsprechend umfangreicher. Dieses Verfahren war daher auch in der Praxis leichter anzuwenden.
In späteren Arbeiten haben sie das Verfahren immer weiter entwickelt. Zum einen verbessern sie die Auswahl der Testfälle für die Fitnessfunktion, zum anderen verbessern sie die Fitnessfunktion selbst. Die Verbesserung der Fitnessfunktion erfolgt durch hinzufügen von Prädikaten (Aussagen) im fehlerhaften Programm. Dadurch kann bei der Durchführung eines Testfalls mehr Information gewonnen werden. Mit der ersten Implementierung konnte nur die Aussage getroffen werden, ob der Testfall erfolgreich ausgeführt worden ist oder nicht. Durch die zusätzliche Überprüfung, wie viele Prädikate gültig sind, erhält man ein aussagekräftigeres Ergebnis des Testfalles. Dieses Ergebnis beeinflusst in weiterer Folge auch die Fitnessfunktion und verleiht ihr dadurch mehr Aussagekraft als in der vorherigen Implementierung.

Bei `Ansatz 3` wurde ein einfacher Ansatz für das automatische Korrigieren von Fehlern gewählt [11]. Zuerst wird der Bereich im Programm, der fehlerhaft ist, eingegrenzt. Das wurde mit Hilfe der `Tarantula Formel` [17] durchgeführt. Diese liefert für jedes Statement im Programmcode die Wahrscheinlichkeit, dass das Statement fehlerhaft ist. Es werden nun der Reihe nach, beginnend mit dem Statement, das die höchste Fehlerwahrscheinlichkeit aufweist, alle Mutationen auf dieses Statement angewandt und die entstandene Programmvariante auf Fehlerfreiheit geprüft. Das wird so lange wiederholt, bis alle möglichen Mutationen auf das erste Statement anwendet worden sind. Wurde der Fehler damit nicht gefunden, wird dasselbe mit dem nächsten Statement in der Liste durchgeführt. Als Abbruchkriterium kann eine feste Anzahl von Statements verwendet werden, die mutiert werden sollen. Diese Grenze kann absolut (die ersten 20 Statements), relativ (die ersten 10 % der Statements) oder über die Fehlerwahrscheinlichkeit (alle Statements mit einer Fehlerwahrscheinlichkeit über 50 %) gewählt werden.

3.2.1. Fitnessfunktion

`Ansatz 1 und 2` greifen für die Bewertung der Fitness des Programms auf Testfälle zurück. Die Testfälle bei `Ansatz 1` liefern zusätzlich zur Information `pass` oder `fail` auch noch die Information, wie weit das Programm von der Lösung entfernt ist. Das gelingt bei `Ansatz 2` erst nachdem die Fitnessfunktion entsprechend um `Prädikate` erweitert wurde, die zusätzlich ausgewertet werden, um weitere Informationen zu erhalten. Durch diese Erweiterung funktioniert aber auch dieser Ansatz nicht mehr 'out of the box' und es muss mehr Aufwand betrieben werden.

Bei beiden Ansätzen erkennt man, dass eine bessere Fitnessfunktion einer der Schlüssel für den Erfolg der `Genetischen Programmierung` ist, und entsprechend große Anstrengungen unternommen worden sind, um diese zu verbessern.

`Ansatz 3` benötigt keine, mit den beiden anderen Ansätzen vergleichbare Fitnessfunktion, da keine `higher-order` Mutanten generiert werden. Es wird nur ermittelt, ob das Programm (ein `first-order` Mutant) alle Testfälle erfüllt und somit eine Korrektur sein kann oder nicht.

3.2.2. Selektion der Testfälle

`Ansatz 2` geht hier zu Beginn den einfachen Weg und verwendet die Testfälle einfach händisch vorausgewählt. Später werden einige Varianten untersucht, um die Performance zu steigern. Der, meiner Meinung nach, einfachste Ansatz - alle negativen Testfälle zu verwenden und nur einen positiven Testfall, der zufällig ausgewählt wird - führte zur größten Steigerung der Performance - 81 %.

`Ansatz 1` verwendet hier einen viel komplexeren Ansatz, der auch die Kernidee der ganzen Arbeit wiedergibt - Testfälle und Programme beeinflussen und verbessern sich gegenseitig. Es werden in Relation zu `Ansatz 2` eine viel größere Anzahl an Testfällen verwendet. Durch die formale Spezifikation gibt es auch einen sehr großen bis unendlichen Vorrat an Testfällen. Leider gibt es keine Vergleiche, wie die Performance mit einer anderen Variante, z.B. der von `Ansatz 2` gewesen wäre.

Die Selektion der Testfälle wirkt sich in beiden Ansätzen auf die Performance aus, ihr wird daher entsprechend Aufmerksamkeit geschenkt.

`Ansatz 3` verwendet hier im Ansatz alle verfügbaren Testfälle. Da es im Zuge des Experiments allerdings gar nicht zur Durchführung der Testfälle kommt (es wird ein `String-Vergleich` durchgeführt), hat das in dem Fall keine Auswirkung. Es wird allerdings darauf hingewiesen, dass man in der Praxis sehr wohl die Testfälle einschränken sollte - Details dazu gibt es keine.

3.2.3. Mutation

Bei `Ansatz 1` werden die Mutations-Operatoren auf einen Knoten im Baum angewandt. Sie bieten die gleichen Möglichkeiten wie bei `Ansatz 2`, können aber das selbe Ergebnis in weniger Mutationsschritten erreichen, weil sie das Ändern von mehreren Knoten in einem Schritt unterstützen.

Bei `Ansatz 2` wird davon ausgegangen, dass für die Korrektur eines Programms ein oder mehrere Teile von einer anderen Stelle im Programm verwendet werden können. Das schränkt den Suchraum für die Lösung ein und es wäre sogar möglich, dass keine Lösung vorhanden ist. Im Experiment hat sich gezeigt, dass dies jedoch nie der Fall war. Sie haben auch Ansätze vorgestellt, um dem ggf. entgegen steuern zu können.

Bei `Ansatz 3` sind die Mutationsmöglichkeiten, und damit auch die Möglichkeit Fehler zu korrigieren, sehr stark eingeschränkt. Sie können aber erweitert werden, lösen dann aber immer nur eine Klasse von Fehlern.

3.2.4. Crossover

In `Ansatz 1` stellte sich heraus, dass Crossover notwendig für eine gute Performance ist, der Wert aber nicht zu groß sein darf. Es wurden leider keine genauen Informationen zu Veränderungen der Performance in Abhängigkeit von der `Crossover Rate` angeführt.

Bei `Ansatz 2` hatte Crossover noch eine weitere Funktion: Crossover sollte verhindern, dass sich die Individuen zu weit vom ursprünglichen Programm entfernen. Bei `Ansatz 1` wird dies dadurch erreicht, indem je Generation zufällig das ursprünglich fehlerhafte Programm wieder in die Population der Generation eingefügt wird.

`Ansatz 3` verwendet kein Crossover, da nur `first-order` Mutanten erzeugt werden.

3.2.5. Parametrierung

Große Unterschiede gab es auch bei der Parametrierung. Teilweise ist sie nicht vergleichbar, weil unterschiedliche Ansätze unterschiedliche Parameter benötigten. Einer der signifikantesten Unterschiede war aber sicher die Größe der Population, die bei `Ansatz 1` 1.000 und bei `Ansatz 2` nur 40 war. Auch wurden bei `Ansatz 1` 5x mehr Generationen erzeugt als bei `Ansatz 2`. Das dürfte aber am Wesen der Programme liegen: das einfache Programm aus `Ansatz 1` ist viel schneller zu testen als die, in Relation dazu, hochkomplexen Programme von `Ansatz 2`.

Bei `Ansatz 1 und 2` sieht man, dass die Auswahl der Parameter sehr wichtig ist und alle Autoren wollen dahingehend auch weiter forschen. In der `Genetischen Programmierung` hängt sehr viel vom Zufall ab und dieser wird großteils über Parameter gesteuert.

`Ansatz 3` benötigt keine Parameter für Wahrscheinlichkeitswerte. Lediglich für das Abbruchkriterium können Werte definiert werden.

3.2.6. Ergebnisse

Alle Ansätze haben ihre Aufgabenstellung erfolgreich gelöst.

Die Ergebnisse bei `Ansatz 2` sind bei Weitem umfangreicher und bieten viel Spielraum für weiteres Forschen. Dieser Ansatz ist auch leichter in die Praxis zu übernehmen, weil schon für die Durchführung der Tests Programme verwendet wurden, die sich im produktiven Einsatz befinden.

Auch `Ansatz 1` liefert gute Ergebnisse. Um es näher an die Praxis zu bringen, müsste es aber direkt anwendbar werden.

Dass die Fehler bei `Ansatz 1` künstlich erzeugt wurden, spricht ebenfalls weniger für die direkte Anwendbarkeit in der Praxis - wobei hingegen bei `Ansatz 2` Fehler herangezogen wurden, wie sie schon bei der Entwicklung des jeweiligen Programms aufgetaucht sind.

Die Stärken von `Ansatz 3` liegen darin, dass durch die Wahl der Mutations-Operatoren das Programm auf die Korrektur von bestimmten Fehlern getrimmt werden kann. Wenn aber für den jeweiligen Fehler, der in der Regel im Vorfeld nicht bekannt ist, kein Mutations-Operator implementiert ist, wird das Finden des Fehlers nicht möglich sein. Der Ansatz implementiert auch nicht das Finden von mehreren Fehlern, wie es bei den ersten beiden Ansätzen möglich ist. Es werden aber mögliche Ansätze angegeben, wie man das implementieren kann. Das Finden von mehreren Fehlern gliedert sich in Fehler, die in der selben Zeile vorhanden sind und solchen, die an unterschiedlichen Stellen auftreten; diese notwendige Unterscheidung erschwert die Implementierung. Auch diese Limitierung ist bei den ersten beiden Ansätzen nicht vorhanden.

3.3. Einfluss anderer Arbeiten auf die Implementierung der JCC Engine

Die Berechnung der Fitnessfunktion wird analog zu `Ansatz 2` implementiert. Die Parameter für die Bewertung der Testfälle werden ähnlich gewählt. Im Zuge der Evaluierung werden davon abweichende Parametrierungen verwendet um die Auswirkungen zu beobachten.

Bei der Selektion der Testfälle wird ebenfalls `Ansatz 2` verfolgt. So wird zuerst mit einer reduzierten Anzahl an Testfällen die Mutation hinsichtlich ihres Fitnesswertes geprüft. Erst wenn dieser Wert das Maximum erreicht hat, wird die Mutation mit allen Testfällen geprüft. Die Implementierung der `JCC Engine` bietet, im Vergleich zu den untersuchten Ansätzen, noch weitere Möglichkeiten. So können Testfälle hinsichtlich ihrer Laufzeit selektiert werden und die Anzahl der Testfälle kann beliebig angepasst werden. Die Auswirkungen dieser Einstellungen wird die Evaluierung zeigen.

Bei den zu implementierenden `Mutationsoperatoren` wird der `Ansatz 2` verfolgt, der davon ausgeht, dass fehlerhafter Code durch Hinzufügen oder Vertauschen von Code aus anderen Bereichen korrigiert werden kann. Ebenso wird das Löschen von Code als eigener `Mutationsoperator` implementiert.

Auf das direkte Implementieren von einer `Crossover` Operation, wie sie in `Ansatz 1 und 2` verwendet werden, wird verzichtet. Es gibt keine Programmvariante, die durch `Crossover` entstehen könnte, die nicht auch durch einfaches Mutieren erzeugt werden könnte. Sehr wohl wird aber der Ansatz des `Backcrossings` implementiert; nicht als vollständige Kreuzung der Programmvariante mit dem ursprünglichen Programm, sondern nur indem Teile des bereits Mutierten wieder mit dem ursprünglichen Programm ersetzt werden. Dadurch soll, wie in `Ansatz 2` beschrieben, verhindert werden, dass sich die Programmvarianten in der Population zu weit vom ursprünglichen Programm entfernen.

Im Bereich der `Population` bietet die `JCC Engine` auch mehr Möglichkeiten als die untersuchten Ansätze. So gibt es Parameter um Programmvarianten mit hohem Fitnesswert vor dem Ausscheiden aus der Population zu schützen, oder Programmvarianten mit hohem Fitnesswert öfter zu mutieren als solche mit niedrigerem.

Kapitel 4

Automatische Fehlerkorrektur

Im Folgenden werden nun die Zielsetzungen in Abschnitt 4.1 betrachtet, die an eine automatische Fehlerkorrektur gestellt werden. Des Weiteren wird ein Algorithmus in Abschnitt 4.2 vorgestellt, der diese Zielsetzungen abbildet. Mit einem praktischen Demonstrationsbeispiel in Abschnitt 4.3 wird dieses Kapitel abgerundet.

4.1. Zielsetzung

Das Hauptziel ist das automatische Korrigieren von fehlerhaften Programmen. Das Korrigieren von Fehlern soll ohne spezielle Vorverarbeitungsschritte erfolgen können; z.B. war es im ersten Ansatz bei den verwandten Arbeiten (siehe Kapitel 3.2) notwendig, dass das fehlerhafte Programm zuerst manuell in ein geeignetes Modell übergeführt werden musste, um korrigiert werden zu können.

Es wird auch versucht, bestehende Informationen bestmöglich zu nutzen. So werden neben dem Programm auch die Testfälle des Programms verwendet. Die fehlgeschlagenen Testfälle dienen zum Auffinden von Fehlern und auch zum Feststellen, ob eine Lösung für den Fehler gefunden wurde. Die positiven Testfälle helfen, die Spezifikation des Programms beizubehalten. Ansonsten wäre es ein Leichtes, den fehlerhaften Bereich in der Software zu entfernen und es so als Lösung anzubieten; der Fehler würde dann nicht mehr auftreten und der Testfall würde positiv ausgeführt werden können. In diesem Fall sollten Teile der bisher positiven Testfälle negativ ausfallen und damit eine Verletzung der Spezifikation anzeigen.

Die Lösungsfähigkeit soll nicht auf gewisse Fehlerklassen eingeschränkt sein. Es wird zwar gefordert, dass das Programm kompilierbar sein muss, aber abgesehen davon, soll es keine Einschränkungen geben.

Die Zeit, die für das Finden einer Lösung benötigt wird, darf nicht zu lang sein. Es darf daher die Performance bei der späteren Umsetzung nicht aus den Augen gelassen werden. Auch die verfügbare Hardware soll bestmöglich genutzt werden. In der Praxis gibt es für das automatische Korrigieren von Programmen 2 Anwendungsszenarien:

- **Laufende Entwicklung**: Während der Entwicklung werden vom Entwickler, nach Änderungen am Programm, die Testfälle ausgeführt. Tritt hierbei ein Fehler auf, so sollte die automatische Fehlerkorrektur gestartet werden. In diesem Fall muss gewährleistet sein, dass die Fehler schneller automatisch als manuell korrigiert werden können.
- **Nach der Entwicklung**: Viele Firmen, die Software entwickeln, lassen in der Nacht die Testfälle über die gesamte Software laufen. Treten hier Fehler auf, soll ebenfalls versucht werden die Fehler automatisch zu korrigieren. In diesem Fall ist der Zeitdruck nicht so groß und es kann umfangreicher nach einer Korrektur gesucht werden.

Es soll daher möglich sein, die Zeit, die für das Korrigieren verfügbar ist, zu begrenzen.

Nachdem eine Korrektur für ein Programm gefunden worden ist, kann diese unter anderem als `Patch` repräsentiert werden. Darin sind nur die Änderungen vom fehlerhaften zum korrigierten Programm enthalten. Dieser soll keine unnötigen Statements enthalten, um möglichst effizient und leicht lesbar zu sein. Daher sollte eine Lösung hinsichtlich obsoleter Teile im Programm untersucht und bereinigt werden. Ein Beispiel für obsolete Statements wäre z.B. `x=y++; x=5;`, dabei ist `x=y++` obsolet, da `x` im nächsten Schritt überschrieben wird.

4.2. Algorithmus

Der Algorithmus, für das automatische Korrigieren von Fehlern, wird in Form von Pseudocode (siehe Abbildung 4.1) und als Ablaufdiagramm (siehe Abbildung 4.3) präsentiert.

Input: Fehlerhaftes Programm P
Input: Zu P gehörige Testfälle T mit mindestens einem fehlgeschlagenen Testfall
Input: Fehlerwahrscheinlichkeiten F je Statement in P
Output: Korrigiertes Programm

1: $\langle TS_p, TS_f \rangle \leftarrow createTestSuiteSets(P, T)$
2: $FV_p \leftarrow calculateFitnessValue(P, TS_p)$
3: $FV_f \leftarrow calculateFitnessValue(P, TS_f)$
4: $Population \leftarrow \langle P, FV_p, FV_f \rangle$
5: **repeat**
6: $\quad \overline{P} \leftarrow getProgram(Population)$
7: $\quad \overline{P} \leftarrow mutate(\overline{P}, F)$
8: $\quad FV_p \leftarrow calculateFintessValue(\overline{P}, TS_p)$
9: $\quad$ **if** $FV_p = MAX_FITNESS_VALUE$ **then**
10: $\quad\quad FV_f \leftarrow calculateFitnessValue(\overline{P}, TS_f)$
11: $\quad\quad$ **if** $FV_f = MAX_FITNESS_VALUE$ **then**
12: $\quad\quad\quad \overline{P} \leftarrow createFinalRepair(\overline{P})$
13: $\quad\quad$ **end if**
14: $\quad$ **else**
15: $\quad\quad FV_f \leftarrow null$
16: $\quad$ **end if**
17: $\quad Population \leftarrow Population \cup \langle \overline{P}, FV_p, FV_f \rangle$
18: $\quad$ **while** $size(Population) > MAX_POPULATION_SIZE$ **do**
19: $\quad\quad Population \leftarrow reducePopulation(Population)$
20: $\quad$ **end while**
21: **until** $\{\exists \langle \overline{P}, FV_p, FV_f \rangle \in Population \mid FV_f = MAX_FITNESS_VALUE\}$
22: **return** $\overline{P}$

Abbildung 4.1.: Pseudocode für die automatische Fehlerkorrektur eines Programmes. Aus einem fehlerhaften Programm und dessen Testfällen wird nach einer Korrektur gesucht. Die Funktion zur Berechnung des Fitnesswerts (calculateFitnessValue) ist in Abbildung 4.2 ersichtlich.

Als Input erwartet der Algorithmus das fehlerhafte Programm, die zugehörigen Testfälle mit mindestens einem fehlgeschlagenen Testfall und die Fehlerwahrscheinlichkeit je Statement im Programm.

In Zeile 1 wird zuerst das reduzierte (TS_p) und vollständige (TS_f) `TestSuiteSet` berechnet. Eine `TestSuite` umfasst alle Testfälle, die für ein Programm verfügbar sind, ein `TestSuiteSet` ist eine Teilmenge davon. Das reduzierte `TestSuiteSet` enthält nur eine geringe Anzahl an Testfällen, mindestens jedoch einen fehlgeschlagenen Testfall. Die Absicht hinter dem reduzierten `TestSuiteSet` ist, dass nicht bei je-

Input: Programm P
Input: TestSuiteSet TS
Output: Fitnesswert
1: $FV \leftarrow 0$
2: **for all** $T \in TS$ **do**
3: **if** $testCaseSucceeded(T)$ **then**
4: $FV \leftarrow FV + MAX_FITNESS_VALUE$
5: **else**
6: $FV \leftarrow FV + FAILED_FITNESS_VALUE$
7: **end if**
8: **end for**
9: $FV \leftarrow FV/size(TS)$
10: **return** FV

Abbildung 4.2.: Pseudocode zur Berechnung des Fitnesswertes für ein Programm mit einem `TestSuiteSet`. Im `TestSuiteSet` sind die Testfälle definiert, die auf das Programm ausgeführt werden. Aus dem Ergebnis der Testfälle wird der Fitnesswert berechnet.

der Fitnesswertberechnung alle Tests durchgeführt werden müssen und somit die Performance gesteigert wird. Das vollständige `TestSuiteSet` enthält alle Testfälle.

In den Zeilen 2-4 wird für das jeweilige `TestSuiteSet` der reduzierte (FV_p) und vollständige (FV_f) `Fitnesswert` berechnet. Die Berechnung des Wertes ist im Pseudocode (siehe Abbildung 4.2) hinterlegt. Das fehlerhafte Programm wird mit seinem reduzierten und vollständigen Fitnesswert in die leere Population eingefügt.

Die Zeilen 6 - 21 werden wiederkehrend ausgeführt, bis in der Population ein Programm vorhanden ist, dessen Fitnesswert (vom vollständigen `TestSuiteSet`) gleich dem MAX_FITNESS_VALUE ist.

In den Zeilen 6-8 wird zuerst ein Programm aus der Population gewählt und anschließend mutiert. Als `Mutationsoperatoren` werden unter anderem `DELETE`, zum Entfernen eines Statements und `CHANGE`, zum Ändern eines Statements, verwendet. Für das Mutieren sind die Fehlerwahrscheinlichkeiten je Statement wichtig, um Statements mit einer hohen Fehlerwahrscheinlichkeit öfter mutieren zu können, als solche mit niedrigen. Mit dem mutierten Programm wird der Fitnesswert auf Basis des reduzierten `TestSuiteSets` berechnet.

In der Zeile 9 wird überprüft, ob der Fitnesswert für das reduzierte `TestSuiteSet`, dem MAX_FITNESS_VALUE entspricht. Ist das der Fall, wird in Zeile 10 der Fitnesswert für das vollständige `TestSuiteSet` berechnet. Entspricht dieser auch dem MAX_FITNESS_VALUE, wurde eine Korrektur gefunden.

In Zeile 10 wird das `Final Repair` gebildet, wenn zuvor eine Korrektur gefunden wurde. Im `Final Repair` werden die Statements entfernt, die durch das Mutieren entstanden sind, aber nicht für die Korrektur benötigt werden. Es wird "toter" Code entfernt.

In den Zeilen 17-20 wird das mutierte Programm mit den Fitnesswerten in die Population eingefügt. Wurde dadurch die maximale Größe der Population überschritten (MAX_POPULATION_SIZE), wird sie auf diese vorgegebene Größe reduziert.

Zeile 22 liefert das korrigierte Programm zurück, wenn die Schleife über die Zeilen 5-21 terminiert.

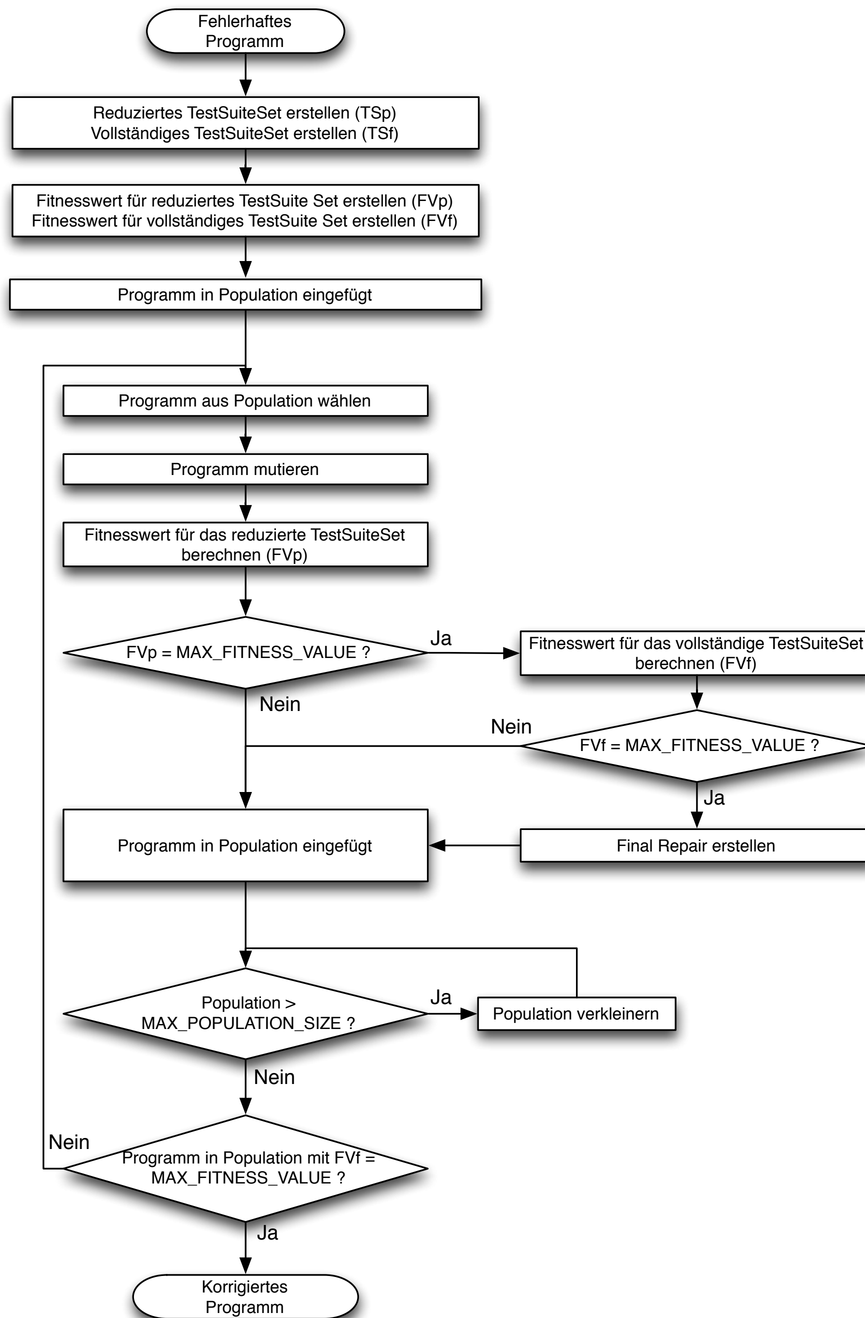


Abbildung 4.3.: Übersicht über den Programmablauf

4.3. Demonstration anhand eines Code-Beispiels

Anhand eines Beispiels soll demonstriert werden, wie dieser Algorithmus in der Praxis arbeitet. Als fehlerhaftes Programm wird `JTOPASv1` mit dem `Fehler Nr. 2` verwendet (siehe Kapitel 8.4).

Der Fehler liegt in der Klasse `de.susebox.java.io.ExtException` in der Funktion `ExtIOException`. Die Aufgabe der Funktion ist das Setzen von Klassenvariablen in Abhängigkeit von den Funktionsparame-

tern. Der Programmfehler besteht darin, dass die Zuweisungen der `_isWrapper` Klassenvariable vertauscht wurde (siehe Auflistung 4.1).

```
1  public ExtIOException(Exception ex, String fmt, Object[] args) {
2    super(fmt);
3    if (ex != null && fmt == null) {
4      _isWrapper = false;          // Fehler, korrekt: _isWarapper = true;
5    } else {
6     _isWrapper = true;            // Fehler, korrekt: _isWarapper = false;
7    }
8    _next = ex;
9    _args = args;
10 }
```

Auflistung 4.1: Fehlerhafter Programmabschnitt von JTOPASv1 Fehler Nr. 2

Für `JTOPASv1` sind 126 Testfälle verfügbar. Beim Ausführen aller Testfälle sind 3 fehlgeschlagen (siehe Auflistungen 4.2 bis 4.4). Diese fehlgeschlagenen Tests bilden das reduzierte `TestSuiteSet`. Fehlgeschlagene Testfälle werden mit 0,1 (FAILED_FITNESS_VALUE) bewertet, positive mit 1,0 (MAX_FITNESS_VALUE). Daher ergibt sich für das reduzierte `TestSuiteSet` ein Fitnesswert von $FV_p : \frac{3*0,1}{3} = 0,1$. Für das vollständige `TestSuiteSet` ergibt sich ein Fitnesswert von $FV_f : \frac{3*0,1+123*1,0}{126} = 0,98$. Das Programm wird mit beiden Fitnesswerten in die Population eingefügt.

Die fehlgeschlagenen Testfälle, aus denen das reduzierte `TestSuiteSet` für das Beispiel gebildet wird, sind im Einzelnen hier aufgelistet. Bei 2 Testfällen schlägt ein `Assert` fehl (siehe Auflistungen 4.2 und 4.3), beim dritten Testfall (siehe Auflistung 4.4) wird eine Exception verursacht. Ob eine Exception verursacht oder ein `Assert` fehlgeschlagen ist, hat auf den `Fitnesswert` keine Auswirkung. Beides wird als fehlgeschlagener Testfall behandelt.

```
1   public void testWrappedException() throws Throwable {
2      // prerequisitories
3      Class[]                   paraTypes = new Class[] { new Exception().getClass() };
4      String                    msg       = "This is an illegal argument.";
5      IllegalArgumentException  argEx     = new IllegalArgumentException(msg);
6
7      // construct the exception to test
8      Constructor    constr  = Class.forName(_classToTest).getConstructor(paraTypes);
9      ExceptionList ex       = (ExceptionList)constr.newInstance(new Object[] { argEx } );
10
11     // do the checks
12     assertTrue("rtEx: Wrapper exception not recognized.", ex.isWrapperException());
13     assertTrue("rtEx: Didn't retrieve the wrapped exception.", ex.nextException() == argEx);
14     assertTrue("rtEx: Messages not equal.", ((Exception)ex).getMessage().equals(argEx.getMessage()));
15  }
```

Auflistung 4.2: Testfall testWrappedException. Assert in Zeile 12 schlug fehl.

```
1   public void testNestedExceptions() throws Throwable {
2      // prerequisitories
3      Object[]                  objArray   = new Object[1];
4      String                    format     = "This is exception no {0} of class {1}.";
5      String                    msg        = "Message without format parameters.";
6      IllegalArgumentException  argEx      = new IllegalArgumentException(msg);
7      Class[]                   paraTypes  = new Class[] { new Exception().getClass(), msg.getClass(),
          objArray.getClass() };
8
9      // construct the exception to test
10     Constructor    constr   = Class.forName(_classToTest).getConstructor(paraTypes);
11     ExceptionList ex1       = (ExceptionList)constr.newInstance(new Object[] { argEx, format, new
          Object[] { new Integer(1), _classToTest } } );
12     ExceptionList ex2       = (ExceptionList)constr.newInstance(new Object[] { ex1,   format, new
          Object[] { new Integer(2), _classToTest } } );
13
14     // do the checks
15     assertTrue("ex1: False wrapper exception.", ! ex1.isWrapperException());
16     assertTrue("ex2: False wrapper exception.", ! ex2.isWrapperException());
17     assertTrue("ex1: Didn't retrieve the nested exception.", ex1.nextException() == argEx);
18     assertTrue("ex2: Didn't retrieve the first nested exception.", ex2.nextException() == ex1);
19     assertTrue("ex2: Didn't retrieve the second nested exception.", ((ExceptionList)ex2.nextException
          ()).nextException() == argEx);
```

```
20      assertTrue("ex1: Format not found.", ex1.getFormat() == format);
21      assertTrue("ex2: Format not found.", ex2.getFormat() == format);
22  }
```

Auflistung 4.3: Testfall testNestedExceptions. Assert in Zeile 15 schlug fehl.

```
1  public void testMessageFormatting() throws Throwable {
2      // prerequisitories
3      String    format    = "Class {0}, reason \"{1}\", user {2}.";
4      Object[]  paras     = new Object[] { _classToTest, "bad weather", "myself" };
5      Class[]   paraTypes = new Class[] { format.getClass(), paras.getClass() };
6
7      // construct the exception to test
8      Constructor   constr  = Class.forName(_classToTest).getConstructor(paraTypes);
9      ExceptionList ex      = (ExceptionList)constr.newInstance(new Object[] { format, paras } );
10
11     // do the checks
12     assertTrue("Format not found.", ex.getFormat() == format);
13
14     String str1 = MessageFormat.format(format, paras);
15     String str2 = ((Exception)ex).getMessage();
16     assertTrue("Formating failed. Expected \"" + str1 + "\", got \"" + str2 + "\".",
17                str1.equals(str2));
18  }
```

Auflistung 4.4: Testfall testMessageFormatting. Null Pointer Exception in Zeile 15 geworfen.

