Technische Universität Braunschweig

Masterarbeit

Ein Erfahrungsbericht zur Deduktiven Verifikation mit KeY

Autor:

Carsten Immanuel Pardylla

01. Dezember 2017

Betreuer:

Prof. Dr. Ina Schaefer, Dr. Thomas Thüm, Alexander Knüppel

Institut für Softwaretechnik und Fahrzeuginformatik

Prof. Dr. Reiner Hähnle, Dominic Steinhöfel

Technische Universität Darmstadt, Institut für Softwaretechnik

https://dx.doi.org/10.24355/dbbs.084-201801311013

Pardylla, Carsten Immanuel:
Ein Erfahrungsbericht zur Deduktiven Verifikation mit KeY
Masterarbeit, Technische Universität Braunschweig, 2017.

https://dx.doi.org/10.24355/dbbs.084-201801311013

Inhaltsangabe

Die Qualität von Software hängt direkt von ihrer Zuverlässigkeit ab. Um diese sicherzustellen, kommen verschiedene Techniken zum Einsatz. Die bekanntesten Techniken hierzu sind testen und Code-Reviews, diese sind jedoch nicht formal. Deduktive Verifikation stellt eine statische, formale Alternative hierzu dar. Der Vorzug dieser Lösung ist, dass nicht einzelne Ausführungen getestet werden, sondern dass das korrekte Verhalten mit Hilfe von Deduktionen bewiesen werden kann. Trotz dieses großen Vorzuges wird die Technik jedoch scheinbar nur spärlich eingesetzt. In dieser Arbeit wird deshalb das Vorgehen mit KeY, einem Werkzeug zur deduktiven Verifikation von Java-Quelltext, an Hand der bestehenden Java Platform API exemplarisch vorgenommen, um den Aufwand einer solchen Verifikation zu betrachten und Probleme und Lösungen zu erkennen.

Hierzu werden zunächst einzelne Methoden der Java-Platform-API spezifiziert und mit KeY verifiziert. Die Erfahrungen aus dieser Arbeit dienen späteren *Anwendern*, wenn sie ähnliche Fälle betrachten, und den Entwicklern von KeY, um die Verifikation zu vereinfachen. Da die Wahl der richtigen Optionen für die Beweisbarkeit und den Beweisaufwand entscheidend ist, spielen diese bei der Arbeit mit KeY eine zentrale Rolle. Deshalb werden aus vorliegenden Beschreibungen Hypothesen entwickelt, die anschließend mit Hilfe der zuvor erstellten Spezifikationen quantitativ untersucht werden. Um die Untersuchung durchzuführen, wurden zahlreiche Verifikationen durchgeführt. Die im Rahmen dieser Experimente ermittelten Daten werden ebenfalls genutzt, um mit Hilfe von SPLConqueror ein Modell zu berechnen, dass den Aufwand einer Verifikation in Abhängigkeit ihrer Optionen beschreiben soll. Mit Hilfe der Hypothesen, dem Modell und den Erfahrungen aus der Spezifikation werden Empfehlungen für Anpassungen an KeY erarbeitet. Die Hypothesen können außerdem zukünftige Anwender bei der Auswahl der richtigen Optionen unterstützen. Dadurch wird ein leichterer Einstieg in die Arbeit mit KeY ermöglicht.

https://dx.doi.org/10.24355/dbbs.084-201801311013

https://dx.doi.org/10.24355/dbbs.084-201801311013

Inhaltsverzeichnis

Abbildungsverzeichnis vii

Tabellenverzeichnis ix

Quelltextverzeichnis xiii

Beschreibungsverzeichnis xvi

Hypothesenverzeichnis xviii

Abkürzungsverzeichnis xix

1 Einführung 1

2 Grundlagen 5
- 2.1 Java . . . 5
  - 2.1.1 Geschichtlicher Hintergrund . . . 6
  - 2.1.2 Das Verhalten von Ganzzahlen . . . 6
  - 2.1.3 Das Verhalten von Arrays . . . 6
  - 2.1.4 JavaDoc . . . 7
- 2.2 Das Liskovsche Substitutionsprinzip . . . 7
- 2.3 Deduktive Verifikation . . . 8
- 2.4 JML . . . 9
  - 2.4.1 Spezifikationsausdrücke . . . 9
  - 2.4.2 Spezifikation von Methoden . . . 12
- 2.5 KeY . . . 14