Generation 1 - Delete Operator

In der 1. Generation wird das letzte Statement (Zeile 9) durch den zufällig gewählten `DELETE` Operator gelöscht (siehe Auflistung 4.5). Die Testfälle des reduzierten `TestSuiteSet` werden ausgeführt, der Fitnesswert bleibt unverändert bei $FV_p : 0,1$. Das Programm wird mit dem Fitnesswert in die Population eingefügt.

```
1  public ExtIOException(Exception ex, String fmt, Object[] args) {
2    super(fmt);
3    if (ex != null && fmt == null) {
4      _isWrapper = false;
5    } else {
6     _isWrapper = true;
7    }
8    _next = ex;
9    ;                  // DELETE Operator. Statement wird entfernt bzw. mit Empty Statement ersetzt.
10 }
```

Auflistung 4.5: Programmabschnitt, nachdem das Statement in Zeile 9 durch den `DELETE` Operator entfernt wurde.

Generation 2 - Delete Operator

In der 2. Generation wird der ganze `if` Block entfernt (siehe Auflistung 4.6). Durch das Entfernen wird die Klassenvariable `_wrapper` nicht mehr verändert. Das hat zur Folge, dass der Wert nicht immer falsch zugewiesen wird. Von den 3 Testfällen schlagen nur noch 2 fehl. Der Fitnesswert des reduzierten `TestSuiteSet` verbessert sich auf $FV_p : \frac{2*0,1+1*1,0}{3} = 0,4$. Das Programm wird mit dem Fitnesswert in die Population eingefügt.

```
1  public ExtIOException(Exception ex, String fmt, Object[] args) {
2    super(fmt);
3    ;  // DELETE Operator
4    _next = ex;
5    ;
6  }
```

Auflistung 4.6: Programmabschnitt, nachdem das Statement in Zeile 3 durch den `DELETE` Operator entfernt wurde.

Generation 3 - Insert Operator

In der 3. Generation wird von einer anderen Stelle im Programm das korrekte `if` Statement eingefügt (siehe Auflistung 4.7). Von den bisherigen 3 Testfällen schlägt nur noch 1 Testfall fehl. Der Fitnesswert verbessert sich weiter auf $FV_p : \frac{1*0,1+2*1,0}{3} = 0,7$. Das Programm wird mit dem Fitnesswert in die Population eingefügt.

```
1  public ExtIOException(Exception ex, String fmt, Object[] args) {
2    super(fmt);
3    if (ex != null && fmt == null) {  // INSERT Operator
4      _isWrapper = true;
5    } else {
6      _isWrapper = false;
7    }
8    ;
9    _next = ex;
10   ;
11 }
```

Auflistung 4.7: Programmabschnitt, nachdem das korrekte Statement in Zeile 3 durch den `INSERT` Operator hinzugefügt wurde.

Generation 4 - Backcrossing Operator und Final Repair

In der 4. Generation wird der `BackCrossing` Operator auf das letzte Statement angewandt. Dabei wird das aktuell an dieser Position stehende Statement durch das ursprüngliche, an dieser Stelle stehende Statement, ersetzt (siehe Auflistung 4.8). Alle Testfälle des reduzierten `TestSuiteSet` werden positiv ausgeführt, der daraus resultierende Fitnesswert ist $FV_p : \frac{3*1,0}{3} = 1,0$.

```
1  public ExtIOException(Exception ex, String fmt, Object[] args) {
2    super(fmt);
3    if (ex != null && fmt == null) {
4      _isWrapper = true;
5    } else {
6      _isWrapper = false;
7    }
8    ;
9    _next = ex;
10   _args = args;  // BACKCROSSING Operator
11 }
```

Auflistung 4.8: Korrigiertes Programm, nachdem das Statement in Zeile 10 wieder durch den `BACKCROSSING` Operator in das ursprüngliche Statement geändert wurde.

Im nächsten Schritt werden alle Tests des vollständigen `TestSuiteSet` ausgeführt. Da alle 126 Tests positiv ausgeführt werden, ergibt das einen Fitnesswert von $FV_f : \frac{126*1,0}{126} = 1,0$ für das vollständige `TestSuiteSet`.

Nachdem der Fitnesswert sowohl vom reduzierten, als auch vom vollständigen `TestSuiteSet` $1,0$ ergeben hat, wird die `Final Repair` gebildet (siehe Auflistung 4.9). Dabei wird das Empty Statement entfernt. Die Fitnesswerte verändern sich dadurch nicht. Das `Final Repair` wird mit den beiden Fitnesswerten in die Population eingefügt.

Der Algorithmus terminiert und liefert als Lösung das korrigierte Programm (siehe Auflistung 4.8).

```
1  public ExtIOException(Exception ex, String fmt, Object[] args) {
2    super(fmt);
3    if (ex != null && fmt == null) {
4      _isWrapper = true;
5    } else {
6      _isWrapper = false;
7    }
8    _next = ex;
9    _args = args;
10 }
```

Auflistung 4.9: Final Repair für das korrigierte Programm. Unnötige Statements wurden entfernt.

Kapitel 5

Implementierung

5.1. Programmiersprache

Als Programmiersprache für die Implementierung wurde `JAVA` gewählt. Die Wahl fiel auf diese Programmiersprache aufgrund folgender Eigenschaften:

- Auf vielen Plattformen (Windows, Mac OSX, Linux) verfügbar
- Plattformunabhängig
- Sehr mächtige Relektion API
- Möglichkeit zum dynamischen Erzeugen, Verändern, Compilieren und Ausführen des Codes zur Laufzeit
- Programme mit Testfällen und vordefinierten Fehlern verfügbar, z.B. `Software-artifact Infrastructure Repository (SIR)`[12]

5.2. Entwicklungsumgebung

Als Programmierumgebung wurde `Eclipse` gewählt. Eclipse gilt als eine renommierte, stabile Plattform für die Java Entwicklung. Sie ist selbst in Java geschrieben und ebenso für viele Plattformen verfügbar. Für die laufende Sourcecode Verwaltung wurde `Subversion` verwendet.
Entwicklungsumgebung im Detail:

- **Hardware**: Intel Core i7-870 4x 2.93GHz, 8 GB RAM, 120 GB SSD
- **Betriebssystem**: Max OS X Lion 10.7.2
- **JAVA Version**: 1.6.0
- **Eclipse Version**: Indigo 3.7.0

5.3. Bibliotheken

Um das Rad nicht mehrmals neu erfinden zu müssen, wurde auf bestehende und erprobte Bibliotheken zugegriffen.

5.3.1. AST Manipulation

Bereits im `Master Projekt` wurden Möglichkeiten erprobt, um den Source Code automatisiert zu ändern und auszuführen. Dazu muss der Source Code automatisch interpretiert werden, um gezielte Änderungen darin durchführen zu können. Eine gängige Form der Repräsentation von Source Code ist der `Abstract Syntax Tree (AST)`; dabei wird der Source Code als Baum Struktur repräsentiert. Auf die einzelnen Knoten und Kanten kann nun einfach zugegriffen werden und Manipulationen können durchgeführt werden. Wenn die Änderungen abgeschlossen sind, kann aus dem `AST` wieder der Source Code erstellt, kompiliert und ausgeführt werden. `Eclipse` liefert dort schon eine entsprechende API mit; sie befindet sich allerdings nicht im Standardumfang und muss getrennt eingebunden werden [19, 18].

5.3.2. Eclipse JDT

Nachdem der Code geändert worden ist, muss er erneut kompiliert werden. Dabei ist wichtig, dass alles im Hauptspeicher durchgeführt wird. Müssten wir zuerst den geänderten Code auf die Festplatte schreiben, dort den Compiler starten und anschließend den Bytecode wieder von der Festplatte lesen, würde sich das in einer schlechten Performance niederschlagen. Des Weiteren muss sichergestellt sein, dass nur das oder die geänderten Dateien neu kompiliert werden müssen und nicht jedes Mal das gesamte Programm.

Es gibt zwar direkt in Java seit Version 1.5 die Möglichkeit den Java Compiler zur Laufzeit auf dynamisch generierten Code anzuwenden, jedoch muss bei dessen Verwendung immer das gesamte Programm neu kompiliert werden. Es sollte zwar laut Dokumentation die Möglichkeit geben, Code auch teilweise zu kompilieren, jedoch wurde man in der Praxis vom Gegenteil belehrt. Auch Entwicklerforen bestätigten diese Erfahrung. Es dürfte sich hier um einen BUG oder unvollständige Implementierung handeln.

`Eclipse Java Development Tools (JDT)` bieten ebenfalls einen JAVA Compiler an, der alle unsere Anforderungen unterstützt. Da dieser schon im `Master Projekt` gute Dienste erwiesen hat, wird er auch für diese Implementierung verwendet.

5.3.3. Apache log4

Ein umfangreiches und flexibles Logging erleichtert das Aufspüren von Fehlern während der Entwicklungsphase. Auch im laufenden Betrieb einer Software ist es notwendig, um im Fehlerfall den Hergang besser rekonstruieren zu können. Java selbst bietet seit Version 1.4 eine einfache Möglichkeit für das Logging an. Die `log4j` Bibliothek von Apache ist aber umfangreicher und wurde daher gewählt.

5.4. Spezielle Anforderungen an die Implementierung

5.4.1. Lose Koppelung der Komponenten

Die `JAVA Code Correction (JCC) Engine` besteht aus mehreren Komponenten (siehe Kapitel 6), die über Interfaces lose miteinander gekoppelt sind; damit soll ein einfaches Auswechseln der Komponenten ermöglicht werden. Wie später beschrieben, haben sich im Laufe der Entwicklung interessante Ideen entwickelt, die ein effektiveres Testen und/oder Kompilieren ermöglichen könnten. Durch die saubere Trennung der einzelnen Komponenten, die nur über definierte Interfaces miteinander verbunden sind, soll dies einfach möglich sein.

5.4.2. Threadsicherheit und parallele Verarbeitung

Selbst Smartphones sind heutzutage schon mit Dual Core CPUs ausgestattet. PCs haben inzwischen 2, 4 und 6 Cores und am Horizont tauchen die ersten 8 Core Rechner für den Consumer-Markt auf.

Durch diese Entwicklung wird auch von der Software verlangt, dass sie die neue Hardwaregeneration optimal ausnutzen kann um bestmögliche Performance zu erzielen; diese spielt bei der automatischen Code Korrektur eine entscheidende Rolle, daher wird auch von der Implementierung gefordert, dass sie mehrere CPUs unterstützen kann.

Ein wichtiger Schritt dahin ist, dass die einzelnen Komponenten Threadsicherheit gewährleisten und damit den parallelen Zugriff von mehreren Prozessen zugleich unterstützen. Diese Voraussetzung liefert eine gute Ausgangsposition um die Anwendung parallel ausführen zu können und eine gesteigerte Performance zu erzielen.

5.4.3. Ausgaben auf System.err

Java stellt mit `System.err` eine standardmäßige Ausgabe für Fehlermeldungen aus Exceptions bereit. Da durch die Mutationen und das Ausführen des fehlerhaften Programms viele Fehler hervorgerufen werden, würde diese Ausgabe regelrecht überschwemmt werden und sich mit Fehler der `JCC Engine` vermischen (sofern solche auftreten). Besonders während der Entwicklung der `JCC Engine` war es daher notwendig, die Fehlerausgabe des zu testenden Programms zu deaktivieren. Da das fehlerhafte Programm in einem eigenen Thread gestartet wurde, konnte diesem ein eigener `Errorhandler` mitgegeben werden, der in weiterer Folge diese Ausgaben unterdrücken konnte.

5.5. Einschränkungen aus der Implementierung

5.5.1. Aufrufe von System.exit()

Mit `System.exit()` stellt Java einen Befehl zur Verfügung, mit dem die Java VM sofort beendet wird. Führt nun die `JCC Engine` ein anderes Programm aus, in dem ein `System.exit()` ausgeführt wird, so wird dadurch auch die `JCC Engine` selbst beendet. Auch wenn meist im Codeabschnitt, der mutiert wird, kein `System.exit()` vorhanden ist und es auch im Normalfall bei der Testfallausführung nicht erreicht wird, so kann es doch durch `Mutationen` (insert oder swap) an Stellen eingefügt und danach direkt ausgeführt werden. Meine Recherchen diesbezüglich haben ergeben, dass es nicht möglich ist diesen `System Call` abzufangen und zu ignorieren.

Bei der späteren Evaluierung müssen alle zu testenden Programme dahingehend untersucht und angepasst werden. In den meisten Fällen kann dieser Befehl ohne weitere Folgen auskommentiert bzw. entfernt werden.

5.5.2. Eindeutiger Name für Testfälle

Da zu Beginn einmalig das unmodifizierte, fehlerhafte Programm mit allen Testfällen ausgeführt wird, um in weiterer Folge die Testfälle einschränken zu können, muss jeder Testfall eindeutig identifiziert werden können. Das geht im Umfeld von `Junit` am einfachsten über den Namen des Testfalls. Da eine Testsuite teilweise sehr komplex aufgebaut sein kann, und Testfälle teilweise auch dynamisch erstellt werden, wird das nicht immer gewährleistet. In einem solchen Fall müssen die Testfälle entsprechend angepasst werden, damit jeder Testfall eindeutig einen Namen zugewiesen bekommt.

Sollten bei einem zu testenden Programm dennoch mehrere Testfälle den selben Namen haben, so wird die `JCC Engine` mit einer entsprechenden Fehlermeldung beendet, da es in weiter Folge zu Problemen kommen kann.

5.5.3. Compile Fehler im fehlerhaften Programm

Da zu Beginn zuerst das Programm ohne Modifikation ausgeführt wird um die Testfälle zu klassifizieren, ist es notwendig, dass das zu untersuchende Programm kompilierbar und damit ausführbar ist. Programme, die nicht kompilierbar sind, können daher auch mit der `JCC Engine` nicht automatisch korrigiert werden.

Es wäre möglich, diese Einschränkung aus der Implementierung zu entfernen, sollte das für weitere Arbeiten notwendig sein. Dadurch könnten die Testfälle allerdings nicht automatisch eingeschränkt werden und es müsste zu Beginn zuerst nach einer kompilierbaren Version gesucht werden, um die Testfälle in weiterer Folge zu selektieren. Je nach Korrektur kann es aber dazu führen, dass nicht die optimalen Testfälle selektiert werden und ein korrektes Ergebnis schwer zu finden wäre. In der Regel sind Compile Fehler leicht zu finden und zu korrigieren. Der verwendete Algorithmus für Mutationen würde dazu neigen, die nicht kompilierbaren Code Fragmente zu entfernen.

5.5.4. Endlosschleife im fehlerhaften Programm

Wie schon im vorherigen Teil beschrieben, muss das fehlerhafte Programm kompilierbar und ausführbar sein. Sollten ein oder mehrere Testfälle nicht terminieren, können die Testfälle nicht selektiert werden. Die Implementierung des Programms erkennt Endlosschleifen in der Art, dass das Ausführen der Testfälle in ein voreingestelltes Timeout läuft. Die Testfälle werden allerdings im selben Thread ausgeführt, daher gibt es nur ein “alles oder nichts” bei der Ausführung. Die Entscheidung alle Testfälle in einem Thread auszuführen fiel zu Gunsten einer besseren Performance; diese würde sich in diesem Fall sehr verschlechtern, weil das Initialisieren der `JUnit` Engine eine nicht zu unterschätzende Zeit benötigen würde.

Da solche Fehler mit Endlosschleifen schwer zu finden sein können, sollten zukünftige Implementierungen darauf ausgelegt sein, auch mit diesen Fehlern arbeiten zu können. Es ist dabei notwendig einen Kompromiss zwischen Funktionalität und Performance zu finden.

5.6. Anforderungen an das zu untersuchende Programm

5.6.1. Programm

Zu korrigierende, fehlerhafte Programme müssen in `Java` geschrieben sein. Die Programme können externe Bibliotheken (jar Archive) einbinden. Sie müssen nicht eigenständig lauffähig sein, die Testfälle müssen aber ausgeführt werden können. Dadurch ist es auch möglich, Bibliotheken oder nur Teile von Programmen zu testen und korrigieren.

5.6.2. Testfälle

Um nach erfolgter Mutation das Programm bzgl. Fehlverhalten prüfen zu können, werden Testfälle benötigt. Es ist mindestens ein Testfall notwendig, der beim zu untersuchenden Programm fehl schlägt. Je mehr Testfälle verfügbar sind, desto bessere Ergebnisse können in der Regel erwartet werden.

Beim Testen eines `Individuums` kann auch nur auf einen Teil dieser Testfälle zugegriffen werden, um die Performance zu verbessern. Wird ein `Individuum` als Lösung präsentiert, muss sichergestellt sein, dass alle verfügbaren Testfälle positiv durchgeführt worden sind.

5.6.3. Fehlerlokalisierung

Ein Programm besteht in der Regel aus vielen tausend Zeilen Code; bei größeren Projekten geht es sogar in die Millionen. Einen oder mehrere Fehler in einem solchen Programm zu finden, indem einfach

willkürlich beliebiger Code geändert wird, würde zu keinem befriedigenden Ergebnis führen; der Suchraum wäre einfach zu groß. Das haben auch die zuvor betrachteten Veröffentlichungen (siehe Kapitel 3) gezeigt. Lokalisieren oder Eingrenzen von Fehlern ist ein verbreiteter Forschungszweig, zudem es mehrere Strategien gibt (siehe Kapitel 3.1).

Das Eingrenzen des Fehlers ist nicht Teil der `JCC Engine`. Um aber, wie vorher beschrieben, nicht jeden Teil des fehlerhaften Programms mutieren zu müssen, muss zusätzlich zum defekten Programm noch eine Liste mit Fehlerwahrscheinlichkeiten geliefert werden. Ist diese Information nicht vorhanden, wird jedem Statement im Code dieselbe Fehlerwahrscheinlichkeit zugewiesen.

Kapitel 6

Implementierung der Komponenten

6.1. Program Engine

Zu Beginn ist es wichtig, dass eine Struktur geschaffen wird, in der das zu untersuchende Programm zusammen mit seinen benötigten Ressourcen verwaltet werden kann. Des Weiteren müssen Änderungen am Programm (durch Mutationen) verwaltet werden können. Eine Übersicht über den Zusammenhang der einzelnen Komponenten liefert die Abbildung 6.1.

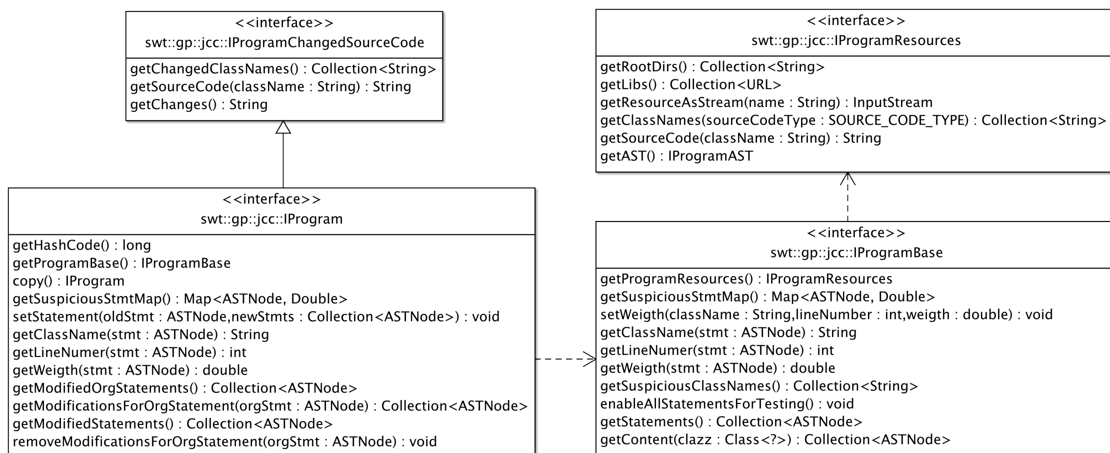


Abbildung 6.1.: Klassendiagramm für Program Engine

6.1.1. ProgramFactory

Während der Initialisierungsphase der `JCC Engine` wird durch Übergabe der Programm-Konfiguration (`IConfigProgram`) an die `ProgramFactory` (siehe Abb. 6.2) das zu untersuchende Programm geladen und zurückgeliefert.

Zusätzlich werden in der `ProgramFactory` bereits die Gewichtungen der einzelnen Statements hinsichtlich der Fehlerwahrscheinlichkeit zugewiesen. Diese liegen in eigenen Konfigurationsfiles dem zu untersuchenden Programm bei (siehe Kapitel 5.6.3).

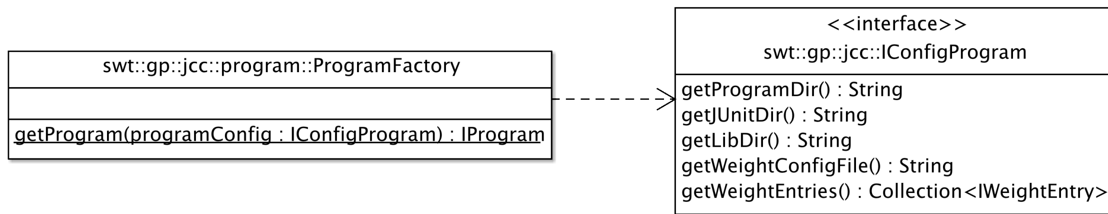


Abbildung 6.2.: Klassendiagramm für Program Factory

6.1.2. IProgramResources

Zur Laufzeit existiert nur eine Instanz der `IProgramResources`. In ihr wird der Quellcode, vom Programm benötigte Bibliotheken sowie weitere Ressourcen (Konfigurationsdateien, etc.) des Programms gespeichert. Der Quellcode unterteilt sich in:

- **Programm**: Quellcode des zu untersuchenden Programms.
- **JUnit**: Quellcode für die `Java Unit Tests`.

Bei der Initialisierung der `IProgramResources` wird auch der `Abstract Syntax Tree (AST)` für das zu untersuchende Programm erstellt. Diese Erstellung ist sehr zeitintensiv! Je nach Programm und zugrunde liegender Hardware werden dafür mehrere Sekunden benötigt. Es muss daher vermieden werden, dass der `AST` mehrmals zur Laufzeit erzeugt werden muss. Mit der Funktion `getAst()` kann auf den `AST` des Programms zugegriffen werden.

6.1.3. IProgramBase

Auch von der `IProgramBase` existiert zur Laufzeit nur eine Instanz. Neben den Programm-Ressourcen über `IProgramResources` hält es Funktionen für den Zugriff auf den `AST` des Programms bereit und verwaltet die Fehlergewichtung für die Statements. Diese werden zu Beginn von der `ProgramFactory` eingelesen und hier gespeichert.

6.1.4. IProgram

Das `IProgram` ist das abstrahierte Interface für das zu untersuchende Programm nach außen. Dabei kann es sich um das zu untersuchende Programm oder eine Programmvariante - durch Mutation erzeugt - handeln.

Es stellt die Funktionalität bereit, um eine Kopie des Programms zu erzeugen und damit weitere Änderungen durchzuführen. Dabei werden die gemeinsamen Ressourcen nur einmalig im Speicher abgelegt. Lediglich Geändertes zwischen dem Programm und der Programmvariante wird gespeichert. Dadurch ist es möglich, viele unterschiedliche Programmvarianten vorzuhalten und trotzdem den Speicherbedarf gering zu halten.

Des Weiteren bietet es Funktionen, um Änderungen am Programm durchzuführen oder diese Änderungen auszulesen.

6.1.5. IProgramChangedSourceCode

`IProgram` implementiert das Interface `IProgramChangedSourceCode`. Damit ist ein direkter Zugriff auf die, gegenüber dem zu Beginn geladenen Programm, geänderten Klassen möglich. Diese Funktionalität ist für ein effizientes Compilieren mit dem `DeltaCompiler` (siehe Kapitel 6.3.1) notwendig, um nur geänderte Klassen neu compilieren zu müssen.

6.2. Mutation Engine

Ein weiteres Modul der `JCC Engine` ist die `Mutation Engine` (siehe Abb. 6.3). Ihre Aufgabe ist es, ein übergebenes Programm zu mutieren und somit eine Programmvariante vom ursprünglichen Programm oder einer anderen Programmvariante zu erstellen. Dazu gibt es unterschiedliche `Mutationsoperatoren` auf die die `Mutation Engine` zugreifen kann. Welche Klassen und Statements mit welcher Wahrscheinlichkeit mutiert werden sollen, sind im `IProgramBase` und damit auch über das `IProgramm` zugänglich (siehe Kapitel 6.1).

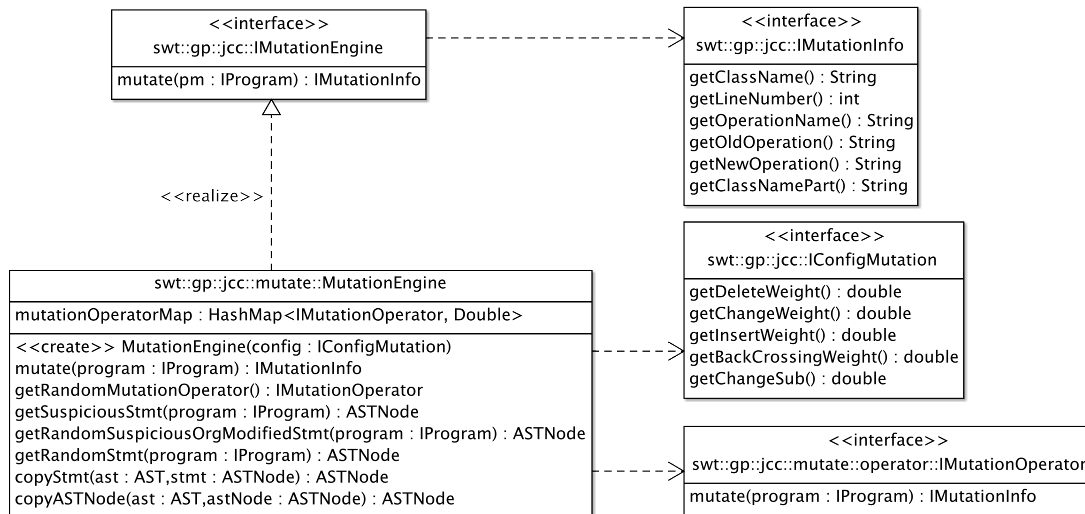


Abbildung 6.3.: Klassendiagramm für die Mutation Engine

6.2.1. IMutationEngine

Die `IMutationEngine` ist ein sehr einfaches Interface mit nur einer Funktion. Dadurch wäre es leicht möglich, sie durch eine andere Implementierung zu ersetzten; denn wie schon die Untersuchung von ähnlichen Arbeiten zu diesem Thema gezeigt hat, gibt es für das Mutieren unterschiedliche Ansätze (siehe Kapitel 3).

Zur Initialisierung der `Mutation Engine` wird ihr die Konfiguration für die einzelnen `Mutationsoperatoren` übergeben. Diese werden in der `Mutation Engine` mit einer Ausführungswahrscheinlichkeit registriert und anschließend für die Mutationen verwendet.

Wird die Funktion `mutate(IProgram)` in der `Mutation Engine` ausgeführt, werden folgende Schritte durchgeführt:

- Vom Programm wird eine Liste mit `verdächtigen` Statements geholt - das sind Statements mit einer Fehlerwahrscheinlichkeit > 0.0.
- Entsprechend den Fehlerwahrscheinlichkeiten wird durch eine Zufallszahl das Statement bestimmt, welches mutiert werden soll.
- Entsprechend der Ausführungswahrscheinlichkeiten der `Mutationsoperatoren` wird durch eine Zufallszahl ein `Mutationsoperator` ausgewählt.
- Der ausgewählte `Mutationsoperator` wendet die Mutation auf das ausgewählte Statement an.

- Sollte der ausgewählte `Mutationsoperator` keine Mutation durchführen können, wird für dasselbe Statement ein anderer Operator bestimmt. Sollte kein Operator auf das Statement anwendbar sein, wird ein anderes Statement gewählt.
- Um eine Information über die durchgeführte Mutation zu bekommen, liefert jeder `Mutationsoperator` eine `IMutationInfo` zurück, die dem Aufrufer der Funktion `mutate(IProgram)` returniert wird.

6.2.2. IMutationOperator

Jeder `Mutationsoperator` führt Änderungen an einem `IProgram` durch; natürlich nur sofern es mit seiner Implementierung möglich ist. Um das Programm zu mutieren, greift er auf die Funktionen des `IProgram` Interfaces zu (siehe Kapitel 6.1). Die Mutationen werden mit dem `AST` des Programms durchgeführt und im Programm gespeichert. Jeder `Mutationsoperator` returniert ein `IMutationInfo` Objekt; darin wird beschrieben, was mutiert worden ist. Sollte keine Mutation möglich sein, wird `null` returniert.

Delete Operator

Das übergebene Statement wird entfernt.

Change Operator

Das übergebene Statement wird durch ein beliebig anderes Statement im Programm ersetzt.

Insert Operator

Es wird ein beliebig anderes Statement im Programm ausgewählt um, dem Zufall nach, vor oder nach dem übergebenen Statement in den Code eingefügt zu werden.

Back Crossing Operator

Das übergebende Statement wird mit dem ursprünglichen Statement, das an dieser Stelle war, getauscht. Diese Idee stammt von einer ähnlichen Arbeit zu diesem Thema [26]. Die Idee dahinter ist, dass sich das Programm nicht zu weit von seinem ursprünglichen Programm entfernt bzw. um Mutationen rückgängig zu machen.

Sollte noch keine Änderung am Programm stattgefunden haben und daher `Back Crossing` keine Wirkung haben, wird `null` returniert.

Change Sub Operator

Es werden Teile des Statements durch passende Teile aus anderen Bereichen des Programms ersetzt.

- **Operatoren**: + ersetzt durch -, $>=$ ersetzt durch $>$, etc.
- **Zahlen**: 1 durch 0, 10 durch 33, etc.
- ...

Sollte kein Teil des Statements getauscht werden können (z.B. wenn es sich um einen Funktionsaufruf ohne Parameter handelt), wird `null` returniert.

6.2.3. IMutationInfo

Dient als Information über die durchgeführte Mutation der `Mutationsoperatoren` bzw. der `Mutation Engine`. Folgende Informationen sind enthalten:

- Der Name des Operators
- Der Name der Klasse, die geändert worden ist
- Die erste Zeile, ab der Änderungen durchgeführt worden sind
- Der alte, ursprüngliche Codeabschnitt
- Der neue, geänderte Codeabschnitt

6.3. Compile Engine

Nachdem das Programm geladen und geändert worden ist, muss es kompiliert werden, damit es anschließend ausgeführt bzw. getestet werden kann. Diese Aufgabe erledigt die `Compile Engine`. Da auch das Kompilieren von Programmen sehr oft durchgeführt wird, ist hier besonders auf Performance und Threadsicherheit zu achten.

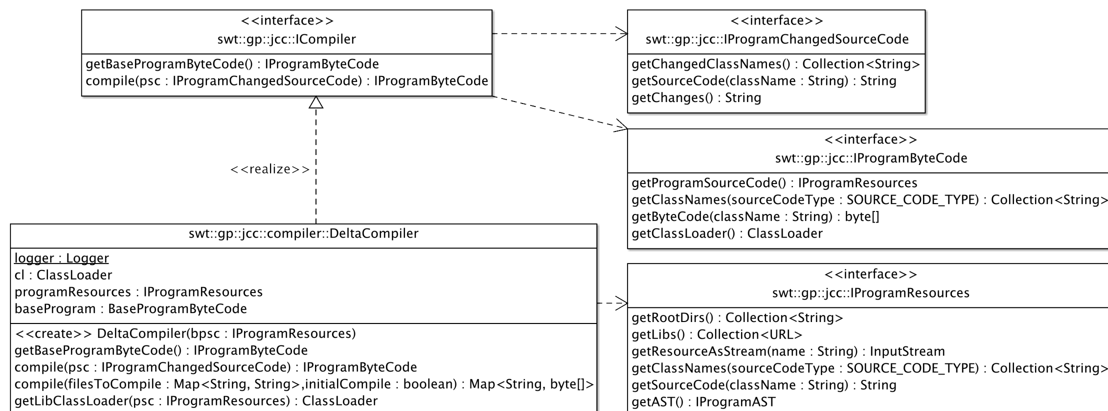


Abbildung 6.4.: Klassendiagramm für die Compile Engine

6.3.1. ICompiler

Das Interface des `Compilers` ist bewusst sehr schlank gehalten, um die Implementierung bei Bedarf leicht austauschen zu können. Um eine möglichst gute Performance zu erhalten, wurde er als `Delta Compiler` implementiert: anstelle jedes Mal den gesamten Quellcode neu übersetzen zu müssen, wird nur der, im Vergleich zum ursprünglichen Programm geänderte Quellcode neu übersetzt.