3 Erstellung von Verträgen für OpenJDK 17
- 3.1 Allgemeines Vorgehen . . . 17
- 3.2 Hilfsmethode Array.copyOf(Object[], int) . . . 20
- 3.3 Verträge für die Java Collections . . . 23
  - 3.3.1 Spezifikation von Modell-Feldern . . . 24
  - 3.3.2 Allgemeine Spezifikation der ArrayList . . . 25
  - 3.3.3 Spezifikation der Collection.size() . . . 28
  - 3.3.4 Spezifikation von ArrayList.ensureCapacity(int) . . . 29
  - 3.3.5 Spezifikation von Collection.add(Object) . . . 33
  - 3.3.6 Spezifikation von Constructor(int) der ArrayList . . . 36
- 3.4 Missverständnisse bei der Spezifizierung . . . 38
  - 3.4.1 Array Instanziierung - Nullable . . . 38

https://dx.doi.org/10.24355/dbbs.084-201801311013

3.4.2 Array Zugriffe - Datentypen 39
3.4.3 Das Signals-Keyword in mehreren Spezifikationsfällen 40
3.4.4 Invarianten nach Fehlerfällen in Konstruktoren 40
3.5 Zusammenfassung 41

4 Parameter und Optionen von KeY 43
4.1 Optionen zur Strategie der Beweissuche 44
4.2 Taclet Optionen 56
4.3 Allgemeine Optionen 66
4.4 Zusammenfassung 66

5 Empirische Evaluierung von Key-Optionen 71
5.1 Entwurf der Experimente 72
5.2 Entwicklung des KeY-Controllers 79
5.3 Auswertung der Experimente 79
5.4 Berechnung eines Parameter-Modells 84
5.5 Gültigkeit der Resultate 85
5.6 Zusammenfassung 92

6 Verwandte Arbeiten 95

7 Zusammenfassung 97

8 Ausblick 101

A Anhang 105
A.1 OpenJDK GPLv2 105
A.2 Weitere erstellte Verträge 112
A.2.1 Spezifikation von Collection.contains(Object) 112
A.2.2 Spezifikation von Collection.toArray() 112
A.2.3 Spezifikation von Collection.isEmpty() 113
A.2.4 Spezifikation von Collection.clear() 113
A.2.5 Spezifikation von List.indexOf(Object) 114
A.2.6 Spezifikation des Konstruktors der ArrayList() 114
A.2.7 Spezifikation von ArrayList.fastRemove(int) 115
A.2.8 Spezifikation von ArrayList.outOfBoundsMsg() 115
A.2.9 Spezifikation von ArrayList.rangeCheck() 115
A.2.10 Spezifikation von ArrayList.rangeCheckForAdd() 116
A.2.11 Spezifikation von ArrayList.trimToSize() 116
A.2.12 Spezifikation von Math.abs(int) 116
A.2.13 Spezifikation von Math.min(int, int) 117
A.2.14 Spezifikation von Math.max(int, int) 117
A.3 Weitere JavaDoc-Einträge 119
A.4 Feature-Modelle aus FeatureIDE 124
A.5 Speicherverbrauch von KeY 128
A.6 Modelle zum Beweisaufwand 129
A.7 Betrachtung der Qualifikation des Autors als Softwareingenieur 131

Literaturverzeichnis 133

https://dx.doi.org/10.24355/dbbs.084-201801311013

Abbildungsverzeichnis

2.1 Darstellung verschiedener Verifikationstechniken (eigene Darstellung) 8

2.2 Einfluss auf automatische Beweise in KeY (eigene, erweiterte Darstellung, basiert auf [ABB+16, S. 496]). 16

3.1 Ausschnitt aus dem Beweisbaum des Beweises des zweiten Spezifikationsfalls von ArrayList.size() 29

4.1 Darstellung der Optionen zur Strategie der Beweissuche als Feature Diagramm 45

4.2 Darstellung der Optionen zur Strategie der Beweissuche als Feature Diagramm (mit Einschränkungen) 57

4.3 Darstellung der Taclet Optionen als Feature Diagramm 58

4.4 Darstellung der Taclet Optionen als Feature Diagramm (mit Einschränkungen) 65

4.5 Hypothesen zur Beweisbarkeit (mit vermuteter Reihenfolge) 67

4.6 Hypothesen zum Beweisaufwand (mit vermuteter Reihenfolge) . . . 67

5.1 Boxplot der Differenzen in den Beweisaufwänden der *Stop At* Optionen (für offene Beweise) 81

5.2 Auswertung der Variablen 86

5.3 Abhängigkeiten ausgewählter Optionen zu Beweisbarkeit und Beweisaufwand 92

A.1 Feature-Modell mit arithmeticSematicsIgnoringOF (Taclet Optionen) 124

A.2 Feature-Modell mit arithmeticSematicsIgnoringOF (Search Strategy Optionen) 125

A.3 Feature-Modell mit javaSemantics (Taclet Optionen) 126

A.4 Feature-Modell mit javaSemantics (Search Strategy Optionen) 127

A.5 Darstellung des Speicherverbrauchs von KeY 128

A.6 Darstellung des Garbagecollectors bei der Ausführung von KeY . . . 128

https://dx.doi.org/10.24355/dbbs.084-201801311013

https://dx.doi.org/10.24355/dbbs.084-201801311013

Tabellenverzeichnis

4.1 Hypothesen mit Gleichheitsaussage 68

5.1 Darstellung der Experimente 78

5.2 Resultate der Experimente 82

5.3 Darstellung der manuellen Prüfungen 83

A.1 Darstellung der erzeugten Modelle 1 bis 5 129

A.2 Darstellung der erzeugten Modelle 6 bis 10 130

https://dx.doi.org/10.24355/dbbs.084-201801311013

https://dx.doi.org/10.24355/dbbs.084-201801311013

Quelltextverzeichnis

2.1 Ein Beispiel-JML-Vertrag . . . 10

2.2 Ein Beispiel Taclet: false_to_not_true (Quelle: KeY 2.6.1) . . . 15

3.1 Die Character-Klasse aus JavaRedux . . . 20

3.2 JavaDoc und Signatur zu Arrays.copyOf() . . . 22

3.3 Spezifikation zu Arrays.copyOf() . . . 22

3.4 Die definierten Modellfelder für die Java-Collections . . . 24

3.5 Die definierten Modellfelder der ArrayList . . . 25

3.6 Signaturen und JavaDocs der elementData- und size-Felder . . . 26

3.7 Signatur der ArrayList-Klasse . . . 26

3.8 Auszug des JavaDocs des modCount-Feldes der AbstractList-Klasse . 27

3.9 Die Invarianten und Signaturen der Felder der ArrayList . . . 27

3.10 JavaDoc und Signatur von Collection.size() . . . 28

3.11 Spezifikation von Collection.size() . . . 28

3.12 Implementierung von ArrayList.size() . . . 28

3.13 Angepasste Implementierung von ArrayList.size() . . . 29

3.14 JavaDoc und Signatur zu ArrayList.ensureCapacity(int) . . . 30

3.15 Formale Spezifikation von ArrayList.ensureCapacity(int) . . . 30

3.16 Implementierung von ArrayList.ensureCapacity(int) . . . 30

3.17 Der ursprünglich erstellte Vertrag von ArrayList.ensureCapacity(int) 31

3.18 Verifizierbare Instanziierung eines Arrays mit beliebiger, positiver Länge im int-Bereich . . . 33

3.19 JavaDoc und Signatur von Collection.add(Object) . . . 34

3.20 Formale Spezifikation von Collection.add(Object) . . . 35

3.21 Quelltext der JavaList.add(Object) . . . 36

3.22 JavaDoc und Signatur von ArrayList(int) . . . 37

https://dx.doi.org/10.24355/dbbs.084-201801311013

3.23 Formale Spezifikation von ArrayList(int) 37
3.24 Implementierung von ArrayList(int) 38
3.25 Angepasste Implementierung von ArrayList(int) 38
3.26 Minimalbeispiel zur Array-Instanziierung 39
3.27 Minimalbeispiel zum Datentypen bei Array-Zugriffen 40
3.28 Minimalbeispiel zur Verwendung zweier, nicht disjunkter, Exceptional Behavior-Spezifikationsfälle 40
3.29 Konstruktor, der trotz Exception die Referenz speichert 41