Bei der Initialisierung wird dem `DeltaCompiler` ein Objekt mit den `Programm Ressourcen` übergeben. Das Programm wird übersetzt und ist als `BaseProgramByteCode` verfügbar. Wird nun ein Programm durch das Interface `IProgramChangedSourceCode` übergeben, so werden nur noch die geänderten Klassen übersetzt. Der `IProgramByteCode`, der returniert wird, ist also eine Mischung aus den während der Initialisierung übersetzten Klassen und den geänderten Klassen.

Das Kompilieren wird durch die `Eclipse JDT` (siehe Kapitel 5.3.2) erledigt. Als Ergebnis liefert der Compiler ein Objekt vom Type `IProgramByteCode`. Sollte das Compilieren aufgrund eines Fehlers fehlschlagen, wird `null` returniert. Compiler Warnings werden ignoriert, können aber im Logging ausgegeben werden.

6.3.2. IProgramByteCode

Durch das Interface `IProgramByteCode` wird der Zugriff auf den Byte Code des Programms ermöglicht. Zusätzlich liefert er die Referenzen auf die `IProgramResources` mit. Diese werden beim späteren Ausführen des Codes benötigt. Selbes gilt für einen speziellen `ClassLoader`.

6.4. Test Engine

Eines der umfangreichsten und komplexesten Module der `JCC Engine` ist sicherlich die `Test Engine` (siehe Abbildung 6.5). Beim Testen muss der Code ausgeführt werden und das sollte so effizient wie möglich passieren. Da sich das Programm, das getestet wird, aber bei jedem Durchgang ändert, stellt das eine Herausforderung dar. Des Weiteren dauert das Ausführen von Testfällen an sich eine bestimmte Zeit. Daher ist es wichtig, so wenig Testfälle wie möglich, aber so viele wie nötig, auszuführen. Die Selektion der Testfälle hat allerdings einen entscheidenden Einfluss auf das Endergebnis: ob, welche und wie schnell eine Lösung gefunden werden kann.

Zusätzlich muss bei der Ausführung unter anderem mit folgenden Unannehmlichkeiten gerechnet werden:

- Unterschiedliche Laufzeiten von Testfällen
- Testfälle, die in einer Schleife hängen und endlos laufen würden
- Unterschiedliche Testfälle mit selben Namen und Bezeichnungen (siehe Kapitel 5.5.2)
- Funktion innerhalb des zu testenden Programms, die nach außen in die `JCC Engine` wirken:
  - `System.exit()` Aufrufe - diese können die gesamte `JCC Engine` zum Terminieren zwingen.
  - Ausgaben auf System.err - Damit vermischen sich Fehler vom untersuchenden Programm mit der `JCC Engine`.
  - Threads, die innerhalb des Programms oder der Unit Tests erzeugt werden, können wenn sie nicht sauber beendet werden, das Terminieren von Threads in der `JCC Engine` behindern.

Viele dieser Dinge wurden bei der Implementierung berücksichtigt. Bestehende Einschränkungen sind in Kapitel 4 angeführt.

6.4.1. ITestEngine

Die `ITestEngine` wird mit dem `IProgramByteCode` von dem Originalprogramm initialisiert. Mit dem `IProgramByteCode` wird eine `ITestSuite` erzeugt. Mit der `Test Engine` können beliebig viele `ITestSuiteSet` Objekte erzeugt werden. Mit einem solchen `ITestSuiteSet` und einem compilierten Programm als `IProgramByteCode` kann mit der `Test Engine` ein Testlauf durchgeführt werden. Als Ergebnis wird ein `ITestResults` Objekt returniert; sollte das Testen fehlschlagen, z.B. aufgrund eines Timeouts, wird `null` returniert.

Das Ausführen der `Unit Tests` passiert in einem eigenen2 Thread, dem eine spezielle Threadgruppe zugewiesen wird. Dadurch ist es möglich, die vorgegeben Timeouts zu überwachen und einzuhalten. Sollten die `Unit Tests` nicht in der vorgegebenen Timout Zeit terminieren, so wird der Thread abgebrochen.

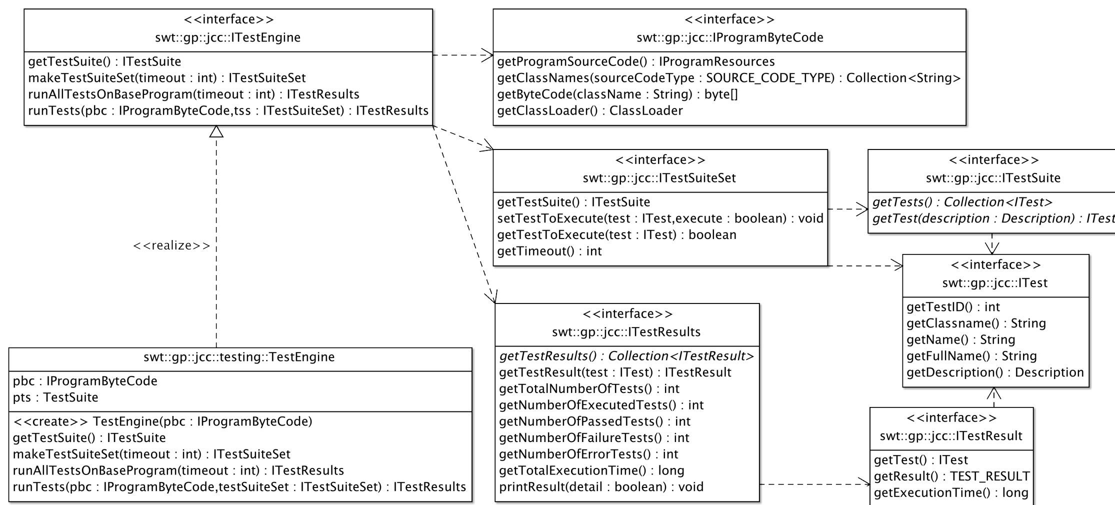


Abbildung 6.5.: Klassendiagramm für die Test Engine

Zusätzlich muss noch überprüft werden, ob in der Threadgruppe weitere Threads vorhanden sind. Es wäre möglich, dass das getestete Programm oder die `Unit Tests` weitere Threads angelegt haben. Sollte das der Fall sein, müssen auch diese terminieren oder terminiert werden. Würde das nicht erfolgen, würden sich immer mehr tote Threads ansammeln und die `JCC Engine` würde instabil werden.

6.4.2. ITestSuite

Es existiert eine `ITestSuite` je `ITestEngine`. Aus dem, bei der Initialisierung übergebenen Programm, werden alle `Unit Tests` geladen und bereitgestellt. Hier ist es auch wichtig, dass die Testfälle über eindeutige Namen verfügen, da diese ansonsten nicht getrennt referenziert werden können (siehe Kapitel 4).

6.4.3. ITestSuiteSet

Einer `ITestSuiteSet` wird bei der Initialisierung eine `ITestSuite` mit einem Timeout übergeben. Das Timeout bestimmt die maximale Ausführungszeit für die Testfälle und wird in Millisekunden angegeben. Zusätzlich zum Timeout werden in der `ITestSuiteSet` auch die Testfälle der `ITestSuite` verwaltet, die ausgeführt werden sollen.

Durch diese Art der Implementierung lassen sich folgende Szenarien später realisieren:

- Es gibt ein `ITestSuiteSet`, in dem alle Testfälle ausgeführt werden; das Timeout wird sehr hoch gesetzt. Das Ergebnis ist die Ausführungszeit der einzelnen Tests, die gesamte Ausführungszeit und der Ausgang der einzelnen Tests (fehlgeschlagen oder erfolgreich durchgeführt).
- Es gibt ein kleines `ITestSuiteSet`, in dem nur fehlgeschlagene Tests durchgeführt werden und zufällig einige erfolgreich durchgeführte.
- ...

Je nach Implementierung des Algorithmus wird das jeweilige `ITestSuiteSet` verwendet. Das `ITestSuiteSet` kann zur Laufzeit geändert werden um mehr oder weniger Tests durchzuführen. Somit ist ein flexibles Selektieren der auszuführenden Testfälle, wie in Kapitel 5.6.2 beschrieben, möglich.

6.4.4. ITest

Ein `ITest` ist die kleinste Einheit in dem gesamten Konstrukt und beschreibt den einzelnen Test in der `ITestSuite`.

6.4.5. ITestResult

Nach dem Ausführen eines `ITestSuiteSets` auf einen `IProgramByteCode` wird für jeden ausgeführten `ITest` ein `ITestResult` returniert. Zum einen erhält man die Ausführungszeit des Test und zum anderen den Status. Der Status kann sein:

- **OK**: Der Testfall wurde erfolgreich ausgeführt.
- **FAILURE**: Der Testfall hat einen Defekt (Assert) hervorgerufen; die erwartete Ausgabe entspricht nicht der berechneten Ausgabe.
- **ERROR**: Der Testfall hat eine unerwartete und unbehandelte Ausnahme (Exception) hervorgerufen.

6.4.6. ITestResults

`ITestResults` sind die Summe aller `ITestResults`, die nach einem Testlauf der `Test Engine` zur Verfügung stehen. Zusätzlich stehen noch Funktionen bereit, die das Ergebnis verdichten:

- Anzahl der Testfälle
- Anzahl der ausgeführten Testfälle
- Anzahl der Testfälle mit dem Status OK, FAILURE oder ERROR
- Gesamte Ausführungszeit für das Durchführen aller Tests

6.5. Fitness Function

Die Aufgabe der `Fitness Function` ist das Bewerten des Testergebnisses. Ausgehend von dieser Bewertung wird später entschieden, wie mit dieser Programmversion umgegangen wird. So ist es möglich, dass die Programmversion weiter getestet wird (mit mehr aktivierten Testfällen), ausgeschieden wird oder in der Population verbleibt. Auch dient die Fitnessfunktion zum Erkennen einer Programmversion, die eine mögliche Lösung ist. Der Zusammenhang der Schnittstellen ist in Abbildung 6.6 ersichtlich.

6.5.1. IFitnessFunction

Initialisiert wird die `IFitnessFunction` mit Parametern aus `IConfigFitnessValue`. Entsprechend diesen Werten wird der `Fitnesswert` für ein Testergebnis `ITestResults` berechnet. Sollte das Compilieren nicht möglich gewesen sein, oder das Testen in ein Timeout gelaufen sein, ist eine direkte Berechnung des `Fitness Value` nicht möglich. Dafür stehen zwei eigene Funktionen zur Verfügung, um auch diese Tests bewerten zu können.

Der Fitnesswert ergibt sich aus folgender Formel:

$$F = \frac{T_s * F_s + T_e * F_e + T_f * F_f}{T_s + T_e + T_f} \tag{6.1}$$

F .... Fitnesswert
T_s, T_e, T_f ... Anzahl der Tests die den Status `success`, `error` oder `failure` hatten
F_s, F_e, F_f ... Der Fitnesswert für den Status `success`, `error` oder `failure`

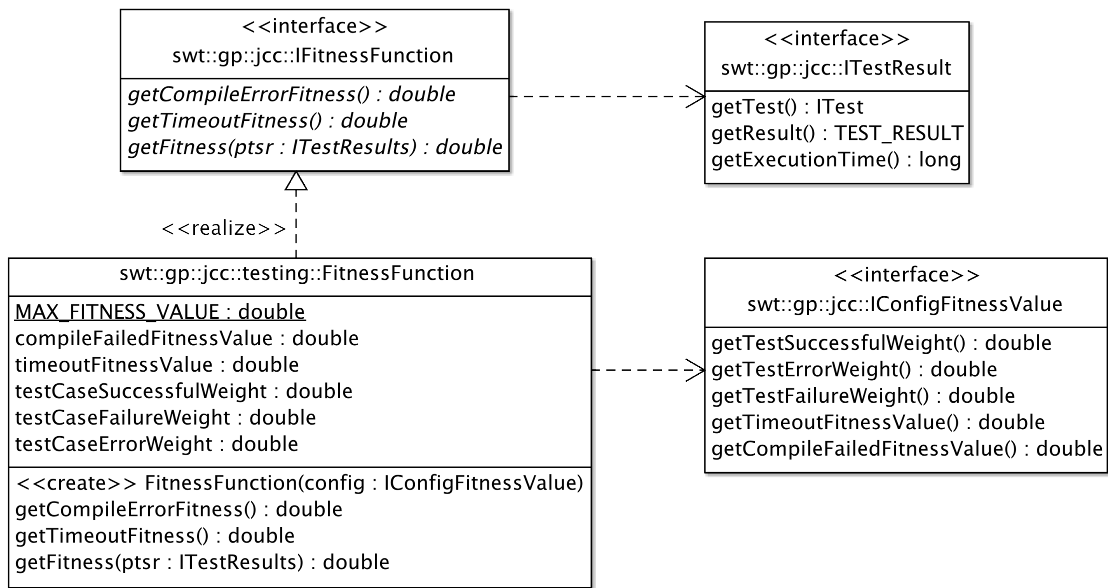


Abbildung 6.6.: Klassendiagramm für die Fitness Funktion

6.6. Population

In der `Population` werden die unterschiedlichen Varianten des zu untersuchenden Programms gespeichert. Diese Varianten sind durch das Interface `IIndividuum` ansprechbar (siehe Abbildung 6.7). In der Population wird die Entscheidung getroffen, welche `Individuen`, und damit auch Programmvarianten, in der Population bleiben und welche ausgeschieden werden. Zusätzlich wird der Zusammenhang zwischen den einzelnen Individuen gespeichert; so kann später nachvollzogen werden, wie das jeweilige Individuum entstanden ist.

Die Population wählt auch das nächste Individuum aus, das mutiert werden soll, und liefert eine Kopie davon zurück, um es mutieren zu können.

6.6.1. IPopulation

Die `Population` wird mit den Konfigurationsdaten `IConfigPopulation` und dem zu untersuchenden Programm `IProgram` initialisiert. Neben dem Zugriff auf die, in der Population verwalteten, `Individuen` bietet die `Population` auch Funktionen um ein neues `Individuum` aus der Population heraus zu erstellen und solche auch wieder in die Population einzufügen.

Neues Individuum erstellen

Wird mit `makeNewIndividuum()` ein `Individuum` zum Mutieren angefordert, wird je nach Parametrierung durch Zufall das ursprüngliche Programm oder eine Programmvariante ausgewählt. Das Programm oder die Programmvariante wird kopiert und als `IIndividuum` returniert (siehe Abbildung 6.8).

Individuum in die Population einfügen

Nachdem das Programm des `Individuums` mutiert wurde, wird es mit `addIndividuum(IIndividuum)` der `Population` hinzugefügt. Nach jedem Hinzufügen wird überprüft, ob die Größe der `Population` den

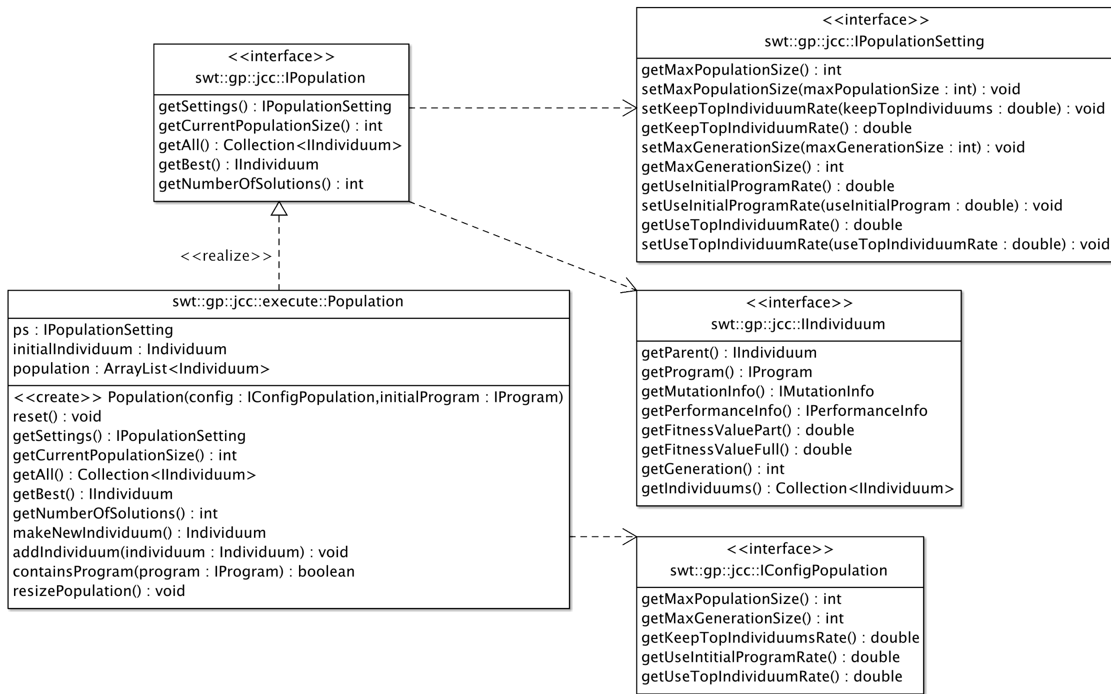


Abbildung 6.7.: Klassendiagramm für die Population

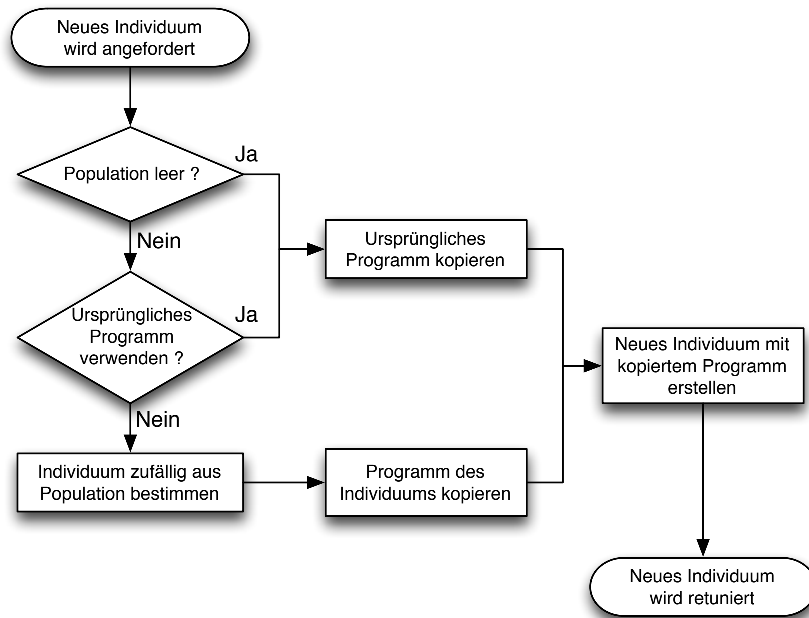


Abbildung 6.8.: Ablauf bei der Erstellung eines neuen Individuums

parametrierten Wert überschritten hat; ist das der Fall, wird die `Population` wieder auf den parametrierten Wert reduziert (siehe Abbildung 6.9). Beim Reduzieren der Population werden folgende Schritte durchgeführt:

- Es werden zufällig zwei Individuen gewählt, die einen Fitnesswert < dem maximalen Fitnesswert haben.
- Das Individuum mit dem kleineren Fitnesswert scheidet aus der Population aus. Sollten beide den selben Wert haben, dann wird das Erste ausgeschieden.
- Ist die Größe der Population noch immer größer als die über Parameter vorgegebene Größe, so werden die vorherigen Schritte wiederholt.

Dieses Auswahlverfahren hat den Vorteil, dass auch `Individuen` mit schlechten Fitnesswerten eine Chance haben in der `Population` zu verbleiben. Des Weiteren wird vermieden, das `Individuen` mit dem höchst möglichen Fitnesswert ausgeschieden werden.

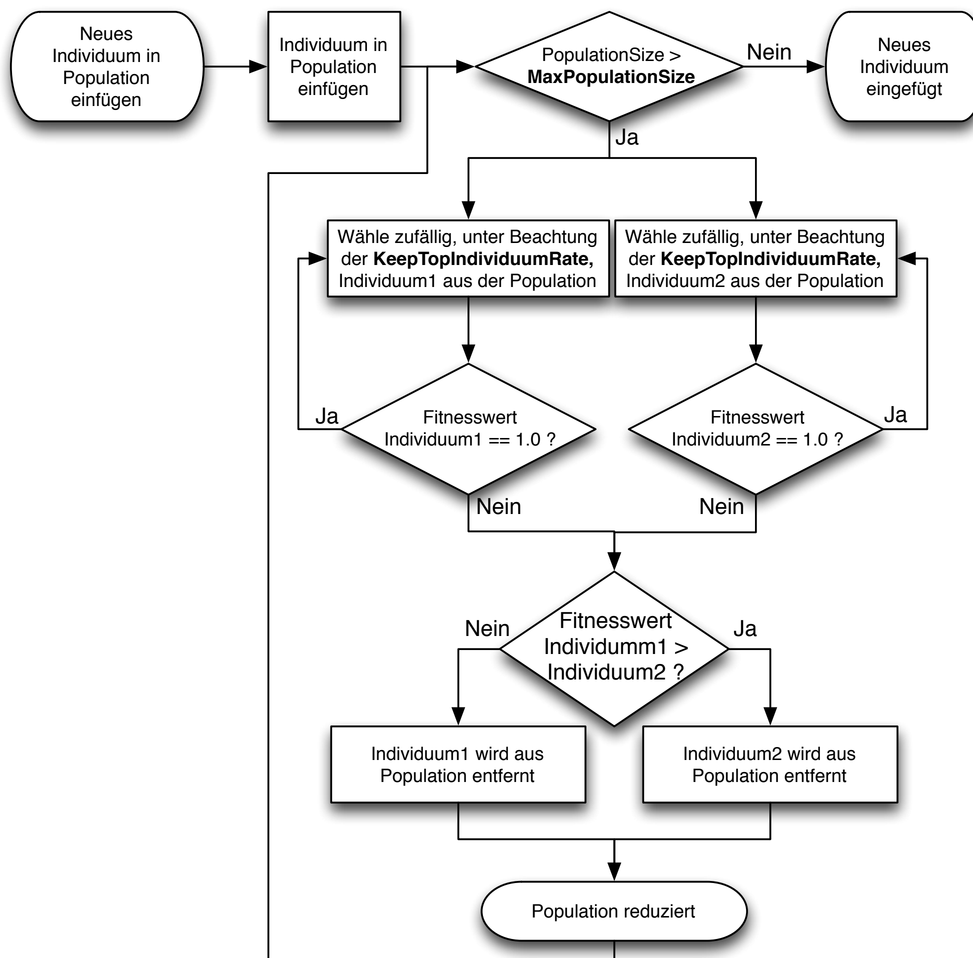


Abbildung 6.9.: Ablauf beim Hinzufügen eines neuen Individuums zur Population

6.6.2. IIndividuum

Objekte innerhalb der Population implementieren das Interface `IIndividuum`. Jedes Individuum repräsentiert ein `IProgram` innerhalb der Population. Zusätzlich zum Programm werden noch folgende Information gespeichert:

- **Parent**: Individuum, von dem dieses Individuum kopiert worden ist. Sollte es sich um das ursprüngliche Programm handeln, so ist dieser Wert `null`.

- **MutationInfo**: Information über die Änderungen in dieser Programmversion (siehe Kapitel 6.2.3). Sollte es sich um das ursprüngliche Programm handeln, ist dieser Wert `null`.
- **FitnessValuePart, FitnessValueFull**: Die Fitnesswerte für das reduzierte bzw. vollständige `TestSuiteSet`. Fitnesswerte für die ganze TestSuite werden nur erzeugt, wenn der `FitnessValuePart` Wert den maximalen Wert hat.
- **Generations**: Gibt an, in welcher Generation sich das Individuum bzw. die Programmvariante befindet; d.h. wie viele Vorgänger es hat bzw. wie oft es zuvor mutiert worden ist.

6.6.3. IPopulationSetting

Da die Parameter der `Population` recht umfangreich sind, wurde ihnen ein eigenes Interface zugewiesen. In der `IPopulationSetting` werden folgende Parameter verwaltet:

- **MaxPopulationSize**: Maximale Anzahl an Individuen, die in der Population gehalten werden.
- **KeepTopIndividuumRate**: Anzahl der am höchsten gereihten Individuen (entsprechend dem Fitnesswert), die vor dem Ausscheiden aus der Population gefeit sind. Werte zwischen 0,0 und 1,0 sind erlaubt.
- **MaxGenerationSize**: Maximale Anzahl der Mutationen des ursprünglichen Programms. Hat ein Individuum den Wert erreicht, wird es von der `Population` nicht mehr als Kandidat für weiteres Mutieren ausgegeben.
- **UseInitialProgramRate**: Wahrscheinlichkeit, dass das ursprüngliche Programm anstelle einer Programmvariante zum Mutieren verwendet wird. Werte zwischen 0,0 und 1,0 sind erlaubt.
- **UseTopIndividuumRate**: Gibt an, ob Individuen in der Population mit einem höheren Fitnesswert öfter weiter mutiert werden sollen, als solche mit einem niederen. Werte zwischen 0,0 und 1,0 sind erlaubt.

6.7. Execution Engine

Der Einstiegspunkt der `JCC Engine` ist die `Execution Engine`. Von hier aus werden alle anderen Module gesteuert. Auch auf das Ergebnis kann über die `Execution Engine` zugegriffen werden.

6.7.1. IExecutionEngine

Initialisiert wird die `Excecution Engine` mit den Konfigurationsparametern über das Interface `IConfig` und mit der Anzahl der Threads, auf die die Arbeit aufgeteilt werden soll. Die Threads sind besonders für die Performance Entwicklungen auf unterschiedlicher Hardware wichtig. Über die Konfiguration wird das gesamte Verhalten der `JCC Engine` gesteuert. Während der Initialisierung der `Execution Engine` werden die Module `Program Engine`, `Mutation Engine`, `Compile Engine`, `Test Engine` und die `Population` initialisiert.

Je Thread wird ein Prozess in der `Execution Engine` gestartet. Die Abläufe sind in der Abbildung 6.11 ersichtlich und im Detail hier beschrieben.

1. Aus der `Population` wird ein `Individuum` für die Mutation bestimmt.
2. Das `Programm` des `Individuums wird mutiert`
3. Es wird überprüft, ob das mutierte Programm schon als Individuum in der Population vorhanden ist. Ist es vorhanden, wird das Programm verworfen und es wird wieder bei Punkt 1 fortgesetzt. Somit müssen keine teuren Compile und Test Schritte auf dem Programm ausgeführt werden, obwohl das Ergebnis schon bekannt ist.

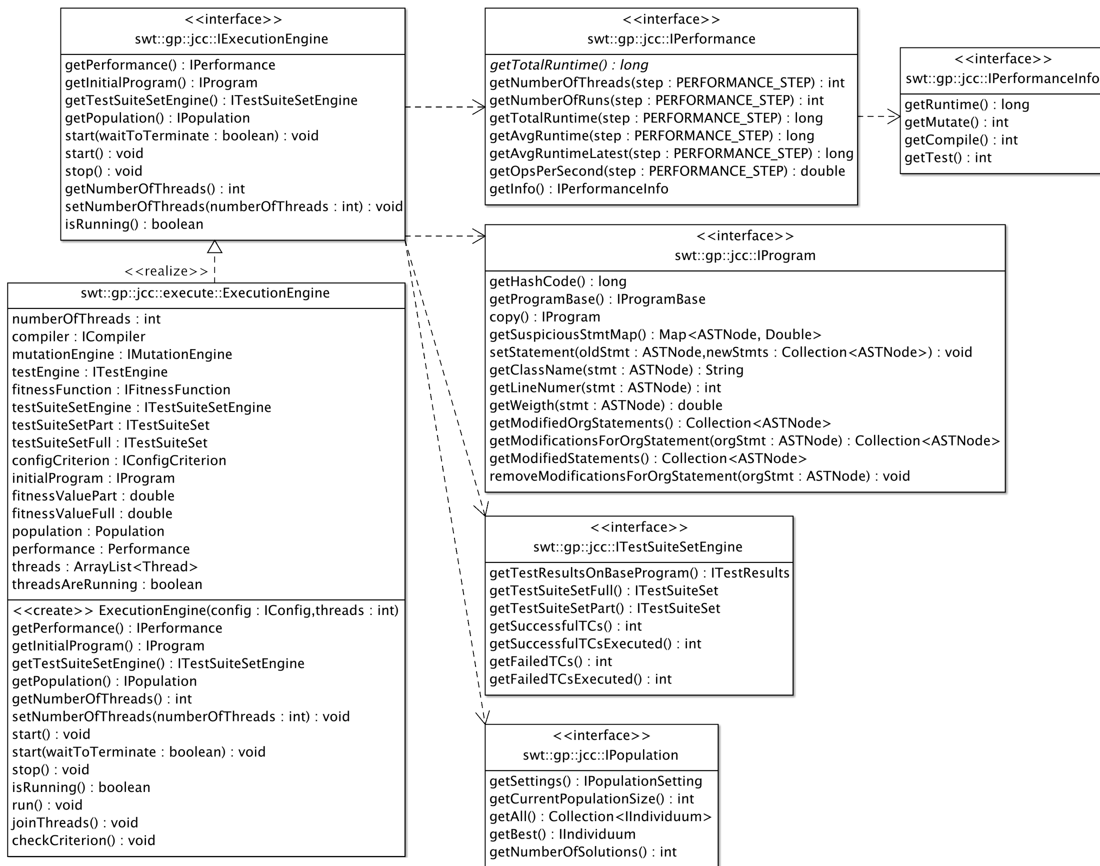


Abbildung 6.10.: Klassendiagramm für die Execution Engine

4. Das Programm wird compiliert.
5. Sollte das Compilieren fehlschlagen, wird das Programm entsprechend bewertet und als `Individuum` in die Population eingefügt. Es wird bei Punkt 1 fortgesetzt.
6. Das Compilieren war erfolgreich.
7. Das Programm wird mit dem `reduzierten TestSuiteSet` getestet.
8. Sollte das Testen aufgrund eines Timeouts abgebrochen werden, wird das Programm entsprechend bewertet und als `Individuum` in die `Population` eingefügt. Es wird bei Punkt 1 fortgesetzt.
9. Das Testen mit dem `reduzierten TestSuiteSet` war erfolgreich.
10. Es wird der Fitnesswert für das `reduzierte TestSuiteSet` berechnet.
11. Sollte der Fitnesswert unter dem maximalen Fitnesswert liegen, wird das Programm als `Individiuum` in die `Population` eingefügt. Es wird bei Punkt 1 fortgesetzt.
12. Punkte 7 - 11 werden mit dem `vollständigen TestSuiteSet` durchgeführt.
13. Sollte der Fitnesswert für das `vollständige TestSuiteSet` maximal sein, wird auf das Programm die Funktion zum Erzeugen der `Final Repair` angewendet.
14. Das Programm wird als `Individuum` entsprechend bewertet und in die `Population` eingefügt.
15. Abbruchkriterien (siehe Kapitel 7.1.2) werden überprüft, ggf. terminiert die Schleife hier.
16. Es wird bei Punkt 1 fortgesetzt.

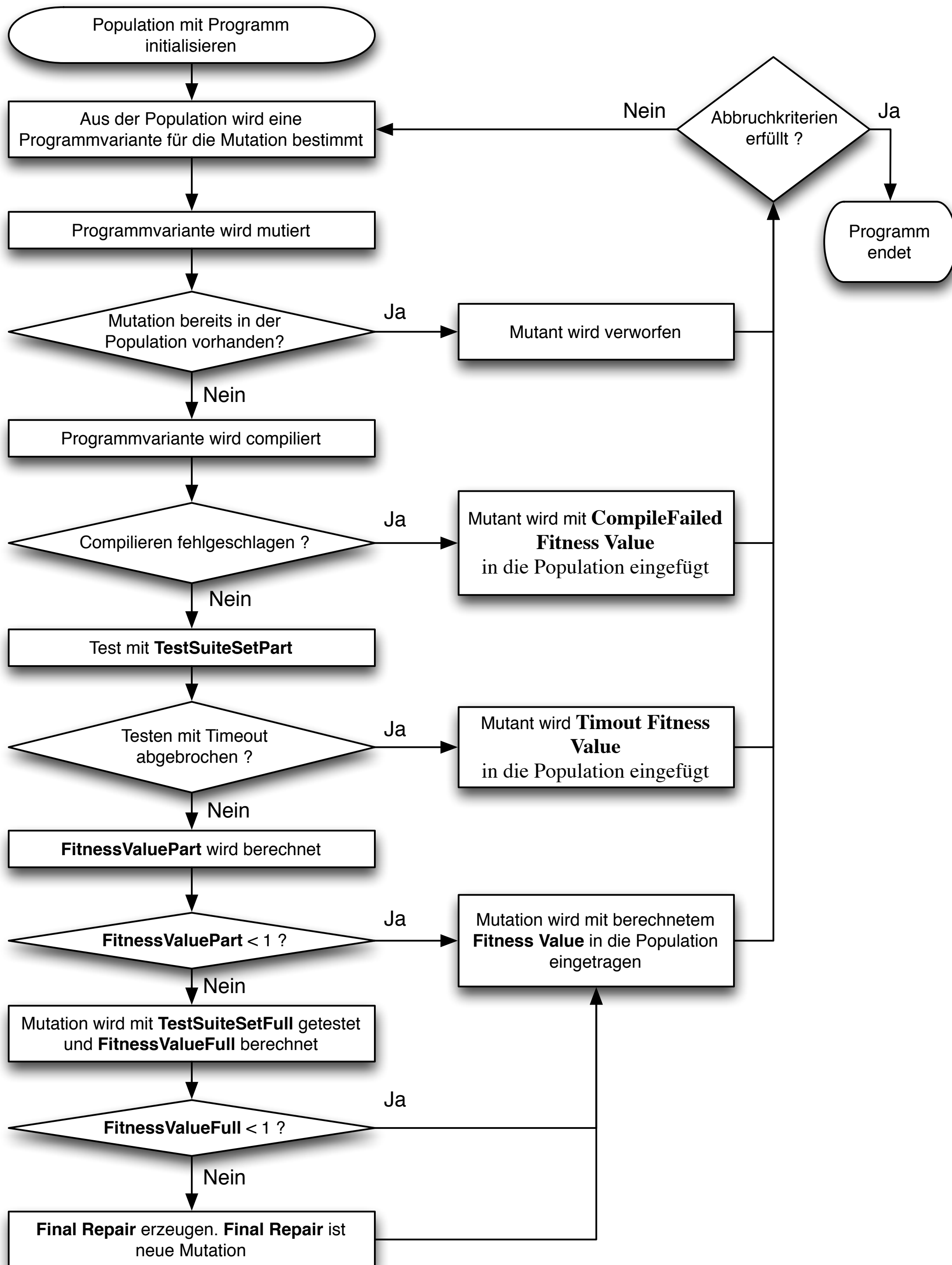


Abbildung 6.11.: Ablauf innerhalb der Execution Engine

6.7.2. ITestSuiteSetEngine

Die `TestSuiteSet Engine` verwaltet die beiden `TestSuiteSets`. Dabei handelt es sich um:

- **TestSuiteSetPart**: Ein `reduziertes TestSuiteSet`, bei dem je nach Parametrierung nur gewisse Testfälle ausgeführt werden (siehe Kapitel 6.8).
- **TestSuiteSetFull**: Das `vollständige TestSuiteSet`, bei dem alle Testfälle ausgeführt werden.

Damit diese beiden `TestSuiteSets` erzeugt werden können, müssen zuerst alle Tests auf das ursprünglich zu untersuchende Programm angewandt werden. Diese Daten können auch über diese Schnittstelle abgefragt werden.

6.7.3. Abbruchkriterien

Die `Abbruchkriterien` werden über das Interface `IConfigCriterion` parametriert. Nach jedem Durchlauf der `Execution Engine` werden sie überprüft und die Verarbeitung der `Execution Engine` wird beendet. Für eine Beschreibung der einzelnen Parameter siehe Kapitel 6.8.

6.7.4. Reduzierung zur Final Repair

Wenn eine Lösung gefunden wird, so spricht man von einer `Initial Repair`; diese ist das, durch Mutationen veränderte, ursprüngliche Programm. Die Lösung kann daher Mutationen enthalten, die für die Lösung überhaupt nicht relevant sind, den Code nur unnötig verlängern und im schlimmsten Fall eine weitere Fehlerquelle darstellen können. Um dem entgegenzuwirken, wird nach dem Finden der `Initial Repair` die `Final Repair` gesucht. Die `Final Repair` ist die `Initial Repair`, reduziert um die überflüssigen Änderungen am Programm.

In der `JCC Engine` wird das wie folgt implementiert:

- In einer Schleife wird nacheinander jedes geänderte oder neue Statement durchlaufen.
- Das geänderte oder zusätzliche Statement wird zurückgesetzt bzw. entfernt.
- Der Fitnesswert wird für das dadurch neu entstandene Programm berechnet.
- Hat sich der Fitnesswert nicht verschlechtert, so bleibt das Statement entfernt.
- Ansonsten wird die Änderung rückgängig gemacht und das nächste Statement wird überprüft.

Nachdem jedes geänderte Statement auf seine Notwendigkeit für die Lösung geprüft worden ist, erhält man ein Programm ohne unnötige, neue Statements.

Es handelt sich dabei allerdings noch nicht um eine `Minimal Repair`. Es kann immer noch vorkommen, dass ein neues Statement ein anderes Statement, das nicht verändert worden ist, unnötig werden lässt. Um dies zu vermeiden, müsste man den Algorithmus erweitern. Siehe dazu z.B. `Delta Debbugging` [28, 29, 27].

6.7.5. IPerformance

Besonders während der Implementierung war es sehr wichtig, die Performance der `JCC Engine` im Auge zu behalten. Dazu wurde das Interface `IPerformance` geschaffen, mit dem gewisse Daten über die Performance zur Laufzeit oder auch am Ende abgefragt werden können.

Bei diesen Daten handelt es sich um:

- Anzahl der Mutationen, Kompilierungen und Tests

- Die durchschnittliche Zeit für Mutieren, Kompilieren und Testen
- Die durchschnittliche Zeit für Mutieren, Kompilieren und Testen der letzten 10 Durchgänge
- Die gesamte Zeit für Mutieren, Kompilieren und Testen
- Die gesamte Laufzeit

Diese Daten werden auch als Abbruchkriterium herangezogen. Zum Beispiel die Verarbeitung beenden, wenn 100.000 Mutationen erzeugt worden sind und keine Lösung gefunden worden ist.

6.8. Configuration

Die bisher beschriebenen Module werden meist über Objekte mit speziellen Interfaces für die Konfiguration parametriert. Diese werden alle unter einem Interface `IConfig` gebündelt. Somit soll es möglichst einfach sein, die `JCC Engine` mit bestimmten Parametern auszuführen. Die Daten können dann beliebig über ein Konfigurationsfile, die `JCC GUI` oder, wie bei der Evaluierung verwendet, von einer Datenbank bereitgestellt werden.

Abbildung 6.12 gibt einen Überblick über den Zusammenhang der Konfigurationsinterfaces. Im Detail werden die Parameter in Kapitel 7.1 geschrieben.

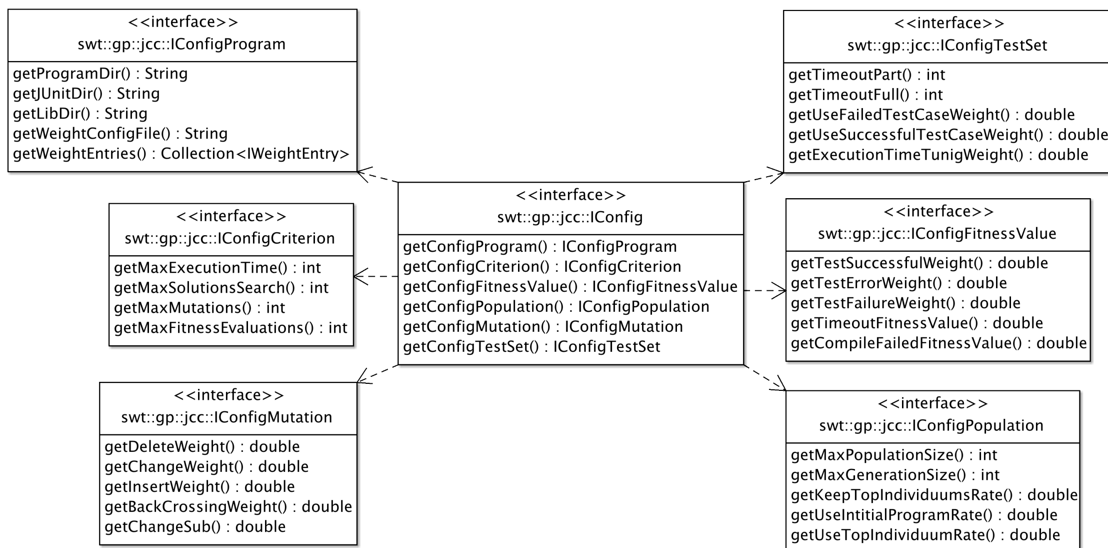


Abbildung 6.12.: Klassendiagramm für die Parameter Konfiguration

Kapitel 7

Bedienung und Steuerung

Das Verhalten der `JCC Engine` wird von unterschiedlichen Parametern bestimmt; diese werden über eine Konfigurationsschnittstelle an die `JCC Engine` übergeben. Die einzelnen Parameter werden in Kapitel 7.1 erläutert.

Die `JCC Engine` ist kein selbstständig ausführbares Programm im eigentlichen Sinne. Sie ist vielmehr dazu gedacht, um in weiterer Folge in eine IDE wie `Eclipse` eingebunden zu werden. Während der Entwicklung war es zu Testzwecken notwendig, die `JCC Engine` auszuführen, um ihr Verhalten zur Laufzeit beobachten zu können. Um das zu ermöglichen und auch um später einzelne Programme testen zu können, wurde eine `JCC GUI` (siehe Kapitel 7.2) entwickelt. Die `JCC GUI` ist ein eigenständiges Programm, das mit einem einfachen Konfigurationsfile für das zu testende Programm gesteuert wird. Die restlichen Parameter sind direkt im Programm hinterlegt, können aber über die `JCC GUI` angepasst werden.

7.1. Konfiguration

Wie im Kapitel zur Implementierung beschrieben (siehe Kapitel 6.8), wird die `JCC Engine` über eine Vielzahl von Parametern gesteuert.

7.1.1. Programm

ProgramDir

Verzeichnis mit dem Source Code des Programms.

JunitDir

Verzeichnis mit den Programm JUnit Source Code Dateien.

LibDir

Verzeichnis mit den vom Programm benötigten Bibliotheken.

WeightConfigFile

Konfigurationsfile, in dem die Fehlerwahrscheinlichkeiten für die einzelnen Code Zeilen abgebildet sind. Ein Beispiel für den Inhalt eines solchen Konfigurationsfile:

```
tcas.Tcas:58-70:1.0
tcas.Tcas:98-118:1.0
```

Auflistung 7.1: Konfigurationsfile mit Fehlerwahrscheinlichkeiten

Jede Zeile des Konfigurationsfiles setzt sich, getrennt durch einen Doppelpunkt, aus folgenden Bereichen zusammen:

- **Klasse**: Der vollständige Klassenname, auf den sich die Zeilennummern beziehen.
- **Zeilennummern**: Der Bereich von Codezeilen (von - bis), für den die Fehlerwahrscheinlichkeit definiert wird.
- **Fehlerwahrscheinlichkeit**: Die Wahrscheinlichkeit (0,0 - 1,0), dass die davor definierten Zeilennummern fehlerhaft sind.

7.1.2. Abbruchkriterien

MaxExecutionTime ($0-2^{32}$)

Maximale Ausführungszeit der `Execution Engine` in Millisekunden. Die Zeit für die Initialisierung wird nicht dazugerechnet.

MaxSolutionSearch ($0-2^{32}$)

Maximale Anzahl der Lösungen. Von einer Lösung wird gesprochen, wenn der Fitnesswert der vollständigen `TestSuiteSet` 1,0 ist, d.h. alle Testfälle positiv ausgeführt worden sind.

MaxMutations ($0-2^{32}$)

Maximale Anzahl an durchgeführten Mutationen.

MaxFintessEvaluations ($0-2^{32}$)

Maximale Anzahl an durchgeführten Fitnesswert Berechnungen.

7.1.3. Mutation

DeleteWeight, ChangeWeight, InsertWeight, BackCrossingWeight, ChangeSub

Gewichtung, mit der die `Mutationsoperatoren` ausgeführt werden. Dabei sind die Werte immer relativ zu den anderen `Mutationsoperatoren`.

Beispiel: Wird `DeleteWeight` auf 0,5 und `ChangeWeight` auf 1,5 gesetzt, bedeutet es, dass der `CHANGE` Operate mit einer 3 mal höheren Wahrscheinlichkeit als der `DELETE` Operator ausgeführt wird. Wird nun der `InsertWeight` auf 3,0 gesetzt, so wird der `INSERT` Operater doppelt so häufig als der `CHANGE` Operator und sechs mal so häufig als der `DELETE` Operator ausgeführt.

7.1.4. Test Set

TimoutPart, TimeoutFull

Timeout für das reduzierte bzw. vollständige `TestSuiteSet`.

UseFailedTestCaseWeight, UseSuccessfulTestCaseWeight

Bestimmt, mit welcher Wahrscheinlichkeit ein fehlgeschlagener bzw. erfolgreicher Testfall zum reduzierten TestSuiteSet hinzugefügt wird. Werden beide Werte auf 1,0 gesetzt, ergibt es ein vollständiges `TestSuiteSet`.

ExecutionTimeTuningWeigth

Bestimmt die Rate, mit der Testfälle hinsichtlich ihrer Laufzeit ausgewählt werden.

Beispiel: Ein Wert von 0,33 bedeutet, es werden nur die schnellsten 33 % der Testfälle im reduzierten TestSuiteSet ausgewählt.

7.1.5. Fitnesswert

TestSuccessfulWeight, TestErrorWeight, TestFailureWeight

Fitnesswert für einen erfolgreichen Testfall, einen Testfall, bei dem eine Exception geworfen wurde oder einem Testfall, bei dem eine Assertion verletzt wurde.

TimoutFitnessValue

Fitnesswert für ein Programm, bei dem der Testlauf mit einem Timeout abgebrochen wurde.

CompileFailedFitnessValue

Fitnesswert für ein Programm, das nicht kompiliert werden konnte.

7.1.6. Population

MaxPopulationSize

Maximale Anzahl an Individuen in der Population.

MaxGenerationSize

Maximale Anzahl an Generationen, die ein Programm haben kann. Ein Wert von 3 bedeutet: Das ursprüngliche Programm kann maximal 3 mal mutiert werden, dann steht dieses Programm nicht mehr für weitere Mutationen zur Verfügung.

KeepTopIndividuumRate

Prozentsatz der besten Programme in der Population, die nicht entfernt werden. Ein Wert von 0,33 bedeutet: Die besten 33 % der Programme werden nicht entfernt, wenn ein neues Programm in die Population eingefügt wird und die Population dadurch verringert werden muss.

UseInitialProgramRate

Prozentsatz, mit dem das ursprüngliche Programm mutiert wird, gegenüber einem bereits mutierten. Ein Wert von 0.33 bedeutet: Bei jedem dritten Durchgang wird das ursprüngliche Programm mutiert, ansonsten ein beliebig anderes aus der Population. Setzt man den Wert auf 0,0, wird nur für die erste Mutation das ursprüngliche Programm verwendet und danach nur noch aus diesem gebildete Programmvarianten. Ein Wert von 1,0 führt dazu, dass immer nur das ursprüngliche Programm mutiert wird; es kommt immer nur zu `first-order Mutanten` und die maximale Generation jedes Individuums in der Population beträgt 2 (0 für das ursprüngliche Programm, 1 für die Programmvariante davon und 2 für ggf. die Final Repair).

UseTopIndividuumRate

Prozentsatz, mit dem bestimmt wird, aus welchem Bereich die Programme aus der Population mutiert werden. Ein Wert von 0,33 bedeutet: Nur die besten 33 % der Population werden weiter mutiert.

7.2. JCC GUI

Anfangs war es geplant, die `JCC Engine` ohne spezielle GUI zu entwickeln, um nicht unnötig Ressourcen dafür aufzuwenden. Im Laufe der Entwicklungen hat sich aber herausgestellt, dass das Thema doch sehr komplex ist. Es wurde zwar von Anfang an ein Logging implementiert, mit dem man sich über die Vorgänge der `JCC Engine` ein sehr detailliertes Bild machen konnte, die Informationen wurden aber immer vielfältiger. Besonders problematisch war, dass man zu Beginn eines Durchlaufes oft nicht wusste, wie sich der Durchgang entwickeln würde und welche Information dann wichtig sein würden. Besonders interessant war auch, wie sich die Performance der `JCC Engine` in diversen Situationen verhält.

Um dem besser entgegen zu wirken, wurde für die `JCC Engine` ein eigene GUI entwickelt (siehe Abbildung 7.1). Die `JCC Engine` selbst sollte dadurch aber nicht geändert werden und sollte auch weiterhin ohne GUI lauffähig sein. Die GUI greift daher auch nur auf die öffentlichen Interfaces der `JCC Engine` zu.

Des Weiteren ist es über die GUI möglich, direkt auf die Parameter (siehe Kapitel 6.8) Einfluss zu nehmen und diese anzupassen. So können z.B. während des Betriebs die Testfälle, die für die `TestSuiteSets` verwendet werden, angepasst werden oder die Population vergrößert/verkleinert werden; die Auswirkungen können wieder über die GUI verfolgt werden.

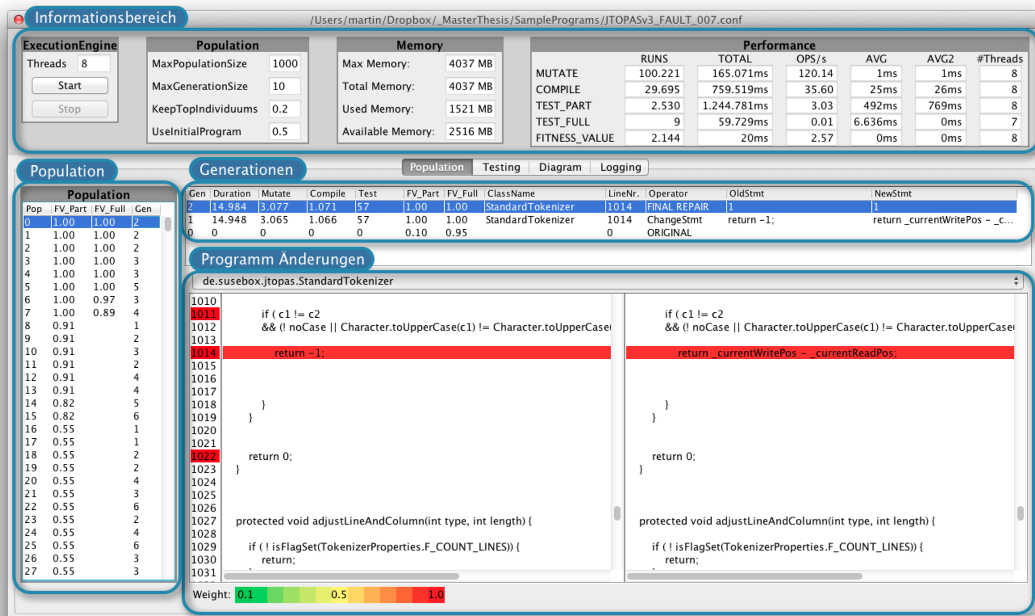


Abbildung 7.1.: Hauptfenster der GUI

7.2.1. Informationsbereich

Execution Engine

Dieser kleine Bereich dient zum Starten und Anhalten der Execution Engine. Das ist besonders interessant, um sich den Inhalt der Population anschauen zu können, der ansonsten zur Laufzeit permanenten Änderungen unterworfen wäre. Zusätzlich können hier die Anzahl der Threads, mit denen die `JCC Engine` arbeitet, abgelesen und geändert werden. Die Auswirkungen können direkt bei der Performance (siehe Kapitel 7.2.1) verfolgt werden.

Population

In diesem Bereich werden die Parameter für die `Population` angezeigt und können auch geändert werden. Im Detail sind die Parameter in Kapitel 7.1.6 beschrieben.

Memory

Die `JCC Engine` ist eine sehr speicherintensive Anwendung. Daher war es von Beginn an wichtig, den Speicherverbrauch immer im Blick zu behalten. Wenn die `JCC Engine` ins Stocken geraten ist, konnte über diese Anzeige schnell herausgefunden werden, ob es vielleicht im Zusammenhang mit dem Speicher gestanden ist. Diese Probleme hatten dann oft mit der Java Speicherverwaltung und dem `Garbage Collector` zu tun.

Performance

Hier werden die Daten der `Performance`, wie sie in Kapitel 6.7.5 beschrieben sind, dargestellt. Um auch über den Verlauf der Performance informiert zu werden, gibt es ein eigenes Diagramm, das den Verlauf der Performance darstellt (siehe Kapitel 7.2.6).

Es werden einige Anwendungsbeispiele aufgeführt, welche Informationen aus der Performance Übersicht entnommen werden können. Dabei wird bei den Beispielen auf die Daten des Screenshots der Abbildung 7.1 Bezug genommen.

Anzahl der einzelnen Schritte Aus dem Verhältnis `COMPILE` zu `MUTATE` kann abgelesen werden, wie viele Mutationen überhaupt compiliert werden, also noch nicht verworfen werden, weil sie bereits in der `Population` vorhanden sind oder waren. In unserem Beispiel sind das aktuell $\frac{29.659}{100.221} = 29,6$ %. Dieser Wert schwankt, ist zu Beginn sehr hoch und würde im Prinzip irgendwann gegen null gehen, weil irgendwann alle möglichen Mutationen durchgeführt worden sind.

Mit dem Verhältnis `TEST_PART` zu `COMPILE` kann abgelesen werden, wie viele der Mutationen kompilierbar sind. In unserem Beispiel sind das aktuell $\frac{2.530}{29.695} = 8,5$ %. Dieser Wert sollte über die Laufzeit des Programms relativ konstant bleiben und ist ein Indikator, wie gut die `Mutationsoperatoren` nicht kompilierbare Versionen erkennen können. Könnte dieser Wert nun auf 50 % erhöht werden, in dem man die `Mutationsoperatoren` verbessert, würde sich beim `COMPILE` Schritt eine Zeitersparnis von $759.519\ ms * (0,5 - 0,085) = 315.200\ ms$ ergeben. Auf die gesamte Laufzeit wäre das eine Zeitersparnis von $\frac{315.200\ ms}{2.229.120\ ms} = 14,1$ %.

TOTAL - Gesamte Laufzeit der Schritte Aus den Verhältnissen der einzelnen `TOTAL` Laufzeiten zueinander sieht man, wofür die meiste Zeit aufgewendet wird. `MUTATE` $\frac{165.071\ ms}{2.229.120\ ms} = 7,5$ %, `COMPILE` $\frac{759.519\ ms}{2.229.120\ ms} = 34,1$ % und `TEST_PART` $\frac{1.244.781\ ms}{2.229.120\ ms} = 55,8$ %. Diese Zahlen bieten eine Ausgangsbasis wenn wir die Performance der `JCC Engine` steigern wollen. So müsste zuerst die Ausführung der Testfälle beschleunigt werden, danach das Compilieren und zuletzt das Mutieren. Verbessert man nur das Mutieren um 50 %, würde sich die gesamte Performance lediglich um 3,7 % verbessern.

OPS/s - Operations per Seconds Die Anzahl der `OPS/s` wurde hauptsächlich für die Optimierung der Performance verwendet. Besonders wenn die Anzahl der Threads geändert wurde, ergaben sich bei den `OPS/s` starke Veränderungen. Damit diese Zahlen eine Relevanz haben, dürfen sie aber nur innerhalb des selben Programmes verglichen werden. Lässt sich ein Programm aufgrund seiner Beschaffenheit selbst durch Mutationen öfter kompilieren als andere Programme, werden auch öfter Tests damit durchgeführt und es kommt zu weniger Mutationen.

Ein interessanter Leistungsindikator für die `JCC Engine` ist der Wert bei `TEST_PART`. Dieser gibt an, wie viele Testdurchläufe pro Sekunde gemacht worden sind. In diesen Wert fließt indirekt auch die Güte beim Compilieren und Mutieren mit ein.

AVG und AVG2 - Durchschnittliche Laufzeit der Schritte `AVG` liefert die durchschnittliche Dauer für die Durchführung eines Schrittes in Bezug auf die gesamte Laufzeit. `AVG2` nimmt nur auf die letzten 10 Durchgänge Bezug. Dadurch können mit der `AVG2` Änderungen sehr rasch sichtbar gemacht werden, auch wenn die `JCC Engine` schon länger in Betrieb ist.

Besonders interessant ist hier die `AVG2` für den Schritt `TEST_PART`. Hier sieht man die durchschnittliche Dauer für die Durchführung der Tests mit dem reduzierten `TestSuiteSet`. Sollte dieser Wert nahe dem eingestellten Timeout sein, muss unbedingt überprüft werden, ob bei den getesteten Programmen wirklich immer eine Endlosschleife das Problem war, oder ob das Timeout zu knapp bemessen war.

7.2.2. Population

Der Bereich `Population` zeigt die Individuen an, die aktuell in der Population vorhanden sind. Die einzelnen Spalten in der Tabelle haben folgende Bedeutung:

- **Pop**: Position des Individuums innerhalb der Population.
- **FV_Part**: Fitnesswert des Individuums beim reduzierten TestSuiteSet.
- **FV_Full**: Fitnesswert des Individuums beim vollständigen TestSuiteSet (sofern eben vorhanden).
- **Gen**: Generation, in der sich das Individuum befindet; wie oft das ursprünglich fehlerhafte Programm bereits mutiert worden ist.

Jedes Individuum ist auswählbar und kann im Detail weiter betrachtet werden (siehe Kapitel 7.2.3).

7.2.3. Generationen

Wird, wie in Kapitel 7.2.2 beschrieben, ein Individuum ausgewählt, so werden in dem Bereich der `Generationen` die einzelnen Programmvarianten des Individuums dargestellt. Die einzelnen Spalten in der Tabelle haben für die Programmvariante folgende Bedeutung:

- **Gen**: Die Generation, in der sich die Programmvariante befindet.
- **Duration**: Die Zeit, die nach der Initialisierung der `ExecutionEngine` vergangen ist.
- **Mutate, Compile, Test**: Die Anzahl der Mutationen, Compilierungen und Tests, die bisher durchgeführt worden sind.
- **FV_Part**: Fitnesswert für das reduzierte TestSuiteSet.
- **FV_Full**: Fitnesswert für das vollständige TestSuiteSet (sofern vorhanden).
- **ClassName**: Klasse, in der die Änderung durchgeführt worden ist.
- **LineNr.**: Codezeile, in der die erste Änderungen vorgenommen worden ist.

- **Operator**: Operator, der für die Änderung verwendet worden ist. Dies kann einer der `Mutationsoperatoren` sein oder die `Final Repair`. Das erste Programm jedes Individuums ist immer das ursprüngliche Programm und wird `ORIGINAL` bezeichnet.
- **OldStmt, NewStmt**: Damit werden die Änderungen, die vom Operator durchgeführt worden sind, angegeben. Bei der `Final Repair` ist das die Anzahl der zuvor und danach geänderten Statements.

Wählt man eine Programmvariante aus, so werden die vollständigen Änderungen und weitere Details im unteren Bereich dargestellt (siehe Kapitel 7.2.4).

7.2.4. Programm Änderungen

Hier wird das ursprüngliche Programm (links) und die ausgewählte Programmvariante (rechts) dargestellt. Geänderte Bereiche werden mit Rot markiert. Sollten mehrere Klassen in dieser Programmvariante geändert worden sein, so können diese mittels Auswahlliste gewählt werden.

Zusätzlich werden auch die Zeilennummern des ursprünglichen Programms angezeigt. Farblich markiert wird die Fehlerwahrscheinlichkeit jeder einzelnen Codezeile. Die Legende dazu befindet sich im unteren Bereich.

7.2.5. TestSuiteSets Einstellungen

Den Inhalt der `TestSuite`, sowie die Einstellungen für das reduzierte und vollständige `TestSuiteSet`, können im Bereich `Testing` (siehe Abbildung 7.2) eingesehen und geändert werden.

- **Name**: Das des `Java JUnit Tests`.
- **Result**: Resultat des Tests beim ursprünglichen Programm.
- **ExecTime**: Ausführungszeit des Tests beim ursprünglichen Programm.
- **ExecutePart**: Test wird beim reduzierten `TestSuiteSet` ausgeführt.
- **ExecuteFull**: Test wird beim vollständigen `TestSuiteSet` ausgeführt.

Werden zusätzliche Tests ausgewählt oder bestehende entfernt, wirkt sich das zum einen auf die Performance aus, zum anderen schlägt es sich in der Berechnung der Fitnesswerte nieder.

7.2.6. Performance Entwicklung

Wie im Kapitel zur Performance beschrieben (siehe Kapitel 7.2.1), wird hier ein Teil der Performancedaten über die gesamte Laufzeit visualisiert. Die Zeit wird auf der x-Achse, die Anzahl der Durchgänge auf der y-Achse dargestellt. Wegen dem großen Wertebereich, der auf der y-Achse darzustellen ist, wird die y-Achse logarithmisch dargestellt. Bei einem normalen Programmlauf wird die Darstellung wie in Abbildung 7.3 sein - die einzelnen Linien verlaufen annähernd parallel zueinander.

Würde die Linie für `MUTATE` steigen und die restlichen konstant bleiben, würde es darauf hindeuten, dass keine neuen Mutationen mehr gebildet werden können.

7.2.7. Logging Einstellungen

Über die GUI können sehr viele Dinge visualisiert werden. Gerade wenn es aber darum geht, zu visualisieren was, wann und in welcher Reihenfolge passiert ist, scheitert die GUI meist. Die Erschwernis beim Logging ist meist, dass im Vorfeld oft nicht bekannt ist, welche Bereiche des Programms in welcher Tiefe für das Logging notwendig sein werden. Jederzeit alles mitzuloggen ist aufgrund der Performance nicht ohne Weiteres möglich.