4.1 Beispiel für eine mit Query Treatment Off verifizierbare Methode . . 52
A.1 Informelle Spezifikation von Collection.add(Object) 112
A.2 Formale Spezifikation von Collection.add(Object) 112
A.3 Informelle Spezifikation von Collection.toArray() 112
A.4 Formale Spezifikation von Collection.toArray() 113
A.5 Informelle Spezifikation von Collection.isEmpty() 113
A.6 Formale Spezifikation von Collection.toArray() 113
A.7 Informelle Spezifikation von Collection.clear() 113
A.8 Formale Spezifikation von Collection.clear() 113
A.9 Informelle Spezifikation von List.indexOf(Object) 114
A.10 Formale Spezifikation von List.indexOf(Object) 114
A.11 Informelle Spezifikation von ArrayList() 114
A.12 Formale Spezifikation von ArrayList() 115
A.13 Informelle Spezifikation von ArrayList.fastRemove(int) 115
A.14 Formale Spezifikation von ArrayList.fastRemove(int) 115
A.15 Informelle Spezifikation von ArrayList.outOfBoundsMsg() 115
A.16 Formale Spezifikation von ArrayList.outOfBoundsMsg() 115
A.17 Informelle Spezifikation von ArrayList.rangeCheck() 115
A.18 Formale Spezifikation von ArrayList.rangeCheck() 116
A.19 Informelle Spezifikation von ArrayList.rangeCheckForAdd() 116
A.20 Formale Spezifikation von ArrayList.rangeCheckForAdd() 116
A.21 Informelle Spezifikation von ArrayList.trimToSize() 116
A.22 Formale Spezifikation von ArrayList.trimToSize() 116
A.23 Informelle Spezifikation von Math.abs(int) 116

https://dx.doi.org/10.24355/dbbs.084-201801311013

A.24 Formale Spezifikation und Implementierung von Math.abs(int) 117
A.25 Informelle Spezifikation von Math.min(int, int) 117
A.26 Formale Spezifikation von Math.min(int, int) 117
A.27 Informelle Spezifikation von Math.max(int, int) 117
A.28 Formale Spezifikation von Math.max(int, int) 118
A.29 JavaDoc zur List-Klasse 119
A.30 JavaDoc zur ArrayList-Klasse 121
A.31 JavaDoc des modCount-Felds der AbstractList-Klasse 123

https://dx.doi.org/10.24355/dbbs.084-201801311013

https://dx.doi.org/10.24355/dbbs.084-201801311013

Beschreibungsverzeichnis

2.1 Model und represents . . . 9
2.2 Quantifizierer . . . 10
2.3 \old . . . 10
2.4 \locSet / Dynamic Frames . . . 10
2.5 \fresh . . . 11
2.6 \typeof und \type . . . 11
2.7 Sichtbarkeit . . . 11
2.8 Pure . . . 11
2.9 Nullable . . . 11
2.10 Invariant . . . 11
2.11 Schleifeninvarianten . . . 12
2.12 Schleifenvariantenfunktion . . . 12
2.13 Spezifikationsfälle . . . 12
2.14 Requires . . . 13
2.15 Ensures . . . 13
2.16 Assignable . . . 13
2.17 Signals und Signals_only . . . 13
2.18 also . . . 14

4.1 Max. Rules Applications . . . 44
4.2 Stop At . . . 44
4.3 One Step Simplification . . . 46
4.4 Proof Splitting . . . 46
4.5 Loop Treatment . . . 47
4.6 Block Treatment . . . 48

https://dx.doi.org/10.24355/dbbs.084-201801311013

4.7 Method Treatment . . . 49

4.8 Merge Point Statements . . . 49

4.9 Dependency Contracts . . . 50

4.10 Query Treatment . . . 50

4.11 Expand Local Queries . . . 52

4.12 Arithmetic Treatment . . . 53

4.13 Quantifier Treatment . . . 53

4.14 Class Axiom Rule . . . 54

4.15 Auto Induction . . . 55

4.16 JavaCard . . . 56

4.17 Strings . . . 56

4.18 Assertions . . . 59

4.19 Bigint . . . 59

4.20 Initialisation . . . 60

4.21 IntRules . . . 60

4.22 IntegerSimplificationRules . . . 61

4.23 ModelFields . . . 61

4.24 Sequences . . . 61

4.25 MoreSeqRules . . . 62

4.26 ProgramRules . . . 62

4.27 Reach . . . 62

4.28 RuntimeExceptions . . . 63

4.29 WdChecks . . . 63

4.30 WdOperator . . . 64

4.31 MergeGeneratesIsWeakeningGoal . . . 64

4.32 Permissions . . . 64

4.33 Minimize Interaction . . . 66

4.34 SMT-Solver-Options . . . 66

https://dx.doi.org/10.24355/dbbs.084-201801311013

Hypothesenverzeichnis

1 Stop At - Beweisbarkeit . . . 46
2 Stop At - Beweisaufwand . . . 46
3 One Step Simplification (Beweisbarkeit) . . . 46
4 One Step Simplification (Beweisaufwand) . . . 46
5 Proof Splitting (Beweisbarkeit) . . . 47
6 Proof Splitting (Beweisaufwand) . . . 47
7 Proof Splitting (Beweisbarkeit Off) . . . 47
8 Proof Splitting (Beweisaufwand Off) . . . 47
9 Loop Treatment (Beweisbarkeit) . . . 48
10 Loop Treatment (Beweisaufwand) . . . 48
11 Dependency Contracts (Beweisbarkeit ohne Accessible-Klauseln) . . . 50
12 Dependency Contracts (Beweisaufwand ohne Accessible-Klauseln) . . . 50
13 Query Treatment (Beweisbarkeit; ohne Query) . . . 51
14 Query Treatment (Beweisaufwand; ohne Query) . . . 51
15 Query Treatment (Beweisbarkeit) . . . 51
16 Query Treatment (Beweisaufwand) . . . 51
17 Expand local queries (Beweisaufwand) . . . 52
18 Expand Local Queries (Beweisbarkeit) . . . 52
19 Arithmetic Treatment (Basic und DefOps) . . . 53
20 Arithmetic Treatment (Model Search) . . . 53
21 Quantifier Treatment ohne Quantifizierungen (Beweisbarkeit) . . . 54
22 Quantifier Treatment ohne Quantifizierungen (Beweisaufwand) . . . 54
23 Quantifier Treatment (Beweisbarkeit) . . . 54
24 Quantifier Treatment (Beweisaufwand) . . . 54

https://dx.doi.org/10.24355/dbbs.084-201801311013

25 Class Axiom Rule (Beweisbarkeit, ohne Axiome) 54
26 Class Axiom Rule (Beweisaufwand, ohne Axiome) 55
27 Class Axiom Rule (mit Axiomen und Schreibzugriffen) 55
28 Class Axiom Rule (Axiome nicht benötigt) 55
29 Strings (Beweisbarkeit) 56
30 Strings (Beweisaufwand) 59
31 Bigint (Beweisbarkeit) 59
32 Bigint (Beweisaufwand) 60
33 IntegerSimplificationRules (Beweisbarkeit) 61
34 IntegerSimplificationRules (Beweisaufwand) 61
35 Sequences (Beweisbarkeit) 62
36 Sequences (Beweisaufwand) 62
37 MoreSeqRules (Beweisbarkeit) 62
38 MoreSeqRules (Beweisaufwand) 62

https://dx.doi.org/10.24355/dbbs.084-201801311013

Abkürzungsverzeichnis

API	Application Programming Interface
BISL	Behavioral Interface Specification Language
JDK	Java Development Kit
JML	Java Modeling Language

https://dx.doi.org/10.24355/dbbs.084-201801311013

https://dx.doi.org/10.24355/dbbs.084-201801311013

1. Einführung

Die Qualität von Software hängt laut ISO 25010 direkt von ihrer Zuverlässigkeit ab [ISO10]. Um diese zu gewährleisten, können verschiedene Techniken genutzt werden. Die meisten Softwaretechniker verwenden heuristische, informelle Methoden zum Überprüfen von Software. Als Alternative steht jedoch auch die formale Verifikation von Software zur Verfügung. Eine Technik hierzu ist deduktive Verifikation, welche mit Hilfe von Tools wie KeY durchgeführt werden kann. Hierbei wird statisch die Ausführung zur Laufzeit nachgebildet und gegen die in Annotationen angegebenen Spezifikationen geprüft [CKLP06, S. 343]. KeY verwendet hierbei die Java Modeling Language (JML)[LBR06] für die Spezifikation der Programme [ABB+16, S. 2ff], welche die Möglichkeit bietet, Java Code zu annotieren. Bei JML handelt es sich um eine Behavioral Interface Specification Language (BISL). Hierbei wird als Interface eine Methode, ein Feld oder ein Typ aufgefasst. Für diese kann mit JML das Verhalten spezifiziert werden [LPC+13, S1.f].