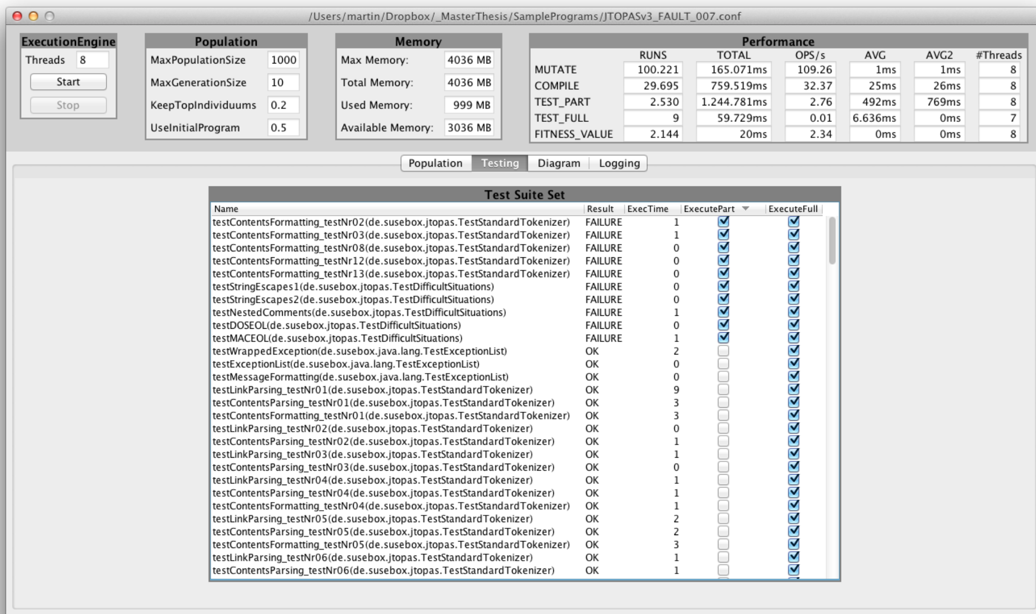


Abbildung 7.2.: GUI für die TestSuiteSets

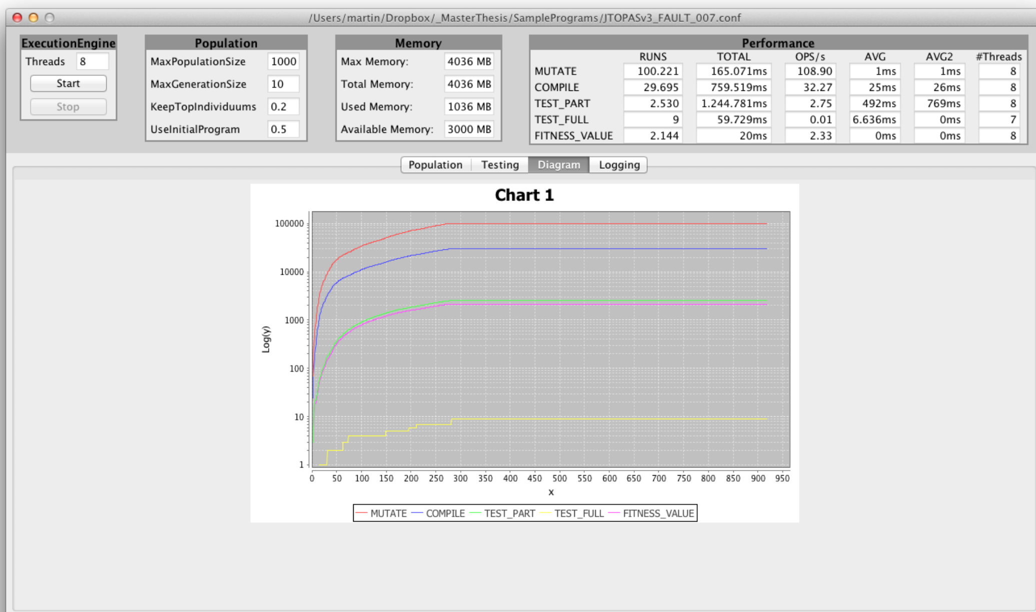


Abbildung 7.3.: GUI für die Performance Entwicklung

Deshalb wurde der Bereich für die Logging Steuerung ebenfalls in der GUI bereitgestellt (siehe Abbildung 7.4). Somit kann nun bei Bedarf das Logging für den gewünschten Programmbereich in der jeweiligen Tiefe gesteuert werden.

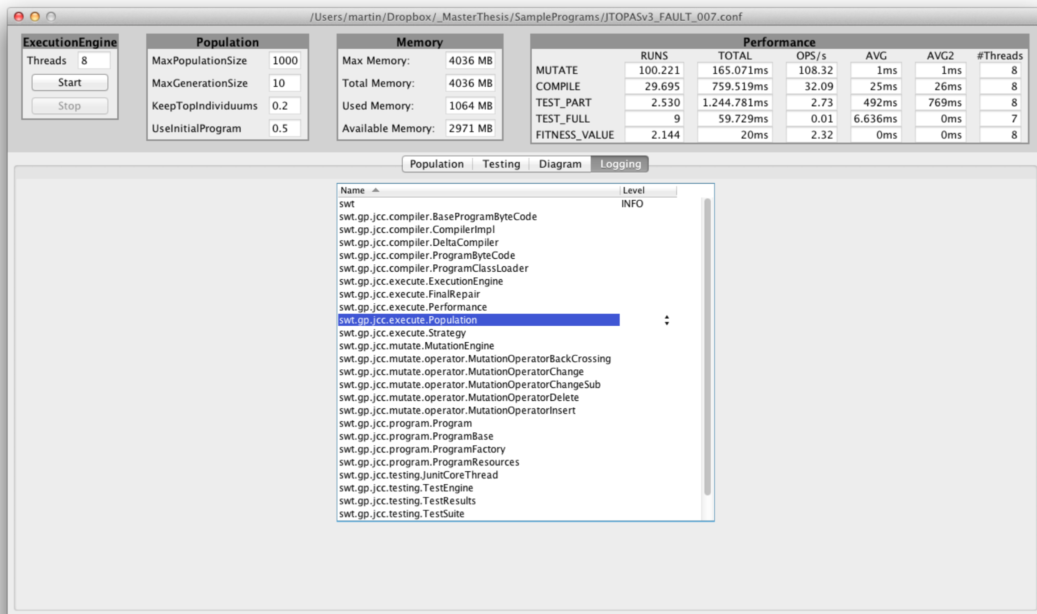


Abbildung 7.4.: GUI für das Logging

Kapitel 8

Evaluierungsaufbau

Einfache Tests und Programmläufe wurden schon mit der `JCC Engine` und der `JCC GUI` durchgeführt, um die korrekte Funktionalität und Stabilität zu prüfen. Während diesen Programmläufen sind Fragestellungen aufgetaucht, deren Antwort nur eine genauere Evaluierung liefern kann. Darunter z.B. Fragen zur Performance Optimierung, optimale Wahl der Parameter und auch der allgemeinen Leistungsfähigkeit, sowie den Grenzen der `JCC Engine`.

Im Weiteren werden die für die Evaluierung notwendigen Werkzeuge und Schritte beschrieben. Anschließend werden in Kapitel 9 die Ergebnisse präsentiert und diskutiert.

8.1. Verwendete Hardware

Für die Evaluierung wird viel Rechenleistung benötigt. Es gibt viele Programme mit unterschiedlichen Fehlern, viele Parameter, die unterschiedliche Ausprägungen haben können und es müssen aufgrund der vielen Zufälligkeiten viele Iterationen durchgeführt werden, um die Streuung der Ergebnisse in den Griff zu bekommen. Diese Faktoren multiplizieren sich und entsprechend viele Programmläufe und damit Rechenleistung wird benötigt.

8.1.1. Lokaler Rechner: Cyria

Für die meisten Programmläufe wird der lokale Rechner verwendet, auf dem auch die Entwicklung der `JCC Engine` durchgeführt wurde. Die Ausstattung und damit die Rechenleistung ist aus heutiger Sicht (2011) gut bis sehr gut. Die Daten des Rechner im Detail:

- **CPU**: Intel Core i7-870 4x 2.93GHz mit Hyperthreading (8 virtuelle Cores)
- **RAM**: 8 GB
- **HDD**: 120 GB Solid State Disk
- **Betriebsystem**: Mac OS X Lion 10.7.2

8.1.2. Virtuelle Cloud Server: JiffyBox 01 - 50

Nach den ersten Testläufen hat sich herausgestellt, dass **Cyria** alleine nicht ausreichen wird, um die Evaluierungen in der angestrebten Zeit zu erledigen. Da die Rechenleistung aber nur temporär benötigt wird,

wäre ein Kauf von zusätzlicher Hardware nicht ökonomisch sinnvoll gewesen. Daher wird zusätzliche Rechenleistung bei Bedarf angemietet.

JiffyBox hat sich nach einigen Recherchen als sehr gutes Angebot herauskristallisiert. Wie später bei den Evaluierungsergebnissen in Abschnitt 9.6 beschrieben, kann damit die Rechenleistung in Summe um den Faktor 15 erhöht werden. Die für die Evaluierung verwendete Konfiguration der Cloud Server im Detail:

- **CPU**: 2x
- **RAM**: 1 GB
- **HDD**: 50 GB HDD
- **Betriebsystem**: Linux Debian Leny 5.0 64bit

Um die Handhabung mit den Servern zu erleichtern, stellt der Anbieter eine API * bereit, die in `JSON` gehalten ist. Für diese API wurde ein Shell Script geschrieben, mit dem die Anzahl der Cloud Server Instanzen einfach skaliert werden kann (siehe Auflistung 8.1). Das Script wird über folgende Argumente gesteuert:

- **CREATE** XX YY: Erzeugt neue `JiffyBoxen` mit den Nummern von XX bis YY.
- **START** XX YY: Startet die `JiffyBoxen` mit den Nummern von XX bis YY.
- **STOP** XX YY: Beendet die `JiffyBoxen` mit den Nummern von XX bis YY.
- **DELETE** XX YY: Entfernt die `JiffyBoxen` mit den Nummern von XX bis YY.

8.1.3. Java VM Parameter

Für Java Programme, wie die `JCC Engine`, hat die `Java Virtual Machine` den Status einer Hardware, deren Verhalten mit Parametern gesteuert werden kann.

Für die Evaluierung sind besonders die Parameter für die Speicherverwaltung notwendig. Welche Parameter für die einzelnen Umgebungen verwendet werden, sind in Tabelle 8.1 abgebildet.

Parameter	Cyria	JiffyBox	Beschreibung
-Xmx	4g	512mb	Bestimmt den maximal verfügbaren Heap Speicher
-Xms	4g	512mb	Bestimmt den zu Beginn verfügbaren Heap Speicher
-XX:MaxPermSize	256M	128mb	Bestimmt den Speicher für Klassen und Methoden
-XX:ReservedCodeCacheSize	512m	128mb	Bestimmt den maximalen Speicher für Code
-XX:+CMSClassUnloadingEnabled	X	X	Ermöglicht das Entladen von Klassen
-XX:+AggressiveHeap	X	X	Versucht den Heap Speicher möglichst gut auszunützen

Tabelle 8.1.: Java Virtual Machine Parameter

8.2. Evaluierungsdatenbank

Um die Evaluierung durchzuführen, ist die Speicherung vieler Daten notwendig. Zum einen müssen die Parameter, mit denen die Evaluierungen durchgeführt werden, verwaltet und gespeichert werden, zum anderen müssen die Ergebnisse von jedem Evaluierungslauf gespeichert werden. Um nicht in Gefahr zu laufen, dass die Daten nach einem Evaluierungslauf zu sehr verdichtet werden und vielleicht später, wenn man z.B. die Fragestellung vertiefen oder erweitern will, nicht mehr die notwendigen Daten vorhanden sind, werden die Ergebnisse möglichst ohne Verdichtung gespeichert.

*https://www.jiffybox.de/doc/jiffybox-api-dokumentation.pdf

```
#!/bin/sh
PLAN_ID=10
MASTER_BOX_ID=16022
TOKEN=[INSERT PASSWORD HERE]

RAW_DATA=`curl https://api.jiffybox.de/${TOKEN}/v1.0/jiffyBoxes`
JIFFY_BOX_LIST=`echo ${RAW_DATA} | egrep -o \"id\"\:[[:digit:]]{5} | egrep -o [[:digit:]]{5}$ | grep -v ${MASTER_BOX_ID}`

if [ $# -eq 0 ]
then
  echo "No Command"
  exit 0
fi

NUMBER_OF_BOXES=100
COUNT=0
if [ $# -eq 2 ]
then
  NUMBER_OF_BOXES=$2
fi

if [ $# -eq 3 ] || [ $1 == "CREATE" ]
then
  for (( i = $2; i <= $3; i++ ))
  do
    index=`printf "%02d" $i`
    COMMAND="curl -d name=JIFFYBOX_P${PLAN_ID}_${index} -d planid=${PLAN_ID} https://api.jiffybox.de/${TOKEN}/v1.0/jiffyBoxes/${MASTER_BOX_ID}"
    $COMMAND
  done
  exit 0
fi

if [ $1 == "START" ]
then
  COMMAND="curl -X PUT -d status=START https://api.jiffybox.de/${TOKEN}/v1.0/jiffyBoxes/"
fi

if [ $1 == "STOP" ]
then
  COMMAND="curl -X PUT -d status=SHUTDOWN https://api.jiffybox.de/${TOKEN}/v1.0/jiffyBoxes/"
fi

if [ $1 == "DELETE" ]
then
  COMMAND="curl -X DELETE https://api.jiffybox.de/${TOKEN}/v1.0/jiffyBoxes/"
fi

for line in $JIFFY_BOX_LIST;
do
  let "COUNT += 1"
    if [ $COUNT -gt $NUMBER_OF_BOXES ]
    then
      break
    fi
    COMMAND_EXEC=${COMMAND}${line}
    $COMMAND_EXEC
done
```

Auflistung 8.1: Script zur einfachen Steuerung der JiffyBox API

Zusammen ergibt das eine große Datenmenge, für deren Speicherung sich eine Datenbank anbietet. Da in weiterer Folge auch die Cloud Server auf diese Datenbank zugreifen müssen, um die Parameter für den Evaluierungslauf abzuholen und die Ergebnisse zurückzuschreiben, wurde eine MySql (Version 5.1.36) bei einem Internet Provider gemietet.

8.2.1. Tabellen für die Parameter

Für alle Konfigurationsinterfaces (siehe Kapitel 6.8) gibt es je eine Tabelle. Darin werden sämtliche Permutationen der Parameter abgebildet. Jeder einzelne Eintrag in einer der Tabellen entspricht einer möglichen

Parametrierung des jeweiligen Konfigurationsinterfaces der `JCC Engine` und ist über eine eindeutige ID referenzierbar. Je nach Evaluierungslauf kann jeder Eintrag aktiviert oder deaktiviert werden. Als Beispiel sieht man in Abbildung 8.1 einen Teil der Tabelle für die Parametrierung der `Mutationsoperatoren`. Der Aufbau der anderen Parameter-Tabellen ist ähnlich.

mutID	mutDelete	mutChange	mutInsert	mutBackCrossing	mutChangeSub	mutEnabled
1	1	1	1	1	1	1
2	0	1	1	1	1	1
3	0.25	1	1	1	1	1
4	0.5	1	1	1	1	1
5	0.75	1	1	1	1	0
7	1.25	1	1	1	1	0
8	1.5	1	1	1	1	1
9	1.75	1	1	1	1	0
10	2	1	1	1	1	1
11	1	0	1	1	1	1
12	1	0.25	1	1	1	0
13	1	0.5	1	1	1	1
14	1	0.75	1	1	1	0
15	1	1.25	1	1	1	0
16	1	1.5	1	1	1	1
17	1	1.75	1	1	1	0
18	1	2	1	1	1	1
19	1	1	0	1	1	1
20	1	1	0.25	1	1	1
21	1	1	0.5	1	1	1
22	1	1	0.75	1	1	0

Abbildung 8.1.: Evaluierungstabelle für die Parametrierung der Mutationsoperatoren

8.2.2. Tabellen für die Programme und Programmfehler

Für die Programme, die getestet werden, gibt es ebenso eine Tabelle (siehe Abbildung 8.2). Sie enthält die Information, welche Programme es gibt und wo die benötigten Ressourcen (Java Source Code, JUnit Source Code und Bibliotheken) zu finden sind. Die Programme können in dieser Tabelle aktiviert oder deaktiviert werden, sollte ein Evaluierungslauf mit nur einem Programm oder einem Teil der Programme durchgeführt werden.

In einer weiteren Tabelle (siehe Abbildung 8.3) sind zu den jeweiligen Programmen die Fehler eingetragen, die ebenso aktiviert und deaktiviert werden können.

prgID	prgName	prgWorkingDir	prgSourceCode	prgJUnitCode	prgLibDir	prgEnabled
1	TCAS	TCAS	40_ORG	00_JUNIT	10_LIBS	1
2	ATMS	ATMS	40_ORG	00_JUNIT	10_LIBS	1
3	REFLECTION-VISIT	REFLECTION-VISITOR	40_ORG	00_JUNIT	10_LIBS	1
4	JTOPASv1	JTOPASv1	40_ORG	00_JUNIT	10_LIBS	1
5	JTOPASv2	JTOPASv2	40_ORG	00_JUNIT	10_LIBS	1
6	JTOPASv3	JTOPASv3	40_ORG	00_JUNIT	10_LIBS	1

Abbildung 8.2.: Evaluierungstabelle für die Programme

8.2.3. Tabellen für die Evaluierungsläufe

Für die eigentlichen Evaluierungsläufe gibt es eine Tabelle (siehe Abbildung 8.4), in der der Name des Evaluierungslaufes, die Anzahl der Iterationen, sowie ob aktiv oder inaktiv definiert ist. Aus dieser Information,

fltID	fltPrgID	fltNumber	fltSourceCode	fltEnabled
1	6	1	50_FAULT_001	0
2	6	2	50_FAULT_002	0
3	6	3	50_FAULT_003	0
4	6	4	50_FAULT_004	0
5	6	5	50_FAULT_005	0
6	6	6	50_FAULT_006	0
7	6	7	50_FAULT_007	1
8	6	8	50_FAULT_008	0
9	6	9	50_FAULT_009	0
10	6	10	50_FAULT_010	1
11	6	11	50_FAULT_011	0
12	6	12	50_FAULT_012	0
13	6	13	50_FAULT_013	1
14	6	14	50_FAULT_014	0
15	6	15	50_FAULT_015	0
16	6	16	50_FAULT_016	0

Abbildung 8.3.: Evaluierungstabelle für die Fehler der Programme

zusammen mit den aktivierten Einträgen aus den Parameter-Tabellen und den bereits verfügbaren Ergebnissen, ergibt sich automatisch die Statustabelle für die Evaluierungsläufe (siehe Abbildung 8.5). Darin ist jeder einzelne Lauf mit der Anzahl der noch zu absolvierenden Iterationen verzeichnet. Das Programm, das die Evaluierungen durchführt, muss hier Eintrag für Eintrag abarbeiten.

runID	runVersion	runIterations	runEnabled
1	InitialAllPrgWithFaults	4	0
2	MutationOperations	25	0
3	MutationOperations2	25	1

Abbildung 8.4.: Evaluierungstabelle für die Evaluierungsläufe

rstRunID	rstFltID	rstCriID	rstFvID	rstMutID	rstPopID	rstTsID	rstIteration
3	7	3	1	1	2	1	25
3	7	3	1	2	2	1	25
3	7	3	1	3	2	1	25
3	7	3	1	4	2	1	25
3	7	3	1	8	2	1	25
3	7	3	1	10	2	1	25
3	7	3	1	11	2	1	38
3	7	3	1	13	2	1	25
3	7	3	1	16	2	1	25
3	7	3	1	18	2	1	25
3	7	3	1	19	2	1	25
3	7	3	1	20	2	1	25
3	7	3	1	21	2	1	25
3	7	3	1	24	2	1	25
3	7	3	1	26	2	1	25
3	7	3	1	27	2	1	25
3	7	3	1	29	2	1	25
3	7	3	1	32	2	1	25
3	7	3	1	34	2	1	25

Abbildung 8.5.: Evaluierungstabelle für den Status der Evaluierungsläufe

8.2.4. Tabellen der Ergebnisse

Schließlich werden die Ergebnisse jedes Evaluierungslaufs ebenfalls in einer eigenen Tabelle gespeichert. Je Evaluierungslauf erfolgt ein weiterer Eintrag. Dieser Eintrag enthält die folgenden Informationen:

- Den Evaluierungslauf, durch den er gestartet wurde.
- Die Parameter, mit denen er gestartet wurde.
- Die Dauer des Evaluierungslaufes.
- Die Anzahl der untersuchten Statements.
- Die Anzahl der gefundenen Lösungen.
- Alle Daten, wie sie auch in der `IPerformance` (siehe Kapitel 6.7.5) vorhanden waren.
- Die Anzahl und Art der Testfälle im reduzierten `TestSuiteSet`.
- Fehlermeldungen, sofern welche aufgetreten sind.
- Der Name des Evaluierungssystems, das die Evaluierung durchgeführt hat, sowie die Version der `JCC Engine`.

Zusätzlich gibt es noch für jeden dieser Einträge in einer weiteren Tabelle die besten 10 Individuen aus der Population mit allen Daten, wie sie auch in der GUI (siehe 7.2.2) angezeigt werden.

8.3. Tools zur Evaluierung

Das eigentliche Evaluierungsprogramm ist vom Aufbau und der Funktion sehr einfach. Es liest aus einem Konfigurationsfile die Anzahl der Threads aus, mit denen das Programm gestartet werden soll, sowie den Basispfad, unter dem die Programme zum Testen gespeichert sind. Der Systemname wird direkt aus den Umgebungsvariablen des Betriebssystems gelesen (somit können später die einzelnen Instanzen der Jiffy-Box unterschieden werden). Die Version der `JCC Engine` wird zur Kompilezeit gebildet, hinterlegt und wird ebenfalls ausgelesen.

Es wird nun ein Eintrag aus der Statustabelle für die Evaluierungsläufe gelesen und der Evaluierungslauf mit den Parametern gestartet. Die Ergebnisse werden wieder in die Datenbank eingetragen (siehe Kapitel 8.2.4). Danach terminiert das Evaluierungsprogramm.

In einer ersten Implementierung wurde das Evaluierungsprogramm so implementiert, dass die Evaluierungsläufe in einer Schleife durchführt werden. Das führte aber zu dem Problem, dass die Evaluierungsläufe in Hinblick auf die Performance teilweise immer besser wurden (weil die Java VM anscheinend Optimierungen vornehmen konnte). Nach einer gewissen Zeit wurden die Ergebnisse allerdings immer schlechter (vermutlich weil Teile im Speicher nicht zuverlässig vom Garbage Collector wiederverwertet wurden). Zusätzlich gab es Stabilitätsprobleme, wenn die Evaluierung über längere Zeit ausgeführt wurde. Daher fiel die Entscheidung auf die oben beschriebene Implementierung, die gewährleistet, dass alle Evaluierungsläufe dieselbe Ausgangssituation haben.

Um die wiederkehrende Ausführung der Evaluierungsläufe sicherzustellen, wurden 2 Scripte implementiert. Das erste Script (siehe Auflistung 8.2) prüft in einer Schleife, ob das Evaluierungsprogramm läuft und startet es gegebenenfalls. Ein weiteres Script (siehe Auflistung 8.3) ist als Cronjob auf den Cloud Servern registriert und überprüft periodisch, ob das erste Script aktiv ist. Somit wird gewährleistet, dass automatisch beim Serverstart oder bei Abstürzen die Evaluierungsläufe immer vorgeführt werden.

```
#!/bin/sh
JAVA_PARAMETER="-Xmx4g -Xms4g -XX:+AggressiveHeap -XX:ReservedCodeCacheSize=512m -XX:MaxPermSize=512m"
OUT_FILE=out.txt

while [ 1 ]
do
  if ps -ef | grep -v grep | grep JCC_EVALUATE.jar > /dev/null
  then
    echo "Java is running, nothing to do"
  else
    echo "Java not running, starting it"
    java ${JAVA_PARAMETER} -jar JCC_EVALUATE.jar >> ${OUT_FILE} &
  fi
  if [ $# -eq 0 ]
  then
    break
  fi
  sleep 4
done
```

Auflistung 8.2: Script zum Ausführen der Evaluierung.

```
#!/bin/sh

if ps -ef | grep -v grep | grep runEvaluation.sh > /dev/null
then
  echo "runEvaluation.sh is running, nothing to do"
else
  echo "runEvaluation.sh not running, starting it"
  ./runEvaluation.sh XXX > /dev/null &
fi
```

Auflistung 8.3: Cronjob zum Starten des Evaluierungsscripts.

8.4. Programme zur Evaluierung

Einer der wichtigsten Punkte für die Evaluierung ist das Vorhandensein von Programmen mit Fehlern, um sie hinsichtlich automatischer Fehlerkorrektur evaluieren zu können. Dabei ist wichtig, dass eine ausreichende Menge an Testfällen für das Programm vorhanden ist. Die Programme von `SIR` [12] erfüllen diese Anforderungen. Von dieser Quelle wurden die Programme **TCAS** (TCAS ist ursprünglich in `C` geschrieben und wurde auf Java portiert) und **JTOPAS** (3 Versionen, mit jeweils unterschiedlichen Fehlerszenarien) verwendet. Des Weiteren wurden die Programme **ATMS** und **REFLECTION-VISITOR** vom **Institut für Software Technologie an der TU Graz** verwendet. In Summe werden die Evaluierungen mit 6 unterschiedlichen Programmen durchgeführt (siehe Tabelle 8.2). Eine Übersicht über die einzelnen Fehler der jeweiligen Programme findet man bei der Evaluierung (siehe Kapitel 9.1).

ProgrammBez.	Programm	# Klassen	# Zeilen	# Testfälle	# Fehler
JTOPAS_v1	JTopas 0.4	19	4.284	126	10
JTOPAS_v2	JTopas 0.5.1	21	3.597	128	12
JTOPAS_v3	JTopas 0.6	50	9.335	209	17
TCAS	Tcas	1	120	1.545	32
ATMS	Atms	34	2.987	16	3
REFLECTION-VISITOR	ReflectionVisitor	12	646	14	5
Summe		137	20.969	2.038	79

Tabelle 8.2.: Übersicht der zu evaluierenden Programme

8.5. Definition für korrigiertes Programm

Ziel der `JCC Engine` ist es, ein fehlerhaftes Programm zu korrigieren. Bei den Programmen, die für die Evaluierung verwendet werden, wird immer das fehlerfreie Programm und der Programmfehler bereitgestellt. Wird nun das fehlerhafte Programm von der `JCC Engine` repariert (alle Testfälle werden positiv ausgeführt), könnte das Ergebnis mit dem fehlerfreien Programm verglichen werden und bei exakter Übereinstimmung kann davon ausgegangen werden, dass eine korrekte Lösung gefunden wurde.

In der Praxis wird es Abweichungen von der gefundenen Lösung zum originalen Programm geben, was einen solchen Vergleich nicht praktikabel erscheinen lässt. Das liegt an den unendlichen Möglichkeiten, eine Spezifikation in Programmcode umzusetzen. Dieses Recht, aus diesen unendlichen Möglichkeiten zu schöpfen, soll auch der `JCC Engine` eingeräumt werden.

Um nun eine Lösung der `JCC Engine` zu überprüfen, müsste jede Lösung individuell betrachtet und hinsichtlich Korrektheit geprüft werden; das würde aber das Vorhandensein einer vollständigen Spezifikation für jedes Programm voraussetzen und würde viel Zeit in Anspruch nehmen. In Summe werden bei den Evaluierungen mehrere 100.000 Evaluierungsläufe ausgeführt, sollte auch nur bei 10 % eine Lösung gefunden werden, so müssten mehrere 10.000 Lösungen überprüft werden.

Da das nicht praktikabel ist, wird für die folgenden Evaluierungen definiert, dass die `JCC Engine` erfolgreich ist, wenn sie zumindest eine Variante des defekten Programms findet, sodass damit alle Testfälle positiv ausgeführt werden können. Ist das der Fall, so sprechen wir von einer `Lösung` für den Programmfehler. Ob die Lösung wirklich eine gültige Korrektur ist, kann letztlich nur individuell geprüft werden.

Besonders während der Implementierung, und im Speziellen beim Testen mit der GUI, wurden viele Lösungen im Detail betrachtet. Ein sehr großer Teil der Lösungen schien sehr plausibel, um eine Korrektur für den Programmfehler zu sein.

8.6. Bestimmung der Fehlerwahrscheinlichkeiten

Wie in Kapitel 5.6.3 und 7.1.1 beschrieben, benötigt die `JCC Engine` zusätzlich zum Programm und den JUnit Tests auch noch Informationen über die Fehlerwahrscheinlichkeit der einzelnen Statements bzw. Codezeilen im Programm. Es gibt mehrere Methoden, um das zu erreichen (siehe Kapitel 3.2). Die Ergebnisse, je nach verwendeter Methode für die Fehlerwahrscheinlichkeitsbestimmung, können unterschiedlich ausfallen und beeinflussen indirekt die Qualität bzw. Ergebnisse der `JCC Engine`. Um das zu vermeiden, wird folgendes Verfahren für das Zuweisen der Fehlerwahrscheinlichkeiten zu den Statements verwendet:

Das fehlerhafte Statement, bzw. die fehlerhaften Codezeilen werden mit einem Wert von 1,0 bewertet. Zusätzlich werden alle anderen Statements, die sich in der selben Methode befinden, ebenfalls mit 1,0 bewertet. Handelt es sich bei dem fehlerhaften Statement um eine Klassenvariable, so wird diese und alle anderen Klassenvariablen mit 1,0 bewertet. Die Überlegungen, die zur Anwendung dieser Methode geführt haben, sind:

- Wie schon oben angeführt, ist diese Methode von keinem anderen Algorithmus abhängig, und die Ergebnisse sollten daher besser untereinander vergleichbar werden.
- Es ist einfach zu implementieren bzw. anzuwenden, die Komplexität der Evaluierung wird dadurch nicht erhöht.
- Mit dem Ergebnis von diesem Verfahren sollte einfach auf ein anderes Verfahren interpoliert werden können (siehe Beispiel).

Beispiel für das Interpolieren auf ein anderes Verfahren: Bei dem gewählten Verfahren (Variante 1) gibt es eine fehlerhafte Methode mit 10 Statements; jedes würde gleich mit 1,0 bewertet werden. Der Fehler wird mit diesem Verfahren im Durchschnitt nach 100.000 Mutationen gefunden. Ein anderes Verfahren (Variante 2) liefert für das fehlerhafte Statement eine Fehlerwahrscheinlichkeit von 0,5, alle anderen

Statements in der Methode werden mit 0,4 bewertet. Zusätzlich werden noch 10 Statements außerhalb der Methode mit 0,1 bewertet.

Die Wahrscheinlichkeit, dass bei Variante 1 nun das fehlerhafte Statement mutiert wird, ist $\frac{1{,}0}{1{,}0*10} = 10{,}0$ %; die Wahrscheinlichkeit bei Variante 2 ist $\frac{0{,}5}{0{,}5+0{,}4*9+0{,}1*10} = 9{,}8$ %. Daher könnte angenommen werden, dass die Lösung bei Variante 2 im Durchschnitt nach $10.000*\frac{0{,}10}{0{,}098} = 10.204$ Mutationen gefunden wird. Natürlich ist das nur eine Hypothese, die genau evaluiert werden müsste. So führt nicht zwingend immer nur das Mutieren des fehlerhaften Statements zu einer korrekten Lösung etc.

Kapitel 9

Evaluierungsergebnisse

In diesem Kapitel werden die Ergebnisse der Evaluierung diskutiert. Zu Beginn wird in Abschnitt 9.1 betrachtet, welche Programme überhaupt mit der `JCC Engine` korrigiert werden konnten und wie schnell und zuverlässig es durchgeführt wurde. In den weiteren Abschnitten 9.2 bis 9.5 werden die Parametereinstellungen der `JCC Engine` evaluiert. Schließlich wird in Abschnitt 9.6 die Performance evaluiert.

9.1. Korrigierbarkeit der Programmfehler

9.1.1. Fragestellung

Für die Evaluierung der Parameter (Population, Mutation, TestSets, etc.) ist eine hohe Anzahl an Evaluierungsläufen notwendig, da mit einer hohen Streuung der Daten gerechnet wird. Daher ist es vor diesen Evaluierungen sinnvoll, Programmfehler aus der Evaluierung auszuschließen.

Bei den verbliebenen Programmfehlern, also jene die korrigiert werden können, ist zu evaluieren, wie zuverlässig und schnell die Fehlerkorrektur möglich ist.

Programmfehler, die von der JCC Engine nicht verarbeitet werden können

Es gibt einige Anforderungen an die Programme und deren Fehler, um sie automatisch korrigieren zu können (siehe Kapitel 5.5). Zusammengefasst dürfen folgende Dinge nicht passieren:

- **NOT COMPILE**: Das Programm ist nicht kompilierbar.
- **ENDLESS LOOP**: Das Programm besitzt Endlosschleifen und terminiert nicht.
- **NO ERROR**: Es gibt keinen Testfall, der fehl schlägt.

Programme, die eines der oben genannten Kriterien erfüllen, können nicht automatisch korrigiert werden. Sie werden daher von weiteren Evaluierungen ausgeschlossen.
Frage 1: *Welche Programmfehler erfüllen eines dieser Kriterien?*

Programmfehler, die nicht korrigiert werden können

Neben Programmen mit Fehlern, die nicht ausgeführt werden können, gibt es noch solche, die zwar ausgeführt werden können, bei denen der Fehler aber nicht korrigiert werden kann (UNCORRECTABLE).

Das kann z.B. daran liegen, dass von den `Mutationsoperatoren` der Fehler nicht abgedeckt wird, dass der fehlerhafte Bereich nicht mutiert wird oder dass der Fehler in den bisherigen Versuchen nicht korrigiert worden ist und mehr Mutationen bzw. Iterationen notwendig gewesen wären.
Frage 2: *Welche Programmfehler können nicht oder nur theoretisch korrigiert werden?*

Qualität der Fehlerkorrektur

Neben der Information, ob der Fehler erkannt und korrigiert werden konnte oder nicht, können mit der selben Evaluierung auch Daten über die Qualität erhoben werden, mit der ein Fehler korrigiert wurde.
Frage 3: *Wie zuverlässig können die Programmfehler korrigiert werden und wie lange dauert die Fehlerkorrektur?*

9.1.2. Versuchsaufbau

Hardware

- Cyria, 32 Threads

Parameter

- **Abbruchkriterien**: MaxExecutionTime: 10m, MaxFitnessEvaluations: 1.500, MaxSolutionSearch: 1
- **Timeouts**: 5s (reduziertes TestSuiteSet), 30s (vollständiges TestSuiteSet)

Evaluierungsläufe

- **Aktivierte Programme**: Alle (6)
- **Aktivierte Fehler**: Alle (79)
- **Parameterpermutationen**: 1
- **Iterationen**: 25
- **Evaluierungsläufe insgesamt**: $79 * 1 * 25 = 1.975$

Messpunkte

Als Messpunkt wird zum einen überprüft, ob ein Durchgang einen Fehler (NOT COMPILE, ENDLESS LOOP, NO ERROR) verursacht hat und daher nicht ausgeführt werden konnte, zum anderen wird überprüft ob ein Fehler zumindest einmalig korrigiert werden konnte. Wenn das nicht der Fall war, wird der Fehler überprüft und begutachtet, ob er überhaupt mit der aktuellen Implementierung der `JCC Engine` korrigiert werden kann oder ob der Fehler nur zufällig in den 25 Iteration bisher nicht korrigiert worden ist. Die maximale Zeit für einen Evaluierungslauf ist mit 10 Minuten sehr lang gewählt, wird allerdings durch maximal 1.500 Testevaluierungen begrenzt. Zusätzlich terminiert der Evaluierungslauf sofort, wenn eine Lösung gefunden wird.

Um die Frage nach der Qualität, mit der die Fehler korrigiert werden, beantworten zu können, wird zusätzlich noch aufgezeichnet, wie viele Tests und Mutationen notwendig waren und wie lange es gedauert hat, um die erste Korrektur des Fehlers zu finden.

9.1.3. Ergebnis

Erkennbare und korrigierbare Fehler

In den folgenden Tabellen (siehe Tabellen 9.1 - 9.6) werden für jedes Programm die einzelnen Fehler aufgelistet und vermerkt, ob der Fehler für das weitere Evaluieren verwendet wird oder nicht.
Antwort auf **Frage 1**: In Summe konnten von den 79 Fehlern 18 Fehler (17x `NO ERROR` und 1x `NOT COMPILE`) nicht durch die Testfälle erkannt werden.
Antwort auf **Frage 2**: Bei weiteren 31 Programmfehlern ist nach dieser Evaluierung nicht davon auszugehen, dass sie automatisch korrigiert werden können.
Die verbleibenden 30 Fehler konnten automatisch korrigiert werden.

JTOPAS_v1 Bei diesem Programm werden einige Fehler, durch das nicht Vorhandensein eines fehlgeschlagenen Testfalles, nicht erkannt. Des Weiteren bestehen zwei Fehler (3, 9) darin, dass eine Methode vollständig entfernt wurde; diese Fehler können in der aktuellen Implementierung nicht korrigiert werden. Von den ursprünglichen 10 Fehlern werden für die weitere Evaluierung nur noch 5 Fehler (1, 2, 5, 6, 10) verwendet (siehe Tabelle 9.1).

FehlerNr	Datei	ZeilenNr	Fehler	Bemerkung
1	ExtIOException	46	Statement entfernt	OK
2	ExtIOException	51, 57	true/false vertauscht	OK
3	ExtIOException	66	Methode entfernt	UNCORRECTABLE
4	AbstractTokenizer	634	+1 entfernt	NO ERROR
5	AbstractTokenizer	772, 782	+1,-1 entfernt u. 0/1 vertauscht	OK
6	AbstractTokenizer	920	Methode vertauscht	OK
7	Token	9-18	Zahlen vertauscht	NO ERROR
8	Tokenizer	37-40	Zahlen vertauscht	NO ERROR
9	ExtIndex...Exception	57	Methode entfernt	UNCORRECTABLE
10	ExtIndex...Exception	46	Statement entfernt	OK

Tabelle 9.1.: JTOPAS_v1 Fehlerübersicht

JTOPAS_v2 6 Fehler konnten nicht erkannt werden. Fehler 3 konnte nicht korrigiert werden. Dieser Fehler sollte allerdings mit zusätzlichen Mutationsoperatoren korrigiert werden können. Von den ursprünglichen 12 Fehlern werden für die weiteren Evaluierungen nur noch 5 Fehler (1, 7, 10, 11, 12) verwendet (siehe Tabelle 9.2).

FehlerNr	Datei	ZeilenNr	Fehler	Bemerkung
1	ExtIOException	65	Statement entfernt	OK
2	ExtIOException	10, 78	Import Definition vertauscht	NO ERROR
3	Throwable...Formatter	14	boolescher Wert negiert	UNCORRECTABLE
4	Throwable...Formatter	33-36	Code entfernt	NO ERROR
5	AbstractTokenizer	612	Funktionsaufruf vertauscht	NO ERROR
6	AbstractTokenizer	601	</>= vertauscht	NO ERROR
7	AbstractTokenizer	665	</<= vertauscht	OK
8	AbstractTokenizer	674	>=/> vertauscht	NO ERROR
9	Token	130	true/false vertauscht	NO ERROR
10	Token	131	boolescher Wert negiert	OK
11	Token	149	&&/|||| vertauscht	OK
12	Token	152	&&/|||| vertauscht	OK

Tabelle 9.2.: JTOPAS_v2 Fehlerübersicht

JTOPAS_v3 9 Fehler konnten nicht erkannt werden. Alle Fehler, die erkannt wurden, konnten auch korrigiert werden. Von den ursprünglichen 17 Fehlern werden für die weiteren Evaluierungen nur noch 8 Fehler (4, 6, 7, 8, 9, 10, 13, 16) verwendet (siehe Tabelle 9.3).

FehlerNr	Datei	ZeilenNr	Fehler	Bemerkung
1	EnvironmentProvider	30	Methodenaufruf geändert	NO ERROR
2	EnvironmentProvider	39	Methodenaufruf geändert	NOT COMPILE
3	EnvironmentProvider	22 - 48	synchronized entfernt	NO ERROR
4	PluginTokenizer	40	Methodenaufruf entfernt	OK
5	StandardTokenizer	1061	+ =/− = vertauscht	NO ERROR
6	StandardTokenizer	1050	+ =/= vertauscht	OK
7	StandardTokenizer	1016	1/−1 vertauscht	OK
8	StandardTokenizer	849	Methodenaufruf geändert	OK
9	StandardTokenizer	842	Methodenaufruf geändert	OK
10	StandardTokenizer	756	+1 entfernt	OK
11	StandardTokenizer	689	+ =/− = vertauscht	NO ERROR
12	StandardTokenizer	649	boolescher Wert negiert	NO ERROR
13	StandardTokenizerProperties	45	Methodenaufruf entfernt	OK
14	StandardTokenizerProperties	158 - 197	synchronized entfernt	NO ERROR
15	StandardTokenizerProperties	69 - 84	synchronized entfernt	NO ERROR
16	StandardTokenizerProperties	1529	static bei Variable hinzugefügt	OK
17	StandardWhitespaceHandler	54	</<= vertauscht	NO ERROR

Tabelle 9.3.: JTOPAS_v3 Fehlerübersicht

TCAS Hier konnte nur Fehler 27 nicht erkannt werden. Er hängt von einem anderen Fehler ab und tritt daher in dieser Evaluierung nicht auf. 20 Fehler können nicht automatisch korrigiert werden. Bei den Fehlern handelt es sich zum einen um Zahlen, die verändert worden sind, zum anderen um Ausdrücke, bei denen Teile entfernt oder geändert worden sind. In beiden Fällen müssten die Mutationsoperatoren besser die einzelnen Teile der Statements mutieren. Mit der aktuellen Implementierung der Mutationsoperatoren ist nicht davon auszugehen, dass die Fehler korrigiert werden können. Von den ursprünglich 32 Fehlern werden für die weiteren Evaluierungen nur noch 11 Fehler (1, 4, 9, 20, 21, 22, 23, 24, 25, 31, 32) verwendet (siehe Tabelle 9.4).

FehlerNr	Datei	ZeilenNr	Fehler	Bemerkung
1	TCAS	64	>/>= vertauscht	OK
2	TCAS	54	Konstante vertauscht	UNCORRECTABLE
3	TCAS	103	&&/|| vertauscht	UNCORRECTABLE
4	TCAS	68	&&/|| vertauscht	OK
5	TCAS	101	Ausdruck entfernt	UNCORRECTABLE
6	TCAS	90	>/>= vertauscht	UNCORRECTABLE
7	TCAS	34	Zahl geändert	UNCORRECTABLE
8	TCAS	24	Zahl geändert	UNCORRECTABLE
9	TCAS	77	>/>= vertauscht	OK
10	TCAS	90, 94	</<= vertauscht	UNCORRECTABLE
11	TCAS	90, 94, 101	</<= u. &&/|| vertauscht	UNCORRECTABLE
12	TCAS	101	&&/|| vertauscht	UNCORRECTABLE
13	TCAS	8	Zahl geändert	UNCORRECTABLE
14	TCAS	12	Zahl geändert	UNCORRECTABLE
15	TCAS	17, 101	Zahl geändert u. Ausdruck entfernt	UNCORRECTABLE
16	TCAS	34	Zahl geändert	UNCORRECTABLE
17	TCAS	34	Zahl geändert	UNCORRECTABLE
18	TCAS	34	Zahl geändert	UNCORRECTABLE
19	TCAS	34	Zahl geändert	UNCORRECTABLE
20	TCAS	61	>/>= vertauscht	OK
21	TCAS	61	Ausdruck geändert	OK
22	TCAS	61	Ausdruck geändert	OK
23	TCAS	77	Ausdruck geändert	OK
24	TCAS	77	Ausdruck geändert	OK
25	TCAS	84	>/>= vertauscht	OK
26	TCAS	101	Ausdruck entfernt	UNCORRECTABLE
27	TCAS	100	Abhängig von anderem Fehler	NO ERROR
28	TCAS	54	Ausdruck geändert	UNCORRECTABLE
29	TCAS	54	Ausdruck geändert	UNCORRECTABLE
30	TCAS	54	Ausdruck geändert	UNCORRECTABLE
31	TCAS	65, 70, 109	Statements hinzugefügt u. Ausdruck verändert	OK
32	TCAS	81, 86, 110	Statements hinzugefügt u. Ausdruck entfernt	OK

Tabelle 9.4.: TCAS Fehlerübersicht

ATMS Hier konnten zwar alle 3 Fehler erkannt werden, es konnte aber keiner automatisch korrigiert werden. Bei 2 Fehlern handelt es sich um vertauschte Variablennamen, die einzeln nicht mutiert werden. Ein, auf diese Fehlerklasse ausgerichteter, Mutationsoperator sollte diesen Fehler korrigieren können. Beim dritten Fehler wurde ein Statement negiert und konnte nicht korrigiert werden, weil die korrekte Version des Statements an keiner anderen Stelle im Programm vorkommt. Ein geeigneter Mutationsoperator sollte den Fehler korrigieren können. Von diesem Programm werden für die weiteren Evaluierungen keine Fehler mehr verwendet (siehe Tabelle 9.5).

FehlerNr	Datei	ZeilenNr	Fehler	Bemerkung
1	ATMSLabel	93	Objekt vertauscht	UNCORRECTABLE
2	ATMSLabel	94	Objekt vertauscht	UNCORRECTABLE
3	ATMSLabel	107	boolescher Wert negiert	UNCORRECTABLE

Tabelle 9.5.: ATMS Fehlerübersicht

REFLECTION-VISITOR Alle 5 Fehler wurden erkannt, es konnte allerdings nur Fehler 4 automatisch korrigiert werden. Die Fehler, die nicht korrigiert werden konnten, bestanden aus vertauschten Vergleichsoperatoren und vertauschten booleschen Werten. Diese Fehler sind auch bei den vorherigen Programmen aufgetreten und sollten ebenfalls mit verbesserten Mutationsoperatoren korrigiert werden können. Von den 5 Fehlern wird nur der vierte Fehler für weitere Evaluierungen verwendet (siehe Tabelle 9.6).

FehlerNr	Datei	ZeilenNr	Fehler	Bemerkung
1	ReflectionHelper	23	boolescher Wert negiert	UNCORRECTABLE
2	ReflectionHelper	33	>/< vertauscht	UNCORRECTABLE
3	ReflectionHelper	47	boolescher Wert negiert	UNCORRECTABLE
4	ReflectionHelper	71	==/! = vertauscht	OK
5	ReflectionHelper	57	>/< vertauscht	UNCORRECTABLE

Tabelle 9.6.: REFLECTION-VISITOR Fehlerübersicht

Qualität der Fehlerkorrektur

Neben der Frage, ob ein Fehler korrigiert werden kann oder nicht, ist in weiterer Folge wichtig zu wissen, wie zuverlässig das Korrigieren erfolgt und wie lange es dauert. Dabei werden zum einen die Durchschnittswerte der Mutationen, Tests und Ausführungszeiten gegenübergestellt, zum anderen wird im Detail betrachtet wie lange es jeweils gedauert hat, den einzelnen Programmfehler zu finden. Bei diesen Ergebnissen werden nur noch Programmfehler betrachtet, bei denen zumindest einmalig eine Lösung gefunden worden ist.

Durchschnittswerte Die Durchschnittswerte mit ihren zugehörigen Standardabweichungen der Evaluierungsläufe sind in Tabelle 9.7 aufgelistet.
Antwort auf **Frage 3:** Von den evaluierten Programmfehlern wurde für einen großen Teil (21/31) bei jedem Evaluierungslauf eine Lösung gefunden. Weitere 6 Programmfehler konnten in mindestens 80 % der Evaluierungsläufe korrigiert werden. Die restlichen 3 Fehler konnten bei ca. jedem zweiten Evaluierungslauf korrigiert werden. In Summe wurde bei 750 Evaluierungsläufen 699 mal eine Lösung gefunden, das entspricht 93,2 %.

Bei den Durchschnittswerten für die Anzahl der Mutationen und Tests, sowie der Ausführungsdauer tritt eine große Standardabweichung im Bereich von 50 % auf. Die Daten streuen in einem großen Bereich, sowohl bei den Evaluierungsläufen zu einem Programmfehler, als auch in Summe über alle Programmfehler. Das liegt daran, dass sehr viele Entscheidungen zufällig getroffen werden.

Die durchschnittliche Ausführungszeit für das Finden einer Lösung betrug rund 62 Sekunden. Betrachtet man die Daten im Detail, sieht man, dass viele Programmfehler aber im Bereich unter 20 Sekunden gelöst worden sind.

Die Anzahl der Tests, die notwendig waren um eine Lösung zu finden, sind ebenfalls eine wichtige Kenngröße. Durchschnittlich benötigt man 333 Tests um die Lösung zu finden. Betrachtet man die Daten im Detail, sieht man allerdings, dass 50 % der Programmfehler bereits mit weniger als 100 Tests korrigiert werden konnten. In der nächsten Tabelle 9.8 und in der Abbildung 9.1 wird das noch deutlicher hervorgehoben. Mit dieser Kenngröße kann die Performance der `JCC Engine` evaluiert werden (z.B. bei unterschiedlichen Parametrierungen). Die Evaluierungsumgebung hat keinen Einfluss auf diesen Wert, im Gegensatz zur Ausführungsdauer, die von der Performance der Evaluierungsumgebung beeinflusst wird. Dadurch können auch Werte miteinander verglichen werden, die auf unterschiedlichen Evaluierungsumgebungen entstanden sind.

Ergebniss der 25 Evaluierungsläufe je Fehler

Programmfehler		Lösungen	Lösung%	ø Mutationen	σ Mutations	ø Tests	σ Tests	ø Dauer[ms]	σ Dauer[ms]
JTOPASv1	1	25	100 %	21.359	9.984	629	256	36.755	9.006
JTOPASv1	2	25	100 %	7.250	3.411	241	130	24.522	7.895
JTOPASv1	5	22	88 %	14.018	6.617	689	334	104.559	40.944
JTOPASv1	6	11	44 %	8.520	4.587	707	421	177.764	77.113
JTOPASv1	10	24	96 %	22.601	9.199	680	229	39.477	10.381
JTOPASv2	1	24	96 %	24.328	14.343	700	350	39.492	13.466
JTOPASv2	7	25	100 %	1.170	173	49	10	40.612	8.873
JTOPASv2	10	25	100 %	2.432	413	87	12	37.120	8.296
JTOPASv2	11	25	100 %	1.374	396	59	11	34.888	6.130
JTOPASv2	12	25	100 %	735	246	42	6	33.887	6.900
JTOPASv3	4	25	100 %	20.726	7.097	435	121	43.937	10.171
JTOPASv3	6	25	100 %	4.003	1.598	242	126	57.485	19.086
JTOPASv3	7	24	96 %	11.791	4.597	456	196	96.710	35.484
JTOPASv3	8	12	48 %	15.905	6.116	1.001	433	154.267	64.471
JTOPASv3	9	13	52 %	13.009	5.492	806	379	185.932	76.706
JTOPASv3	10	24	96 %	2.963	1.661	121	80	251.172	145.292
JTOPASv3	13	25	100 %	50.297	20.553	569	240	171.737	70.479
JTOPASv3	16	20	80 %	26.291	18.675	503	348	134.868	76.967
TCAS	1	25	100 %	5.991	4.566	349	238	29.049	21.894
TCAS	4	25	100 %	1.657	577	117	38	13.384	4.010
TCAS	9	25	100 %	3.006	1.524	192	96	15.944	8.989
TCAS	20	25	100 %	1.787	1.043	129	63	14.804	5.383
TCAS	21	25	100 %	2.319	832	133	48	15.805	5.670
TCAS	22	25	100 %	2.375	1.382	151	78	15.105	6.692
TCAS	23	25	100 %	3.215	1.231	184	74	16.800	8.396
TCAS	24	25	100 %	2.463	1.303	159	78	17.003	7.585
TCAS	25	25	100 %	1.787	559	114	32	14.353	7.228
TCAS	31	25	100 %	2.101	1.022	225	119	19.948	8.921
TCAS	32	25	100 %	1.425	765	153	82	18.215	7.969
REFLECTIONV.	4	25	100 %	2.770	2.506	55	42	17.872	8.577
ø		**23,3**	**93,20 %**	**9.322**	**4.416**	**333**	**156**	**62.449**	**26.299**

Tabelle 9.7.: Durchschnittswerte und Standardabweichungen für korrigierbare Programmfehler. Die Werte werden nur aus den erfolgreichen Evaluierungsläufen berechnet.

Verteilung der Dauer bis zur Fehlerkorrektur Durch die hohe Streuung bei den Durchschnittswerten (siehe Tabelle 9.7) werden die Daten für die Ausführungszeit genauer betrachtet. In der Tabelle 9.8 sind die Programmfehler und die Anzahl der Evaluierungsläufe, die für das Finden einer Lösung notwendig waren, aufgelistet. In der Abbildung 9.1 werden die Daten verdichtet über alle Programmfehler dargestellt.

Die Tabelle 9.8 gibt einen Überblick, bei welchen Programmfehlern mit welchen Zeiten für das Finden einer Lösung gerechnet werden muss. In Summe sieht man aber, dass 434 von 699 Lösungen bereits mit weniger als 250 Tests korrigiert werden, das sind über 62 %. Innerhalb der ersten 500 Tests können bereits 3 von 4 aller Lösungen gefunden werden.

Lösungsverteilung je Fehler

Programmfehler		250	500	750	1.000	1.250	1.500	Summe
JTOPASv1	1	0	13	4	6	0	2	25
JTOPASv1	2	15	10	0	0	0	0	25
JTOPASv1	5	2	5	7	3	3	2	22
JTOPASv1	6	3	1	2	2	2	1	11
JTOPASv1	10	0	6	9	8	0	1	24
JTOPASv2	1	1	9	3	7	1	3	24
JTOPASv2	7	25	0	0	0	0	0	25
JTOPASv2	10	25	0	0	0	0	0	25
JTOPASv2	11	25	0	0	0	0	0	25
JTOPASv2	12	25	0	0	0	0	0	25
JTOPASv3	4	1	14	10	0	0	0	25
JTOPASv3	6	20	3	2	0	0	0	25
JTOPASv3	7	4	15	4	0	1	0	24
JTOPASv3	8	1	1	1	1	4	4	12
JTOPASv3	9	0	3	3	3	2	2	13
JTOPASv3	10	22	2	0	0	0	0	24
JTOPASv3	13	1	11	6	6	1	0	25
JTOPASv3	16	7	4	4	3	1	1	20
TCAS	1	11	8	4	1	1	0	25
TCAS	4	25	0	0	0	0	0	25
TCAS	9	20	5	0	0	0	0	25
TCAS	20	24	1	0	0	0	0	25
TCAS	21	25	0	0	0	0	0	25
TCAS	22	21	4	0	0	0	0	25
TCAS	23	19	6	0	0	0	0	25
TCAS	24	22	3	0	0	0	0	25
TCAS	25	25	0	0	0	0	0	25
TCAS	31	19	5	1	0	0	0	25
TCAS	32	21	4	0	0	0	0	25
REFLECTIONV.	4	25	0	0	0	0	0	25
Summe		434	133	60	40	16	16	699

Tabelle 9.8.: Anzahl der getesteten Programmvarianten je Programmfehler bis die erste Lösung gefunden wurde. Bei der Tabelle wird in jedem Bereich (250 - 1.500) angegeben, bei wie vielen Evaluierungsläufen eine Lösung innerhalb des Bereichs gefunden wurde. Werten über 0 wurde eine grüne Farbe zugewiesen; je dunkler das Grün, desto mehr Evaluierungsläufe sind in diesen Bereich gefallen. *Zum Beispiel für JTOPASv3 Programmfehler Nr. 4: 1 mal wurden weniger als 251 Tests benötigt, um eine Lösung zu finden, 14 mal waren es zwischen 251 und 500 und 10 mal waren es zwischen 501 und 750. In Summe wurde für diesen Fehler bei jedem Evaluierungslauf eine Lösung gefunden 25/25.*

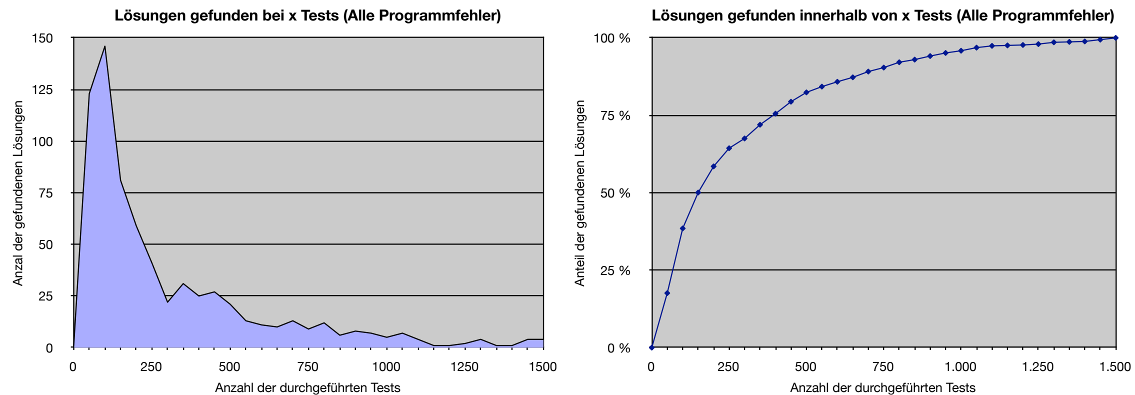


Abbildung 9.1.: Im linken Diagramm wird die Anzahl der Lösungen je durchgeführter Testanzahl dargestellt. *Zum Beispiel: 150 Lösungen konnten mit mehr als 50 aber weniger als 100 Tests gefunden werden.* Das rechte Diagramm beschreibt in Summe, wie viel Prozent der Lösungen mit weniger als x Tests gefunden wurde. *Zum Beispiel: Mit 500 Tests wurden über 75 % der Lösungen gefunden.*

9.2. Parametrierung der Mutationsoperatoren

9.2.1. Fragestellung

Die einzelnen `Mutationsoperatoren` sind die Kernelemente bei der Mutation. In der aktuellen Implementierung sind 5 `Mutationsoperatoren` verfügbar. Bei den bisherigen Evaluierungen hatte jeder `Mutationsoperator` dieselbe Wahrscheinlichkeit, um ausgewählt zu werden. Das Verhältnis der einzelnen `Mutationsoperatoren` zueinander kann über die Parametrierung verändert werden.
Frage 1: *Wie ist das ideale Verhältnis der* `Mutationsoperatoren` *zueinander, um möglichst schnell möglichst viele Lösungen zu finden?*

9.2.2. Erwartetes Ergebnis

Das Ergebnis ist schwer abzuschätzen. Die einzelnen `Mutationsoperatoren` können sich teilweise gegenseitig ersetzten. Somit könnte ein `DELETE` durch ein `CHANGE` eines beliebigen Statements, dass an dieser Stelle keine Auswirkung hat, ersetzt werden. `CHANGE` kann durch ein `INSERT` mit nachfolgendem `DELETE` ersetzt werden. Einzig der `INSERT` Operator kann nicht durch andere Operatoren ersetzt werden.

Neben dem Zusammenhang der `Mutationsoperatoren` untereinander, gibt es noch eine Abhängigkeit zu anderen Parametern. Der `BackCrossing` Operator hat nur Auswirkungen, wenn mindestens 3 Generationen von einem Programm erlaubt sind (0 ist das ursprüngliche Programm, 1 ist die erste Änderung, die rückgängig gemacht werden muss, 2 ist die Änderung, die zur Lösung führt, 3 ist schließlich der `BackCrossing` Operator, der die Änderung von 1 rückgängig macht). Es müssen daher in der Population mindestens 3 Generationen für die Individuen zugelassen sein.

9.2.3. Versuchsaufbau

Hardware

- Cyria, 32 Threads

- 30x JiffyBox 2 Core, 8 Threads

Parameter

Für die Mutationsparameter wäre es natürlich wünschenswert, wenn jeder einzelne `Mutationsoperator` mit allen Permutationen gegenseitig ausgeführt werden würde. Würde man von 0,0 bis 2,0 mit einer Schrittlänge von 0,1 gehen, gäbe das 20 Schritte je Operator. Um bei 5 `Mutationsoperatoren` alle Kombinationen testen zu können, müssten $20^5 = 3.200.00$ unterschiedliche Konfigurationen ausgeführt werden. Bei 30 Fehlern mit je 50 Durchläufen müssten $3.200.000 * 30 * 50 = 4.800.000.000$ Evaluierungsläufe durchgeführt werden. Das ist praktisch nicht durchführbar. Daher wird zum einen die Schrittlänge verkleinert, zum anderen wird darauf verzichtet, jede mögliche Permutation zu evaluieren. Je `Mutationsoperator` werden 4 unterschiedliche Parameter verwendet (0,0/0,5/1,5/2,0), die übrigen `Mutationsoperatoren` verbleiben auf dem Wert 1,0. Sollten sich während der Evaluierung zusätzlich interessante Bereiche für die Evaluierung ergeben, werden weitere Parameter evaluiert (0,25/4/8).

- **Abbruchkriterien**: MaxExecutionTime: 10m, MaxFitnessEvaluations: 1.500, MaxSolutionSearch: 1
- **Timeouts**: 5s (reduziertes TestSuiteSet), 30s (vollständiges TestSuiteSet)
- **MutationsOperatoren**: jeweils 0,0/0,5/1,0/1,5/2 zusätzlich noch vereinzelt (0,25/4/8)

Evaluierungsläufe

- **Aktivierte Programme**: 5
- **Aktivierte Fehler**: 30
- **Parameterpermutationen**: 30
- **Iterationen**: 50
- **Evaluierungsläufe insgesamt**: $30 * 30 * 50 = 45.000$

Messpunkte

Es gibt keine eindeutigen Werte, mit denen die Qualität eines Evaluierungslaufes bestimmt werden kann. Zum einen ist wichtig, dass möglichst viele Lösungen gefunden werden können, zum anderen, dass es möglichst schnell geht. Diese beiden Anforderungen schließen sich oft gegenseitig aus.

Daher wird aufgezeichnet, wie viele Lösungen mit den jeweiligen Parametern gefunden werden konnten (Lösungen gefunden), aber auch wie lange es gedauert hat (ø Tests). Bei der Dauer wird, wie schon in der vorherigen Evaluierung, wieder der Durchschnittswert für die Anzahl der Tests gewählt, als auch wie viele Lösungen nach x Tests bereits gefunden werden.

9.2.4. Ergebnis

Die Ergebnisse sind in der Tabelle 9.9 ersichtlich. Der rot umrahmte Bereich ist das Ergebnis für die Basiswerte, wenn alle `Mutationsoperatoren` mit dem gleichen Verhältnis bewertet werden. Der Prozentwert für die gefundenen Lösungen gibt an, bei wie vielen Durchgängen mit den jeweiligen Parametern eine Lösung gefunden wurde.

Aus den Ergebnissen kann Folgendes abgeleitet werden:

- Ähnlich den vorherigen Tests ist die Streuung der Werte sehr groß, wie aus der Standardabweichung zu erkennen ist.

- Wird auf den `INSERT` Operator verzichtet, ist die Wahrscheinlichkeit, dass eine Lösung gefunden wird deutlich geringer als bei den übrigen Parametrierungen. Zugleich werden die Lösungen aber auch am schnellsten gefunden. Da die Lösungsrate möglichst hoch gehalten werden soll (auch wenn dadurch die Lösungssuche mehr Zeit benötigt), sind gute Parameter für diesen `Mutationsoperator` bei 1,0 angesiedelt.
- Der `DELETE` Operator ist bei dieser Evaluierung am unwichtigsten, bei den führenden Parametereinstellungen ist er meist mit 0,0 oder 0,25 parametriert.
- Die Ergebnisse beim `CHANGE` Operator streuen sehr stark. Hier lieferten Werte im Bereich von 1 - 8 gute Ergebnisse.
- `BACKCROSSING` ist auch mehr im Bereich von 1,0 anzusiedeln.
- Bei dem `CHANGESUB` Operator ist keine Tendenz zu erkennen. Er kann auf 1 belassen werden.

In Summe ist in den Ergebnissen zwar eine Tendenz zu erkennen, eine allgemein gültige, optimale Parametrierung kann daraus jedoch nicht abgeleitet werden. Die Evaluierungsergebnisse könnten natürlich je Programmfehler getrennt betrachtet werden, dort sollte sich die Streuung verringern und es könnten die optimalen Parameter leichter gefunden werden. Da in der Praxis der Programmfehler nicht im Vorfeld bekannt sind, würde diese Parametrierung je Programmfehler keine Anwendung finden.

Die Antwort auf die **Frage 1** lautet daher: Soll die maximale Geschwindigkeit für die Fehlerkorrektur erzielt werden, kann der `INSERT` Operator auf 0,0 gesetzt werden und die anderen Operatoren auf 1,0 belassen werden. Dieser Geschwindigkeitsgewinn wird allerdings mit einer schlechteren Lösungsrate erkauft. Liegt das Hauptaugenmerk jedoch auf einer hohen Lösungsrate, sollte der `DELETE` Operater auf 0,0 oder 0,25 gesetzt, ggf. der `CHANGE` Operator leicht erhöht und die restlichen Operatoren auf 1,0 belassen werden.

Delete	Change	Insert	Back Crossing	Change Sub	Lösungen gefunden	ø Tests	σ Tests	T50	T100	T150	T200	T250	T300	T1.500
0	1	1	1	1	86 %	131	185	40 %	23 %	12 %	7 %	6 %	2 %	11 %
0	1	1	1	1	86 %	143	208	39 %	23 %	12 %	7 %	3 %	3 %	12 %
1	1	1	1	1	86 %	179	256	36 %	23 %	11 %	6 %	4 %	2 %	18 %
1	2	1	1	1	86 %	180	267	36 %	22 %	13 %	6 %	4 %	2 %	17 %
1	4	1	1	1	85 %	157	255	46 %	23 %	6 %	5 %	2 %	3 %	15 %
1	1	1	1	1	85 %	159	218	35 %	25 %	11 %	7 %	4 %	4 %	14 %
1	1	2	1	1	85 %	162	226	37 %	23 %	12 %	7 %	4 %	3 %	15 %
1	2	1	1	1	85 %	176	274	41 %	25 %	7 %	5 %	2 %	2 %	17 %
1	8	1	1	1	85 %	182	301	48 %	21 %	6 %	2 %	3 %	3 %	16 %
0	4	1	4	0	84 %	136	234	56 %	18 %	5 %	2 %	2 %	3 %	13 %
1	1	2	1	1	84 %	159	220	37 %	21 %	13 %	8 %	4 %	3 %	15 %
1	1	1	1	1	84 %	161	209	35 %	22 %	12 %	9 %	4 %	2 %	16 %
1	1	1	2	1	84 %	163	231	38 %	21 %	12 %	7 %	4 %	2 %	17 %
1	1	1	1	2	84 %	178	262	37 %	21 %	12 %	8 %	3 %	3 %	17 %
2	1	1	1	1	84 %	179	255	37 %	21 %	13 %	7 %	3 %	1 %	18 %
1	1	1	2	1	84 %	179	258	36 %	22 %	11 %	7 %	5 %	3 %	17 %
0	4	1	2	0	83 %	122	191	56 %	17 %	6 %	3 %	2 %	2 %	13 %
0	4	1	2	0	83 %	156	261	53 %	17 %	5 %	4 %	2 %	2 %	17 %
1	1	1	8	1	83 %	164	228	37 %	21 %	12 %	7 %	4 %	3 %	16 %
1	1	1	1	2	83 %	176	249	36 %	21 %	13 %	7 %	4 %	3 %	17 %
1	1	1	0	1	83 %	179	262	37 %	22 %	9 %	7 %	4 %	3 %	17 %
1	1	1	1	1	83 %	184	278	36 %	23 %	10 %	7 %	6 %	3 %	16 %
1	1	1	1	1	82 %	174	257	36 %	25 %	12 %	6 %	4 %	2 %	16 %
1	1	1	4	1	82 %	177	255	35 %	22 %	12 %	7 %	5 %	3 %	16 %
2	1	1	1	1	82 %	180	271	37 %	22 %	13 %	5 %	4 %	4 %	16 %
1	1	1	1	0	81 %	173	268	41 %	21 %	11 %	6 %	3 %	3 %	15 %
1	1	1	1	1	80 %	166	241	37 %	23 %	11 %	8 %	3 %	2 %	15 %
1	0	1	1	1	80 %	175	235	33 %	21 %	14 %	8 %	7 %	3 %	16 %
1	1	0	1	1	80 %	185	297	40 %	22 %	9 %	5 %	5 %	2 %	17 %
1	1	0	1	1	71 %	102	148	48 %	21 %	13 %	7 %	4 %	2 %	5 %

Tabelle 9.9.: Lösungsrate und durchschnittlich benötigte Tests zum Finden einer Lösung in Abhängigkeit der unterschiedlichen `Mutationsoperatoren`. Die Werte sind nach dem Anteil der gefundenen Lösungen sortiert. Der rot umrahmte Bereich ist das Ergebnis, wenn alle `Mutationsoperatoren` gleichermaßen verwendet werden. *Zum Beispiel 1. Zeile: Wird der `DELETE` Operator nicht verwendet, wurde in 86 % der Evaluierungsläufe eine Lösung gefunden. Wenn eine Lösung gefunden wurde, geschah dies durchschnittlich nach 131 Tests. Bereits 40 % aller Lösungen wurden innerhalb von 50 Tests gefunden und nur 11 % der Lösungen benötigten über 300 Tests.*

9.3. Parametrierung der Population

9.3.1. Fragestellung

Die `Population` verwaltet die `Individuen`. Durch ihre Parameter wird entschieden, wie viele `Individuen` gespeichert werden, wie oft und welche `Individuen` mutiert werden und nach welchen Kriterien sie wieder aus der `Population` ausgeschieden werden.

Frage 1 : *Wie wirken sich die Größe der Population und die maximale Anzahl an Generationen auf das Lösungsverhalten aus?*

Frage 2: *Ist es notwendig, Individuen mit einem höheren `Fitnesswert` als andere bevorzugt zu behandeln (öfter mutieren, bevorzugt in der Population belassen)?*

Frage 3: *Wie hoch soll die Rate sein, mit der das ursprünglich fehlerhafte Programm, anstelle eines bereits mutierten Programms aus der Population, mutiert wird?*

9.3.2. Versuchsaufbau

Hardware

- Cyria, 32 Threads
- 40x JiffyBox 2 Core, 8 Threads

Parameter

- **Abbruchkriterien**: MaxExecutionTime: 10m, MaxFitnessEvaluations: 1.500, MaxSolutionSearch: 1
- **Timeouts**: 5s (reduziertes TestSuiteSet), 30s (vollständiges TestSuiteSet)
- **MaxPopulationSize**: 10, 20, 40, 80, 160, 320, 640, 1280, 2560
- **MaxGenerationSize**: 2, 4, 8, 10
- **KeepTopIndividuumRate**: 0,0/0,2/0,4/0,6/0,8
- **UseTopIndividuumRate**: 0,0/0,2/0,4/0,6/0,8
- **UseInitialProgramRate**: 0,0/0,2/0,4/0,6/0,8

Evaluierungsläufe

- **Aktivierte Programme**: 5
- **Aktivierte Fehler**: 30
- **Parameterpermutationen**: 30
- **Iterationen**: 50
- **Evaluierungsläufe insgesamt**: $30 * 30 * 50 = 45.000$

Messpunkte

Wie in den vorherigen Beispielen werden die Lösungsquote und die durchschnittlich zur Lösung benötigten Tests als Messpunkte verwendet.

9.3.3. Ergebnis

Wie auch bei der vorherigen Evaluierung ist die Streuung der Daten sehr groß. Aus den Daten (siehe Tabelle 9.10) können aber trotzdem einige Erkenntnisse gewonnen werden.

Die `MaxPopulationSize` lieferte bei kleiner Größe (10, 20) die besten Werte. Das liegt daran, dass die Population sehr schnell aufgefüllt wird. Wenn die Population voll ist (alle `Individuen` haben ihre maximale Anzahl an Generationen erreicht), wird für weitere Mutationen immer das ursprünglich fehlerhafte Programm returniert. Das Lösungsverhalten entspricht dann dem Selben, als wenn die `UseInitialProgramRate` auf 1,0 gesetzt ist (siehe blaue Umrahmung in der Tabelle 9.10).

Bei der `MaxGenerationSize` trifft dasselbe zu wie bei der `MaxPopulationSize`. Da jeder Programmfehler mit 1-2 Mutationen korrigiert werden kann, führen mehrere Mutationen nur zu einem unnötigen Overhead.

Die Parameter `KeepTopIndividuumRate` und `UseTopIndividuumRate` erzielen das beste Lösungsverhalten, wenn möglichst `Individuen` mit hohem Fitnesswert in der Population gehalten (KeepTopIndividuumRate) und oft mutiert werden (UseTopIndividuumRate).

Der `UseInitialProgramRate` Parameter ist besonders kritisch. Wird dieser auf 1,0 gesetzt, so werden alle anderen Parameter der Population außer Kraft gesetzt. Es wird immer nur das ursprüngliche Programm mutiert, unabhängig von der Ausprägung der Population. Mit einem Wert von 1,0 erzielen wir hier besonders gute Werte (blaue Umrahmung in Abbilung 9.10). Mit 88 % Lösungsrate wird nahezu ein Maximum an Fehlern korrigiert und es werden durchschnittlich nur 136 Tests dazu benötigt. Alle anderen Parametrierungen, mit einer ähnlichen Lösungsrate, benötigen um ca. 30 % länger.

Antwort auf **Frage 1**: Die maximale Größe der Population kann sehr klein gehalten werden (10-20), gleiches gilt für die maximale Anzahl an Generationen, wobei hier die Auswirkung nicht wirklich gravierend ist.
Antwort auf **Frage 2**: `Individuen` mit höheren `Fitnesswerten` als andere, sollten möglichst seltener aus der Population entfernt und auch öfter mutiert werden.
Antwort auf **Frage 3**: Die Rate, mit der das ursprünglich fehlerhafte Programm mutiert wird, sollte auf jeden Fall über 0,5 gewählt werden; d.h. bei mindestens jedem zweiten Durchgang wird das ursprünglich fehlerhafte Programm mutiert, ansonsten eine Programmvariante aus der Population. Wird der Wert auf 1,0 gesetzt, so werden Lösungen am schnellsten gefunden; mit dieser Konfiguration ist es allerdings nicht mehr möglich Programme mit mehreren Fehlern zu korrigieren.

Max Population Size	Max Generation Size	Keep Top Individuum Rate	Use Top Individuum Rate	Use Initial Program Rate	Lösungen gefunden	ø Tests	σ Tests	T50	T100	T150	T200	T250	T300	T1.500
100	5	0,2	0,2	0,5	89 %	176	205	29 %	19 %	11 %	12 %	8 %	5 %	16 %
100	5	0,8	1	0,5	89 %	205	236	32 %	16 %	10 %	8 %	5 %	5 %	25 %
100	5	0,6	1	0,5	88 %	204	246	31 %	15 %	12 %	8 %	7 %	6 %	21 %
100	5	0,2	0,6	0,5	88 %	199	240	32 %	17 %	11 %	9 %	5 %	4 %	22 %
100	5	0,2	1	1	88 %	136	177	35 %	24 %	15 %	7 %	5 %	5 %	10 %
100	5	0,4	1	0,5	88 %	190	235	32 %	18 %	12 %	6 %	7 %	5 %	20 %
100	5	0,2	1	0,5	87 %	185	230	32 %	16 %	13 %	8 %	6 %	7 %	18 %
100	5	0,2	1	0,5	87 %	174	219	33 %	17 %	15 %	7 %	6 %	5 %	17 %
100	5	0,2	0,4	0,5	87 %	217	260	31 %	16 %	8 %	8 %	8 %	7 %	22 %
100	5	0,2	0,8	0,5	87 %	185	231	34 %	17 %	11 %	9 %	6 %	5 %	19 %
100	2	0	1	0,5	87 %	154	206	32 %	23 %	16 %	7 %	4 %	4 %	14 %
100	6	0	1	0,5	87 %	172	238	35 %	20 %	14 %	6 %	5 %	4 %	16 %
100	5	0,2	1	0,8	86 %	208	252	29 %	15 %	13 %	11 %	5 %	5 %	22 %
100	10	0	1	0,5	86 %	182	243	33 %	19 %	14 %	8 %	6 %	3 %	18 %
100	8	0	1	0,5	86 %	179	238	34 %	18 %	13 %	8 %	6 %	5 %	16 %
100	5	0	1	0,5	86 %	171	223	34 %	20 %	12 %	7 %	6 %	4 %	17 %
100	5	0,2	1	0,4	86 %	207	257	31 %	16 %	11 %	9 %	7 %	5 %	22 %
100	5	0,2	1	0,2	85 %	204	250	30 %	15 %	12 %	9 %	7 %	6 %	20 %
100	4	0	1	0,5	85 %	164	200	31 %	19 %	15 %	10 %	5 %	4 %	15 %
10	10	0	1	0,1	83 %	214	266	29 %	15 %	13 %	9 %	7 %	5 %	22 %
20	10	0	1	0,1	81 %	221	282	32 %	14 %	11 %	10 %	7 %	4 %	23 %
40	10	0	1	0,1	80 %	240	299	29 %	17 %	8 %	8 %	6 %	4 %	27 %
80	10	0	1	0,1	80 %	214	286	32 %	16 %	12 %	8 %	6 %	5 %	21 %
160	10	0	1	0,1	79 %	226	302	32 %	17 %	13 %	6 %	5 %	3 %	24 %
100	5	0,2	1	0,6	79 %	227	268	31 %	12 %	11 %	9 %	6 %	4 %	26 %
320	10	0	1	0,1	79 %	185	269	38 %	17 %	11 %	8 %	5 %	4 %	16 %
640	10	0	1	0,1	77 %	188	283	38 %	19 %	14 %	6 %	2 %	3 %	18 %
2560	10	0	1	0,1	77 %	150	239	43 %	21 %	11 %	6 %	4 %	2 %	13 %
1280	10	0	1	0,1	76 %	149	239	42 %	21 %	12 %	5 %	3 %	5 %	12 %
100	5	0,2	1	0	76 %	223	280	30 %	13 %	12 %	9 %	7 %	4 %	23 %

Tabelle 9.10.: Lösungsrate und durchschnittlich benötigte Tests zum Finden einer Lösung in Abhängigkeit unterschiedlicher Parametrierungen für die Population. Der blau umrahmte Bereich stellt einen Spezialfall dar: durch die `UseIinitalProgramRate` von 1,0 wird bei jeder Mutation nur das ursprünglich fehlerhafte Programm mutiert, die restlichen Parameter haben keine Relevanz und die Population wird nur zum Speichern von Mutationen aus dem ursprünglichen Programm verwendet.

9.4. Parametrierung der Fitnessfunktion

9.4.1. Fragestellung

Nachdem eine Programmvariante erzeugt wurde, wird sie mit Hilfe der Fitnessfunktion bewertet. Konnte die Programmvariante nicht kompiliert werden oder liefert sie beim Testen, aufgrund eines Timeouts, keine Werte, wird der Programmvariante ein fest parametrierbarer Wert zugewiesen (`Timout FitnessValue`, `Compile Failed FitnessValue`). Konnten die Tests durchgeführt werden, wird der Fitnesswert entsprechend der Parameter berechnet (siehe Kapitel 6.5).

Frage 1: *Wie wirken sich unterschiedliche Parametrierungen auf das Lösungsverhalten aus?*

9.4.2. Erwartetes Ergebnis

Bei allen Programmfehlern handelt es sich um Fehler, die mit einer einzigen Mutation gelöst werden können. Daher ist es eigentlich nicht notwendig, dass eine Programmvariante mit immer besser werdendem Fitnesswert zur Lösung hingeführt wird. Hart formuliert heißt das:

- Mutieren, Compilieren, Testen
- Fitnesswert == 1,0 ?
  - JA: Lösung gefunden ENDE
  - NEIN: Füge Individuum in Population ein und mutiere es vielleicht nochmal

Daher wird es für den Algorithmus in aller Regel auch egal sein, ob eine Programmvariante den Wert 0,99 oder 0,01 hat. Je höher der Wert ist, desto eher wird es weiter mutiert und verbleibt länger in der Population. Durch Variieren der Parameter für die Fitnessfunktion verschieben sich die Werte der fehlerhaften Programmvarianten zwar, die Reihenfolge bleibt jedoch gleich.

Es ist daher nicht davon auszugehen, dass unterschiedliche Parametrierungen der Fitnesswerte sich maßgeblich auf das Lösungsverhalten auswirken werden.

9.4.3. Versuchsaufbau

Hardware

- Cyria, 32 Threads
- 20x JiffyBox 2 Core, 8 Threads

Parameter

Der Fitnesswert für einen erfolgreich ausgeführten Testfall liegt immer bei 1,0. Die restlichen Werte können im Bereich von 0,0 bis 0,99 eingestellt werden.

- **Abbruchkriterien**: MaxExecutionTime: 10m, MaxFitnessEvaluations: 1.500, MaxSolutionSearch: 1
- **Timeouts**: 5s (reduziertes TestSuiteSet), 30s (vollständiges TestSuiteSet)
- **FitnessValueParameter**: jeweils 0,0/0,25/0,5/0,75

Evaluierungsläufe

- **Aktivierte Programme**: 5
- **Aktivierte Fehler**: 30
- **Parameterpermutationen**: 15
- **Iterationen**: 50
- **Evaluierungsläufe insgesamt**: $30 * 15 * 50 = 22.500$

Messpunkte

Wie in den vorherigen Beispielen werden die Lösungsquote und die durchschnittlich zur Lösung benötigten Tests als Messpunkte verwendet.

9.4.4. Ergebnis

Wie schon spekuliert, liefert das Ergebnis in Tabelle 9.11 keinen Anhaltspunkt dafür, dass sich Änderungen an den Parametern der Fitnessfunktion auf das Lösungsverhalten auswirken.

Antwort auf die **Frage 1**: Die Rate, mit der Lösungen gefunden wurde, schwankt zwischen 81 % und 84 %. Der Bereich ist zu gering um eine Aussauge darüber treffen zu können. Die Parametrierung der Fitnessfunktion hat auf das Lösungsverhalten keine erkennbare Auswirkung.

Successful Weight	Error Weight	Failed Weight	Timeout FitnessValue	Compile Failed FitnessValue	Lösungen gefunden	ø Tests	σ Tests	T50	T100	T150	T200	T250	T300	T1.500
1	0,5	0,5	0,05	0,5	84 %	118	132	39 %	21 %	15 %	9 %	4 %	3 %	9 %
1	0,5	0,5	0,05	0	83 %	152	226	37 %	23 %	13 %	6 %	3 %	3 %	14 %
1	0,5	0	0,05	0,05	83 %	115	131	36 %	25 %	13 %	9 %	5 %	4 %	8 %
1	0,5	0,25	0,05	0,05	83 %	152	208	38 %	22 %	12 %	8 %	3 %	2 %	15 %
1	0,5	0,5	0,75	0,05	83 %	154	224	39 %	22 %	13 %	5 %	4 %	2 %	14 %
1	0	0,5	0,05	0,05	83 %	155	233	36 %	25 %	12 %	7 %	5 %	2 %	12 %
1	0,5	0,5	0,05	0,05	83 %	161	233	38 %	21 %	13 %	8 %	3 %	3 %	14 %
1	0,25	0,5	0,05	0,05	83 %	158	217	38 %	22 %	12 %	6 %	4 %	3 %	16 %
1	0,5	0,5	0,25	0,05	83 %	151	214	39 %	22 %	13 %	5 %	4 %	3 %	14 %
1	0,75	0,5	0,05	0,05	82 %	152	225	37 %	22 %	13 %	8 %	4 %	4 %	12 %
1	0,5	0,5	0	0,05	82 %	152	222	37 %	25 %	12 %	7 %	4 %	2 %	13 %
1	0,5	0,5	0,05	0,25	82 %	154	231	35 %	24 %	15 %	8 %	5 %	2 %	12 %
1	0,5	0,5	0,5	0,05	82 %	158	221	36 %	22 %	13 %	7 %	4 %	4 %	13 %
1	0,5	0,5	0,05	0,75	81 %	108	112	39 %	26 %	13 %	8 %	5 %	3 %	8 %
1	0,5	0,75	0,05	0,05	81 %	150	219	42 %	21 %	10 %	7 %	3 %	2 %	15 %

Tabelle 9.11.: Lösungsrate und durchschnittlich benötigte Tests zum Finden einer Lösung in Abhängigkeit unterschiedlicher Parametrierungen für die Fitnesswerte. Da sich die Rate für die gefundenen Lösungen immer im Bereich von 81 % und 84 % bewegt, ist keine Tendenz zu erkennen.

9.5. Parametrierung der TestSuiteSets

9.5.1. Fragestellung

Bereits in den verwandten Arbeiten (siehe Kapitel 3.2) wurde darauf hingewiesen, dass die Auswahl der Testfälle für das Lösungsverhalten sehr ausschlaggebend ist. Um eine gute Performance zu gewährleisten, können nicht bei jeder Fitnesswertberechnung alle Testfälle ausgeführt werden. Daher gibt es ein reduziertes `TestSuiteSet`, bei dem nur einige wenige Tests durchgeführt werden. Dieses `TestSuiteSet` ist über 3 Parameter konfigurierbar (siehe Kapitel 7.1).
Frage 1: *Ist es ausreichend, wenn nur die negativ ausgeführten Testfälle im reduzierten `TestSuiteSet` vorhanden sind?*
Frage 2 *Können die negativ ausgeführten Testfälle reduziert werden (durch FailedTestWeight oder TimeTuning)?*
Frage 3: *Ist ein Vorteil ersichtlich, wenn auch positiv ausgeführte Testfälle im reduzierten `TestSuiteSet` vorhanden sind?*
Frage 4: *Wie weit kann mit dem `TimeTuning` Parameter das Lösungsverhalten bzw. die Ausführungszeit verbessert werden?*

9.5.2. Versuchsaufbau

Hardware

Da bei diesen Evaluierungsergebnissen auch die Ausführungszeit betrachtet wird, müssen die Evaluierungsläufe auf einer einheitlichen Hardware durchgeführt werden, ansonsten würde es zu Verfälschungen der Werte kommen.

- 50x JiffyBox 2 Core, 8 Threads

Parameter

- **Abbruchkriterien**: MaxExecutionTime: 10m, MaxFitnessEvaluations: 1.500, MaxSolutionSearch: 1
- **Timeouts**: 5s (reduziertes TestSuiteSet), 30s (vollständiges TestSuiteSet)
- **FailedTestWeight**: 0,2/0,4/0,6/0,8/1,0
- **SuccessfulTestWeight**: 0,2/0,4/0,6/0,8/1,0
- **TimeTuning**: 0,2/0,4/0,6/0,8/1,0

Evaluierungsläufe

- **Aktivierte Programme**: 5
- **Aktivierte Fehler**: 30
- **Parameterpermutationen**: 22
- **Iterationen**: 50
- **Evaluierungsläufe insgesamt**: $30 * 22 * 50 = 33.000$

Messpunkte

Wie in den vorherigen Beispielen werden die Lösungsquote und die durchschnittlich zur Lösung benötigten Tests als Messpunkte verwendet.

9.5.3. Ergebnis

Im Gegensatz zum vorherigen Evaluierungslauf können bei diesem (siehe Tabelle 9.12) wieder Erkenntnisse gewonnen werden.

Antwort auf **Frage 1**: Ja, es ist ausreichend, wenn im reduzierten `TestSuiteSet` nur negativ ausgeführte Testfälle verwendet werden. Diese Ergebnisse führen die Tabelle 9.12 an.
Antwort auf **Frage 2**: Wenn negativ ausgeführte Tests aus dem `TestSuiteSet` entfernt werden, verschlechtert sich die Lösungsrate. Es sollten daher alle negativen Lösungen für das reduzierte `TestSuiteSet` verwendet werden. Die Ausnahme bildet hier ein `TimeTuning`, dabei werden nur die langlaufenden Tests entfernt (nach der bisherigen Erfahrung sind das meistens wenige Tests, die aber überproportionale Laufzeit haben).
Antwort auf **Frage 3**: Kurz und knapp: Nein! Es verlängert sich die Dauer, bis eine Korrektur gefunden wird; besonders gut ersichtlich ist das in der Tabelle 9.12 bei den rot umrahmten Ergebnissen.
Antwort auf **Frage 4**: Der `TimeTuning` Parameter verbessert das Lösungsverhalten, er sollte aber einen Wert von 0,8 nicht unterschreiten, da dann die Lösungsrate absinkt.

Schlusslicht dieser Evaluierungen bilden die Fitnessläufe, bei denen nahezu alle Testfälle verwendet werden. Dort bricht die Rate, mit der die Lösungen gefunden werden können, auf unter 50 %. Zum einen liegt das daran, weil nun viel weniger Tests durchgeführt werden können, zum anderen daran, weil nun viele Testläufe leichter in ein Timeout laufen können.

Failed Test Weight	Successful Test Weight	Time Tuning	Lösungen gefunden	ø Dauer	σ Dauer	ø Tests	σ Tests	T50	T100	T150	T200	T250	T300	T1.500
1	0	0,8	87 %	63	108	163	245	41 %	19 %	10 %	7 %	5 %	3 %	14 %
1	0	0,4	85 %	54	82	161	234	39 %	19 %	12 %	9 %	4 %	2 %	15 %
1	0	0,6	84 %	52	89	146	209	40 %	25 %	9 %	4 %	5 %	1 %	15 %
1	0	0,2	84 %	59	98	154	217	40 %	23 %	8 %	6 %	4 %	4 %	15 %
1	0	1	84 %	58	90	164	248	37 %	22 %	14 %	7 %	4 %	2 %	14 %
1	0,2	0,4	83 %	73	107	165	249	40 %	22 %	10 %	6 %	3 %	3 %	16 %
0,8	0	1	83 %	60	98	151	224	40 %	23 %	10 %	7 %	3 %	3 %	14 %
1	0,6	0,4	82 %	104	139	138	192	45 %	19 %	9 %	6 %	4 %	4 %	13 %
1	0,2	0,8	82 %	81	115	114	139	40 %	25 %	10 %	5 %	7 %	4 %	8 %
1	0,4	0,4	82 %	78	113	133	209	49 %	19 %	11 %	5 %	2 %	2 %	12 %
1	0,8	0,4	82 %	104	128	121	155	44 %	19 %	12 %	7 %	3 %	4 %	11 %
0,6	0	1	81 %	61	91	158	215	39 %	19 %	11 %	9 %	4 %	2 %	16 %
0,2	0	1	81 %	57	93	150	233	39 %	25 %	12 %	5 %	3 %	1 %	14 %
0,4	0	1	81 %	56	85	158	237	41 %	21 %	12 %	4 %	5 %	2 %	15 %
1	0,4	0,8	73 %	92	136	97	114	45 %	26 %	11 %	5 %	4 %	4 %	6 %
1	0,6	0,8	70 %	102	139	101	123	46 %	24 %	10 %	5 %	5 %	2 %	9 %
1	0,2	1	70 %	85	147	105	140	49 %	22 %	10 %	4 %	3 %	1 %	11 %
1	0,8	0,8	64 %	94	119	90	99	48 %	25 %	11 %	3 %	4 %	2 %	7 %
1	0,4	1	59 %	78	120	89	104	48 %	24 %	11 %	6 %	3 %	3 %	5 %
1	0,6	1	52 %	93	131	90	103	39 %	33 %	13 %	6 %	3 %	1 %	4 %
1	1	1	48 %	109	132	90	98	50 %	20 %	13 %	8 %	2 %	2 %	5 %
1	0,8	1	45 %	93	112	100	108	42 %	26 %	13 %	6 %	6 %	1 %	6 %

Tabelle 9.12.: Lösungsrate und durchschnittlich benötigte Tests zum Finden einer Lösung in Abhängigkeit unterschiedlicher Parametrierungen für das Test Set. Der rot umrahmte Bereich entspricht jenen Parametrierungen, bei denen über 60 % der positiven Testfälle im reduzierten `TestSuiteSet` verwendet werden. Die Lösungsrate verbessert sich in diesen Fällen nicht, die durchschnittliche Dauer zum Finden einer Lösung ist jedoch deutlich im Bereich von 100 Sekunden; werden keine oder wenige positive Testfälle verwendet, ist die Dauer im Bereich von 60 Sekunden und damit um 40 % besser.

9.6. Evaluierung der Performance

9.6.1. Fragestellung

Theads zu CPUs Verhältnis

Am Ende stellt sich die Frage, wie die beste Performance mit der `JCC Engine` erzielt werden kann. Während der gesamten Implementierung wurde darauf geachtet, dass die Applikation möglichst gut mit mehreren CPU Cores zusammenarbeiten kann.
Frage 1: *Resultieren mehrere Threads immer in einer steigenden Performance oder gibt es dabei auch Grenzen?*
Frage 2: *Wie groß ist das ideale Verhältnis von Threads zu CPUs?*

Ökonomischer Aspekt

Um die Evaluierungen möglichst rasch abschließen zu können, wird teilweise auf Cloud Computing zurückgegriffen. Der gewählte Anbieter `JiffyBox` bietet `Cloud Server` mit unterschiedlichen Konfigurationen und Preisen an. Da die Anzahl der Server, die gebucht werden können, im Prinzip unbegrenzt ist, möchte man für die Evaluierung je eingesetztem Euro die maximal mögliche Rechenleistung haben. Im Vorfeld fiel die Wahl auf 2 unterschiedliche Modelle (siehe Tabelle 9.13).
Frage 3: *Kann mit Servern der Variante 2, die die doppelten Kosten je Server verursachen, auch die doppelte Rechenleistung gegenüber den Servern der Variante 1 erzielt werden?*

Variante	RAM	CPUs	HDD	Preis [€/h]
1	1GB	2	50GB	0,02
2	2GB	4	100GB	0,04

Tabelle 9.13.: Verfügbare Leistungsstufen der JiffyBox

9.6.2. Erwartetes Ergebnis

Bei **Cyria** mit den 4 physikalischen und 8 virtuellen CPU Cores würde erwartet werden, dass die Performance bei Parallelverarbeitung mit bis zu 4 Threads in etwa linear steigt und sich mit 8 Threads weiter, allerdings nicht mehr so stark wie vorher, verbessert. Ab 16 Threads wird der Overhead für die Threadsteuerung die Performance einbrechen lassen und die Performance wird mit steigender Thread Anzahl leicht schlechter werden.

Bei **JiffyBox 2 bzw. 4 Core** wird ein ähnliches Ergebnis erwartet. Da es keine virtuellen CPU Cores gibt, wird die maximale Performance dann gegeben sein, wenn die Anzahl der Threads gleich die Anzahl der CPUs ist. Darüber hinaus wird die Performance aufgrund des Overheads, der durch die Threadsteuerung anfällt, schlechter werden.

Der **JiffyBox 4 Core** Server sollte mindestens doppelt so schnell sein wie der **JiffyBox 2 Core** Server bzw. ihn sogar übertreffen, da ihm auch der doppelte Arbeitsspeicher zur Verfügung steht. Daher wird es ökonomischer sein, weniger Server mit mehr Performance zu höheren Preisen für die Evaluierung zu mieten.

9.6.3. Versuchsaufbau

Hardware

- Cyria
- 1x JiffyBox 2 Core
- 1x JiffyBox 4 Core

Parameter

- **Threads**: 7 unterschiedliche Einstellungen (1, 2, 4, 8, 16, 32, 64)
- **Abbruchkriterien**: MaxExecutionTime: 60s
- **Timeouts**: 5s (reduziertes TestSuiteSet), 30s (vollständiges TestSuiteSet)

Evaluierungsläufe

- **Aktivierte Programme**: Alle (6)
- **Aktivierte Fehler**: Alle (79)
- **Parameterpermutationen**: 7
- **Iterationen**: 25
- **Evaluierungsläufe insgesamt**: $79 * 7 * 25 = 13.825$

Messpunkte

Als Messpunkte dienen die Anzahl der durchgeführten Mutationen, Kompilierungen und Tests. Um eine bessere Visualisierung zu erreichen, wurden alle Werte durch den kleinsten Wert in der Messreihe dividiert; damit soll eine bessere Visualisierung erreicht werden, da bei dieser Evaluierung nur das Verhältnis der Messwerte zueinander interessant ist und nicht die absoluten Werte. So wird erreicht, dass der kleinste Wert immer 1,0 ist und die anderen Werte der Messreihe ein entsprechendes Vielfaches davon sind. Es ist das Verhältnis von z.B. 1,0 zu 2,4 in der Tabelle einfacher lesbar, als 13.458 zu 32.399; die Aussage, in Hinblick auf das Verhältnis der Performance, ändert sich dadurch nicht.

9.6.4. Ergebnis

Theads zu CPUs Verhältnis

Zuerst wird das Verhältnis der Performance in Abhängigkeit der Threads je Programmschritt und Evaluierungsumgebung betrachtet (siehe Abbildung 9.2). Es wäre zu erwarten gewesen, dass die einzelnen Programmschritte (`MUTATE`, `COMPILE` und `TEST`) im selben Verhältnis steigen. Das ist aber nur bei `COMPILE` und `TEST` der Fall. `MUTATE` steigt besonders zu Beginn (**Cyria**, 1-8 Threads) stärker als die beiden anderen. Das liegt daran, dass durch mehr Threads auch mehr Durchläufe ausgeführt werden können. Je mehr Durchläufe ausgeführt werden, desto mehr Programmvarianten waren bereits in der Population und werden bei erneutem Auftreten verworfen (siehe Kapitel 6.7). Dadurch werden die Schritte `COMPILE` und `TEST` übergangen. Wenn durch zusätzliche Threads keine Verbesserung der Performance mehr erzielt werden kann, bleibt dieser Wert konstant (**Cyria**, 16-64 Threads).

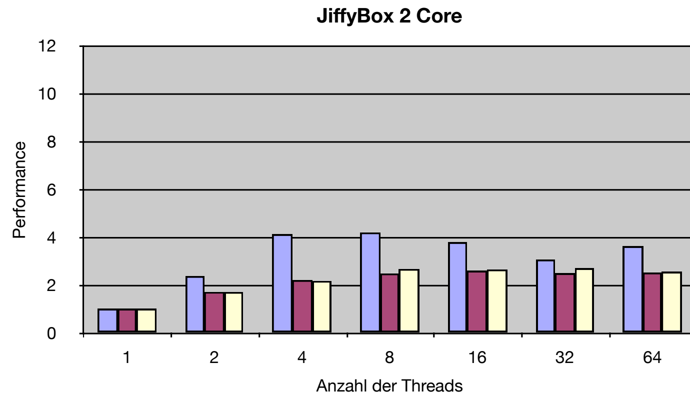
Cyria skaliert mit den Threads sehr gut. Bis zu 16 Threads stellt sich je Verdoppelung der Threads eine Verbesserung der Performance ein; danach steigt die Performance nur mehr minimal. Das Ergebnis ist besser als erwartet (siehe Kapitel 9.6.2) und es zeigt sich, dass man Evaluierungen auf **Cyria** durchaus mit 32 oder sogar 64 Threads durchführen sollte um die optimale Performance zu erzielen. Anschließend wurden noch Versuche mit 128 und 256 Threads durchgeführt, das führte die `Java VM` allerdings an ihre Grenzen und die `JCC Engine` konnte nicht mehr stabil betrieben werden.

JiffyBox 2/4 Core liefern ein ähnliches Bild, dort ist bei 8/16 Threads das Maximum erreicht. Die Ergebnisse werden teilweise minimal schlechter, wenn die Anzahl der Threads weiter erhöht werden. Es zeigt sich auch hier, dass die Performance von bis zu 4 mal so vielen Threads, wie es CPUs gibt, profitieren kann. Die Annahme, die maximale Anzahl an Threads soll die Anzahl der CPUs sein, war nicht richtig (siehe Kapitel 9.6.2).

Antwort auf **Frage 1**: Mehr Threads bringen so lang eine Steigerung der Performance, bis der Overhead für die Threadverwaltung zu groß wird. Werden die Threads weiter gesteigert (128/256), kam es auf den getesteten Evaluierungsumgebungen zusätzlich zu Problemen mit der Stabilität.
Antwort auf **Frage 2**: Mit `#CPUs * 4 = #Threads` konnten im Durchschnitt die besten Ergebnisse hinsichtlich der Performance erzielt werden.

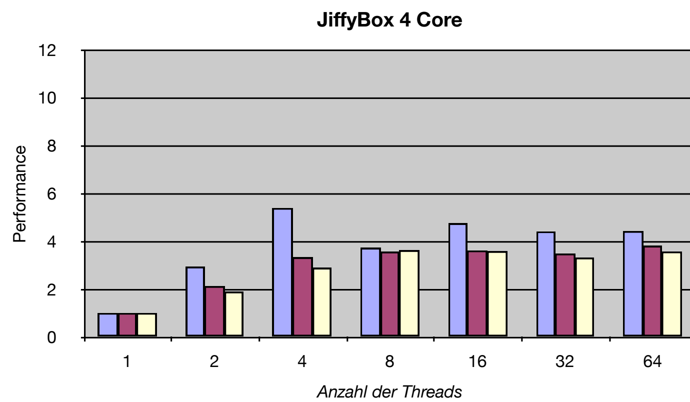
JiffyBox 2 Core

Threads	MUTATE	COMPILE	TEST
1	1,0	1,0	1,0
2	2,4	1,7	1,7
4	4,1	2,2	2,2
8	4,2	2,5	2,7
16	3,8	2,6	2,6
32	3,1	2,5	2,7
64	3,6	2,5	2,5



JiffyBox 4 Core

Threads	MUTATE	COMPILE	TEST
1	1,0	1,0	1,0
2	2,9	2,1	1,9
4	5,4	3,3	2,9
8	3,7	3,6	3,6
16	4,7	3,6	3,6
32	4,4	3,5	3,3
64	4,4	3,8	3,6



Cyria 8 Core

Threads	MUTATE	COMPILE	TEST
1	1,0	1,0	1,0
2	2,4	2,0	1,9
4	4,8	3,8	3,3
8	7,0	5,7	5,0
16	9,9	7,6	6,6
32	9,9	7,9	7,2
64	10,2	8,0	7,6

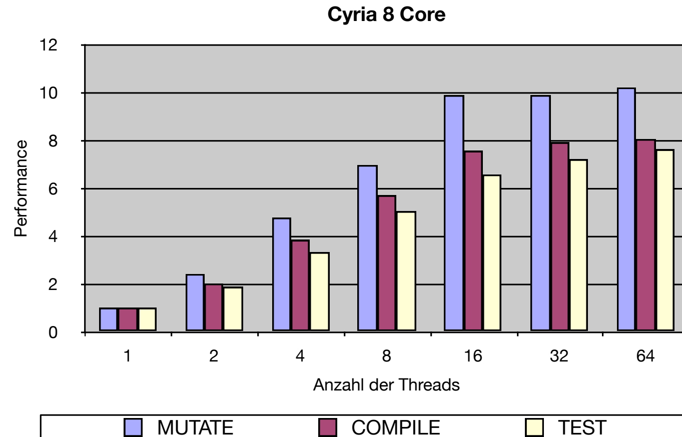
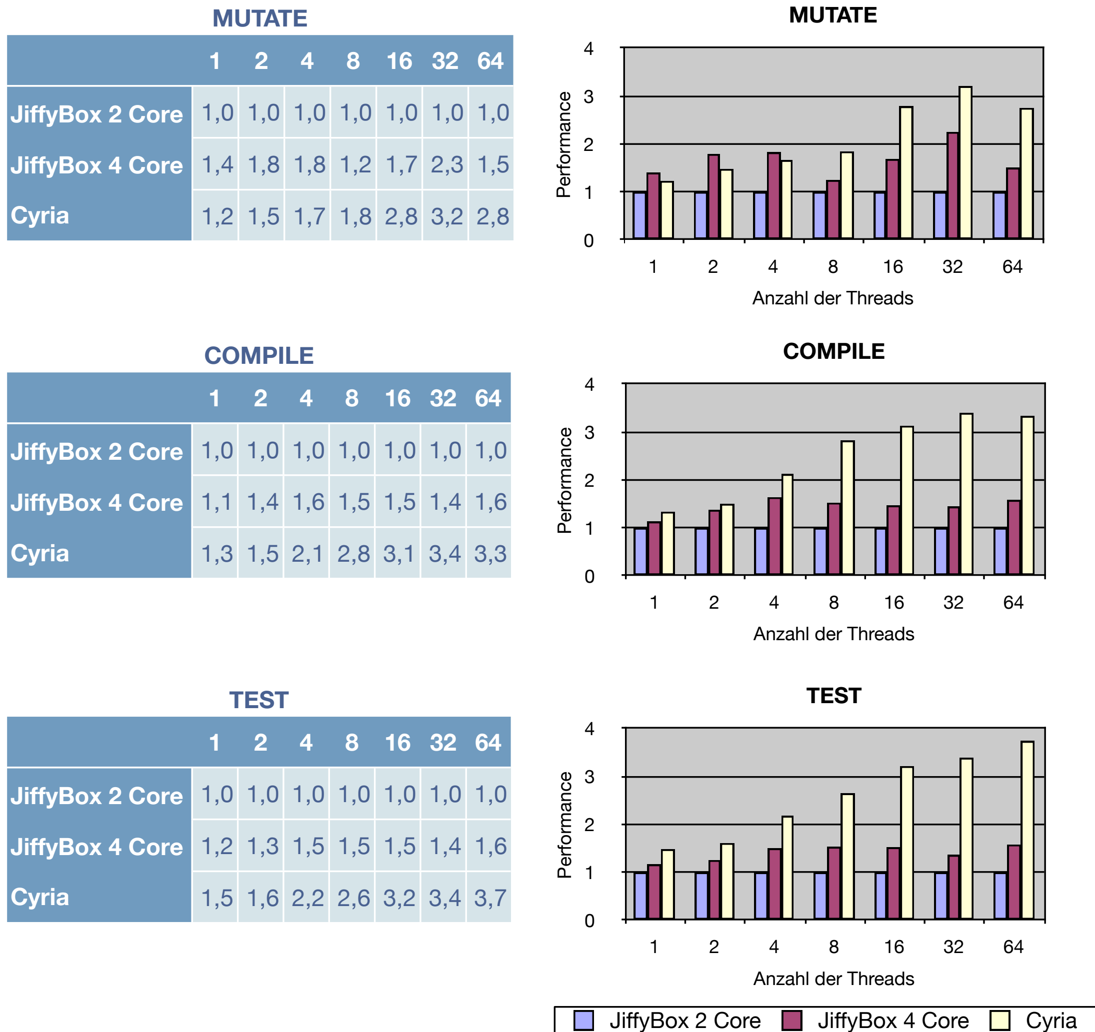


Abbildung 9.2.: Die Tabellen und Diagramme zeigen je Evaluierungsumgebung (Cyria, JiffyBox 2/4 Core), wie sich die Performance, in Abhängigkeit der verwendeten Threads, verhält. Die Performance wird je Programmschritt (MUTATE, COMPILE, TEST) angegeben. *Zum Beispiel für Evaluierungsumgebung JiffyBox 2 Core und Programmschritt TEST: Bei 2 Threads wurden 1,7 mal so viele Tests durchgeführt als bei nur einem Thread, bei 8 Threads waren es 2,7 mal so viele.*

Ökonomischer Aspekt

Abbildung 9.3 beantwortet **Frage 3** hinsichtlich der Serverwahl in Hinblick auf optimale Nutzen/Kosten Relation. Dabei wird je Schritt (`MUTATE`, `COMPILE` und `TEST`) ein Vergleich der Rechner hinsichtlich der



MUTATE

	1	2	4	8	16	32	64
JiffyBox 2 Core	1,0	1,0	1,0	1,0	1,0	1,0	1,0
JiffyBox 4 Core	1,4	1,8	1,8	1,2	1,7	2,3	1,5
Cyria	1,2	1,5	1,7	1,8	2,8	3,2	2,8

COMPILE

	1	2	4	8	16	32	64
JiffyBox 2 Core	1,0	1,0	1,0	1,0	1,0	1,0	1,0
JiffyBox 4 Core	1,1	1,4	1,6	1,5	1,5	1,4	1,6
Cyria	1,3	1,5	2,1	2,8	3,1	3,4	3,3

TEST

	1	2	4	8	16	32	64
JiffyBox 2 Core	1,0	1,0	1,0	1,0	1,0	1,0	1,0
JiffyBox 4 Core	1,2	1,3	1,5	1,5	1,5	1,4	1,6
Cyria	1,5	1,6	2,2	2,6	3,2	3,4	3,7

Abbildung 9.3.: Die Tabellen und Diagramme zeigen je Programmschritt (MUTATE, COMPILE, TEST) wie sich die Performance, in Abhängigkeit der Evaluierungsumgebung (Cyria, JiffyBox 2/4 Core), verhält. Die Performance wird je Thread (1-64) angegeben. *Zum Beispiel für Programmschritt MUTATE und 1 bzw. 32 Threads: Bei 1 Thread wurden mit Cyria 1,2 mal so viele Mutationen durchgeführt als mit der JiffyBox 2 Core, bei 32 Threads waren es 3,2 mal so viele.*

Threads durchgeführt.

Erwartet wird, dass die **JiffyBox** mit 4 CPUs mindestens doppelt so schnell ist, wie jene mit 2 CPUs. Die **JiffyBox 4 Core** kommt zwar beim `MUTATE` Schritt in die Nähe von Faktor 2, bei den wichtigeren `COMPILE` und `TEST` Schritten erreicht sie jedoch nur einen Faktor von 1,4. Damit steht fest, dass die doppelte Anzahl an CPUs und RAM den doppelten Preis nicht aufwiegen können. Für alle weiteren Evaluierungen wird, wenn besonders viel Rechenleistung benötigt wird, die **JiffyBox 2 Core** verwendet.

Je nach betrachtetem Programmschritt, erbringt **Cyria** die maximal 3,7 fache Performance gegenüber einer **JiffyBox 2 Core**; mit den maximal 50 zur Verfügung stehenden Instanzen kann somit in Summe die 13 fache Leistung gegenüber dem lokalen Rechner zur Verfügung gestellt werden. Würde man die Ökonomie vernachlässigen, könnte mit teureren **JiffyBox** Servern noch mehr Leistung erzielt werden.

Kapitel 10

Erkenntnisse

Die Erkenntnisse, die aus der bisherigen Arbeit gewonnen wurden, werden in diesem Kapitel nochmals zusammengefasst. Am Wichtigsten sind dabei die Erkenntnisse aus der Evaluierung, auf die in Abschnitt 10.1 eingegangen wird. Aber auch der Evaluierungsprozess an sich lieferte einige Erkenntnisse, die in Abschnitt 10.2 betrachtet werden. Ein nicht zu geringer Teil der Arbeit bestand in der Implementierung der `JCC Engine`. Über die daraus gewonnen Erkenntnisse wird am Ende des Kapitels in Abschnitt 10.3 resümiert.

10.1. Erkenntnisse aus dem Ergebnis

`Julius Cäsar` sagte:

> Veni, Vidi, Vici.

Ich sage zusammenfassend zu den Ergebnissen:

> Ich suchte, ich fand und reparierte die Fehler - zumindest einige davon.

In den ersten Evaluierungen wurde die Frage geklärt, welche Fehler überhaupt gefunden werden können. Von den Programmfehlern mussten diejenigen ausgeschlossen werden, die nicht durch das Fehlschlagen eines Testfalls erkannt werden konnten. Das war aber auch nicht die Anforderung an die `JCC Engine`. Von den verbliebenen 61 Programmfehlern konnten 30 sehr zuverlässig korrigiert werden, das entspricht 49 %. Die Erkennungsrate bei diesen korrigierbaren Fehlern lag durchwegs im Bereich von 100 %, fiel vereinzelt auf 80 % und in wenigen Fällen auf 50 % ab.

Die 31 nicht korrigierbaren Fehler sollten durch Optimieren und Erweitern der `Mutationsoperatoren` zu einem großen Teil korrigierbar werden. Zuverlässigere Aussagen kann nur eine weitere Evaluierung bieten.

Bei der Evaluierung der Parametereinstellungen hat sich zusammengefasst Folgendes ergeben:

- Die Gewichtung der `Mutationsparameter` kann gleichmäßig über alle erfolgen. Der `DELETE` Operator kann reduziert oder ganz entfernt werden. Der `CHANGE` Operator kann im Gegenzug leicht erhöht werden.
- Bei der `Population` erzielt man die besten Werte, wenn man sie gar nicht verwendet und immer nur das ursprüngliche Programm mutiert. Dass würde sich aber bei komplexeren Programmfehlern ändern, diese könnten mit dieser Parametrierung überhaupt nicht gefunden werden.
- Änderungen der Parameter für die `Fitness Funktion` wirken sich nicht nennenswert auf das Lösungsverhalten aus.

- Das reduzierte `TestSuiteSet` sollte nur aus zu Beginn fehlgeschlagenen Testfällen bestehen. Testfälle mit langer Laufzeit sollten entfernt werden.

10.2. Erkenntnisse aus dem Evaluierungsprozess

Das Evaluieren der `JCC Engine` war ein aufwendiger, aber spannender Prozess. Da viele Daten angefallen sind und eine große Rechenleistung benötigt wurde, musste im Vorfeld eine entsprechende Infrastruktur geschaffen werden (siehe Kapitel 8). Das führte sogar zu einer E-Mail von meinem Provider der Evaluierungsdatenbank, in der nachgefragt wurde, ob es wohl mit rechten Dingen zuginge, dass zugleich 50 unterschiedliche Server auf die Datenbank zugreifen.

Ein Großteil der Berechnungen und Daten fiel bei der Bestimmung der optimalen Parametrierung der `JCC Engine` an. Leider war die Streuung der Daten so groß, dass meistens nur Tendenzen abgelesen werden konnten und keine eindeutigen Ergebnisse.

In Summe sind bei der Evaluierung über 250.000 Evaluierungsläufe durchgeführt worden, die über 1 GB an Daten produziert haben.

10.3. Erkenntnisse aus der Implementierung

Bei der Implementierung wurde man vor zahlreiche Herausforderungen gestellt. Es wurde sich mit vielen, bisher nicht benötigten, Bereichen auseinandersetzen. Darunter die Repräsentation und Bearbeitung von Sourcecode in einem `AST` und das Compilieren und Ausführen eines Programmes zur Laufzeit innerhalb eines anderen Programmes.

Eine besondere Herausforderung war es auch, dass sämtliche Komponenten threadsicher sein mussten, um die Arbeiten parallel ausführen zu können. Auch die eigentliche Parallelverarbeitung der Daten und deren Zusammenführung hatte einige Tücken parat.

Schließlich war auch immer der Druck gegeben, die Performance möglichst hoch zu halten. Besonders bei der Testfallausführung und beim Compilieren wurden einige unterschiedliche Ansätze untersucht, es scheint hier aber noch viel Potenzial zu geben (siehe Kapitel 11).

Kapitel 11

Ausblick

Am Ende wird noch ein kleiner Ausblick gegeben, wie auf diese Arbeit weiter aufgebaut werden kann. Aus der Evaluierung (siehe Kapitel 8) ist bekannt, dass nicht alle Fehler mit der aktuellen Implementierung korrigiert werden konnten. In Abschnitt 11.1 wird beschrieben, wie man das verbessern könnte. Zugleich sollte auch die Evaluierung erweitert werden, darauf wird in Abschnitt 11.2 eingegangen. Eine Lösung, um die `JCC Engine` in der Praxis besser einsetzten zu können, wird in Abschnitt 11.3 beschrieben. Schließlich sind auch während der Implementierung viele Ideen aufgetaucht, denen sich Abschnitt 11.4 widmet.

11.1. Erweitern der Mutationsoperatoren

Bei den Ergebnissen der Evaluierung wurde festgestellt, dass mit den aktuell implementierten `Mutationsoperatoren` nicht alle Fehler, die evaluiert wurden (siehe Kapitel 9.1), gefunden werden konnten. Hier sollte nachgebessert werden, um eine möglichst hohe Lösungsrate zu bekommen.

11.2. Evaluierungen mit komplexeren Fehlern durchführen

Bei der Evaluierung hat sich gezeigt, dass sich das Konzept der genetischen Programmierung bei einfachen Fehlern, die mit einer Mutation gelöst werden können, nicht vollständig entfalten konnte (siehe Kapitel 9.3). Dazu sollte eine Evaluierungsumgebung mit Programmfehlern geschaffen werden, für deren Korrektur mehrere Mutationen in Folge notwendig sind.

11.3. Einbinden der JCC Engine in eine IDE

Um die `JCC Engine` in der Praxis besser und leichter anwenden zu können, sollte sie in eine IDE wie Eclipse integriert werden. Dort könnte automatisch nach einem JUnit Testlauf, beim Auftreten von Fehlern, ein automatischer Korrekturversuch gestartet werden. Auch die Visualisierung könnte in der IDE einfach durchgeführt werden.

11.4. Verbesserungen bei der Implementierung

11.4.1. Entfernen von Einschränkungen

Oft können Implementierungen im ersten Schritt nicht bestmöglich umgesetzt werden, weil entweder Zeit oder die Technik dazu fehlt. Diese Punkte sind dann oft Einschränkungen in der Lösung (siehe Kapitel 5.5). Je nachdem, in welche Richtung sich das Programm weiter entwickeln soll, können nach und nach Lösungen für diese Einschränkungen gesucht werden.

11.4.2. Verbesserter Umgang mit Timeouts bei Testfällen

Die Durchführung der Tests einer Programmvariante ist durch ein Timeout limitiert. Wird dieses Timeout ausgelöst, wird das Testen abgebrochen ohne Ergebnisse zurückzuliefern. Ein Timeout tritt meist dann auf, wenn ein Testfall eine Endlosschleife produziert hat. Von dieser Endlosschleife müssen nicht alle Tests betroffen sein, es können durchaus einige noch ausgeführt und somit für diese Programmvariante ein Fitnesswert berechnet werden. Zwei Lösungsvarianten seien kurz vorgestellt:

- Jeder Testfall für sich wird in einem eigenen Thread ausgeführt. Das wäre einfach zu realisieren, schlägt sich aber sehr schlecht in der Performance nieder, weil für jeden Test das JUnit Framework neu initialisiert werden muss.
- Die Tests werden wie bisher im selben Thread ausgeführt. Tritt ein Timeout auf, werden die bereits ausgeführten Tests deaktiviert (inklusive dem Letzten, von dem das Timeout verursacht worden ist). Die verbleibenden Tests werden erneut ausgeführt.

Welche der beiden Varianten besser ist, müsste mit einfachen Tests und Evaluierungen geprüft werden. Mit dieser Änderung könnten auch fehlerhafte Programme korrigiert werden, bei denen der Fehler eine Endlosschleife verursacht.

11.4.3. Verbessern der Performance

Ein weiteres Verbesserungspotenzial besteht in der Performance. Dazu ist es notwendig, den Engpass zu finden, nur dort macht es meistens Sinn nach Verbesserungen zu suchen. Die Laufzeit des Programmes teilt sich in zwei große Bereiche auf:

- Initialisierung
- Wiederkehrendes Mutieren, Kompilieren, Testen, Evaluieren

Die Initialisierung benötigt meist eine konstante Zeit, je nach Größe des Programms, das untersucht wird. Verbesserungen sind hier sicher vereinzelt möglich, der Anteil an der gesamten Performance, der verbessert werden kann, ist aber limitiert. Bei den wiederkehrenden Schritten zeigt sich (siehe Kapitel 8), dass das Testen und Compilieren mit Abstand am meisten Zeit benötigt. Für beide Punkte gibt es Verbesserungspotenzial.

Testfallausführung

Dass die Durchführung der Tests viel Zeit benötigt, liegt zum einen daran, dass Tests, aufgrund von Berechnungen und sonstigen Aktionen die durchgeführt werden, eine gewisse Zeit benötigen . Hier kann nur gegensteuert werden, indem man bewusst versucht, langlaufende Tests zu identifizieren und diese nicht für die Fitnessfunktion verwendet. Dabei ist natürlich zu beachten, dass die Grundfunktionalität nicht gefährdet wird.

Zum anderen konnte in einfachen Tests erkannt werden, dass viel Zeit durch das Verwerfen und neu Erstellen des Classloaders (siehe Kapitel 6.1) verbraucht wird. Da Java leider das Austauschen einer bereits

geladenen Klasse nicht erlaubt, ist das die einzige Möglichkeit das Programm zu Laufzeit zu ändern und neu auszuführen. Es gab diesbezüglich Recherchen, es konnte aber keine bessere Lösung dafür gefunden werden. Zwar gibt es einige Ansätze [*], doch konnten erste Tests damit zu keinen Verbesserungen führen. Ein direktes Eingreifen und Manipulieren der `Java VM` wäre notwendig, um vielleicht bessere Ergebnisse zu erzielen, was allerdings den Rahmen dieser Arbeit sprengen würde.

Eine weitere mögliche Verbesserung könnte folgender Ansatz bringen:

- In einem einmaligen Vorverarbeitungsschritt wird der Source Code analysiert und sämtliche Klassen, die später potenziell geändert werden können, identifiziert.
- Aus diesen Klassen wird jeweils die Funktionalität in den Methoden entfernt und in eine eigene Klasse verschoben.
- Dort, wo bisher in der Original-Klasse die Funktionalität implementiert war, wird ein Code eingefügt, der dynamisch die Klasse aufrufen kann, die die Funktionalität nun enthält.
- Möchte man zur Laufzeit nun die Funktionalität ändern, so muss man die Klasse mit der Funktionalität nur kopieren, anpassen, kompilieren und zusätzlich laden.
- Der originalen Klasse wird dynamisch mitgeteilt, welche Klasse nun für die Abarbeitung der Funktionalität im jeweiligen Durchgang verantwortlich ist.

Diese Implementierung hat zwei entscheidende Vorteile:

- Der Classloader muss nicht immer komplett neu initialisiert werden. Es muss nur jedes Mal eine kleine Klasse neu geladen werden.
- Zusätzlich kann die Zeit für das Compilieren verringert werden, weil immer nur ein kleiner Teil mit der geänderten Funktionalität neu kompiliert werden muss und nicht die gesamte Klasse.

Leider bringt diese Implementierung auch Einschränkungen mit sich:

- Es können nur Änderungen durchgeführt werden, die sich innerhalb einer Methode befinden. Es können z.B. keine globalen oder Klassenvariablen geändert werden.
- Nach einer gewissen Zeit müsste trotzdem der Classloader entsorgt und neu geladen werden, da es sonst zu einem Memory Overflow kommen würde. In Relation zum erwarteten Performance-Zugewinn, sollte dieser Umstand allerdings vernachlässigbar sein.

Da dieser Ansatz recht komplex und aufwendig ist, wurde er nicht weiter verfolgt. Er bietet aber für die Zukunft sicher einen interessanten Anknüpfungspunkt, um die Performance zu verbessern.

Kompilieren

In der aktuellen Implementierung wird das Kompilieren immer mit dem Eclipse Compiler durchgeführt. Möglich wäre es auch, direkt den JAVA Compiler zu verwenden, der seit JAVA 1.5 zur Verfügung steht. Ob das Verbesserungen an der Performance bringt, ist unklar und müsste geprüft werden. Des Weiteren gibt es noch einige Bibliotheken, die ebenfalls Code zur Laufzeit kompilieren können, und das sogar auf Methodenebene (z.B. `JavaAssist`[†]). Wenn jeweils nicht mehr das komplette File bzw. die Klasse neu kompiliert werden muss, sondern nur der Bereich bzw. die Funktion die sich geändert hat, könnte das auch zu einem weiteren Performance Gewinn beim Compilieren führen.

Bei der zuletzt beschriebenen Variante der Testfallausführung würde sich eine zusätzliche Verbesserung der Performance automatisch einstellen, da der Bereich, der je Durchgang neu kompiliert werden muss, stark reduziert ist.

[*] http://zeroturnaround.com/jrebel/, http://download.oracle.com/javase/1.4.2/docs/guide/jpda/enhancements.html
[†] www.javassist.org/

Literaturverzeichnis

[1] Rui Abreu, Wolfgang Mayer, Markus Stumptner, and Arjan J. C. van Gemund. Refining spectrum-based fault localization rankings. In *Proceedings of the 2009 ACM symposium on Applied Computing*, SAC '09, 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 *Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing*, PRDC '06, pages 39–46, Washington, DC, USA, 2006. IEEE Computer Society.

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

[4] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. An observation-based model for fault localization. In *Proceedings of the 2008 international workshop on dynamic analysis: held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2008)*, WODA '08, pages 64–70, New York, NY, USA, 2008. ACM.

[5] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, New York, NY, USA, 1 edition, 2008.

[6] Andrea Arcuri. On the automation of fixing software bugs. In *Companion of the 30th international conference on Software engineering*, ICSE Companion '08, pages 1003–1006, New York, NY, USA, 2008. ACM.

[7] Andrea Arcuri. Evolutionary repair of faulty software. *Appl. Soft Comput.*, 11:3494–3514, June 2011.

[8] Andrea Arcuri and Xin Yao. A novel co-evolutionary approach to automatic software bug fixing. In *In Proceedings of the IEEE Congress on Evolutionary Computation (CEC 08*, pages 162–168. IEEE Computer Society, 2008.

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

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

[11] Vidroha Debroy and W. Eric Wong. Using mutation to automatically suggest fixes for faulty programs. *Software Testing, Verification, and Validation, 2008 International Conference on*, 0:65–74, 2010.

[12] Hyunsook Do, Sebastian G. Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal*, 10(4):405–435, 2005.

[13] Ethan Fast, Claire Le Goues, Stephanie Forrest, and Westley Weimer. Designing better fitness functions for automated program repair. In *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, GECCO '10, pages 965–972, New York, NY, USA, 2010. ACM.

[14] Stephanie Forrest, ThanhVu Nguyen, Westley Weimer, and Claire Le Goues. A genetic programming approach to automated software repair. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, GECCO '09, pages 947–954, New York, NY, USA, 2009. ACM.

[15] Tom Janssen, Rui Abreu, and Arjan J.C. van Gemund. Zoltar: a spectrum-based fault localization tool. In *SINTER '09: Proceedings of the 2009 ESEC/FSE workshop on Software integration and evolution @ runtime*, pages 23–30, New York, NY, USA, 2009. ACM.

[16] Barbara Jobstmann, Stefan Staber, Andreas Griesmayer, and Roderick Bloem. Finding and fixing faults. *Journal of Computer and System Sciences (JCSS)*, –, 2011. To appear.

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

[18] Thomas Kuhn and Olivier Thomann. Abstract syntax tree. `http://www.eclipse.org/articles/article.php?file=Article-JavaCodeManipulation_AST/index.html`, 2006.

[19] Manoel Marques. Exploring eclipse's astparser. `http://www.ibm.com/developerworks/opensource/library/os-ast/`, 2005.

[20] Naoya Maruyama and Satoshi Matsuoka. Model-based fault localization in large-scale computing systems. In *IPDPS*, pages 1–12, 2008.

[21] Glenford J. Myers and Corey Sandler. *The Art of Software Testing*. John Wiley & Sons, 2004.

[22] Ron Patton. *Software Testing*. Sams, Indianapolis, IN, USA, 2000.

[23] Goutam Kumar Saha. Understanding software testing concepts. *Ubiquity*, 2008:2:1–2:1, February 2008.

[24] Eric Schulte, Stephanie Forrest, and Westley Weimer. Automated program repair through the evolution of assembly code. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ASE '10, pages 313–316, New York, NY, USA, 2010. ACM.

[25] Westley Weimer, Stephanie Forrest, Claire Le Goues, and ThanhVu Nguyen. Automatic program repair with evolutionary computation. *Commun. ACM*, 53:109–116, May 2010.

[26] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 364–374, Washington, DC, USA, 2009. IEEE Computer Society.

[27] Andreas Zeller. Yesterday, my program worked. today, it does not. why? In *Joint 7th European Software Engineering Conference (ESEC) and 7th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-7)*, volume 1687, pages 253–267, Toulouse, France, September 1999.

[28] Andreas Zeller. Automated debugging: Are we close? *IEEE Computer*, pages 26–31, November 2001.

[29] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, 28:2002, 2002.