Zielstellung der Arbeit

Das Ziel des KeY-Projekts ist es, formale Software-Analyse-Methoden für die Software Entwicklung zugänglicher zu machen [ABB+16, S. 1]. KeY unterstützt hierfür Java, macht jedoch Einschnitte bei den unterstützten Funktionen [ABB+16, S. 3]. Das Ziel dieser Arbeit ist es, mittels Stichproben zu ermitteln, wie leicht Software Entwickler mit KeY Java-Quellcode verifizieren können. Als Grundlage wird hierfür der Quellcode der Java Platform Application Programming Interfaces (APIs) [GJSB05, S. 7][Ora15] verwendet, welcher mit dem Java Development Kit (JDK) mitgeliefert wird. Diese Stichproben werden mit Hilfe der informellen Spezifikationen, welche in Form von Javadoc vorliegen, um formale Spezifikationen in JML ergänzt. Sollte KeY den Quelltext in der derzeitigen Fassung nicht unterstützen, wird er entweder so angepasst, dass er verifiziert werden kann oder nicht weiter betrachtet.

Als Erkenntnis sollen dabei aktuelle Probleme bei der Arbeit mit KeY aufgedeckt und gegebenenfalls Ausblicke für mögliche Lösungen erarbeitetet werden. Diese Arbeit betrachtet allerdings nur den derzeit verfügbaren Funktionsumfang von KeY.

https://dx.doi.org/10.24355/dbbs.084-201801311013

Wichtig ist hierbei für diese Arbeit, dass mit dem Wissen eines in die Materie eingearbeiteten Softwareentwicklers und nicht mit dem eines KeY-Experten an die Probleme herangegangen wird, um beim Erreichen des vom KeY-Team selbst formulierten Ziels zu unterstützen: dem „Einbringen von formalen Software-Analyse-Methoden [...] in den Bereich der mainstream Softwareentwicklung" [ABB+16, S. 1]. Aus diesem Grund ist ein weiteres Ziel der Arbeit, grundsätzliche Tipps für die Herangehensweise an alltägliche Quellcode-Beispiele zu geben, weshalb der Quelltext der Java Platform API genutzt wird. Hierzu sollen auch die Parameter von KeY betrachtet werden, da diese einen entscheidenden Einfluss auf die Beweisbarkeit und den Aufwand einer Verifikation nehmen können. Zu diesem Zweck sollen die Parameter und Optionen zunächst anhand vorliegender Beschreibungen vorgestellt und anschließend diskutiert werden. Das Ziel dieses Abschnittes soll es sein, Informationen zu den Parametern zusammenzutragen und Empfehlungen für potenzielle Anpassungen von KeY zu geben. Ein Nebenergebnis der Arbeit sind auch Erkenntnisse darüber, inwiefern die Java Platform API die informelle Spezifikation erfüllt. Desweiteren soll untersucht werden, inwiefern man existierenden Quelltext möglichst ohne dessen Veränderung verifizieren kann. Dies steht im Widerspruch zum Prinzip des Design by Contract, oder auch Programming by Contract, bei welchem Entwickler während des Designs oder spätestens während der Implementierung bereits Kontrakte schreiben (vgl. [Mey88, S. 111ff]).

Die Arbeit wird dabei bewusst von der Verifikation einzelner, komplexer Probleme (vgl. [dGRdB+15]) abgegrenzt, um ein breiteres Spektrum an Problemen betrachten zu können. Aus diesem Grund wird auch von der Verifikation von ausschießlich zusammenhängenden Sinneinheiten (vgl. [SSZ+15]) abgesehen. Das OpenJML-Projekt bietet bereits viele in JML geschriebene Spezifikationen [Cok11]. Da es sich bei dieser Arbeit um einen Erfahrungsbericht handeln soll, der den Prozess bei der formalen Verifikation begleiten soll, wird von der Verwendung dieser bestehenden Spezifikationen abgesehen. Andere Arbeiten (vgl. [KMS+11]) beschäftigen sich ebenfalls mit einem Erfahrungsbericht zur Verifikation von Software, gehen jedoch mehr auf den Vergleich verschiedener Tools und Techniken ein. Diese Arbeit legt im Gegensatz dazu einen Fokus auf die deduktive Verifikation mit Hilfe von KeY, um eine tiefergehende Diskussion zu ermöglichen und damit Anwendern einen größeren Mehrwert zur Verfügung stellen zu können.

Gliederung der Arbeit

Die Arbeit soll zunächst auf die verschiedenen Versionen von Java eingehen, um eine Grundlage für die Auswahl der Java-Version zu bieten, die als Grundlage für die Fallstudie verwendet wird (siehe Abschnitt 2.1). An dieser Stelle werden auch einzelne Java-Sprachelemente mit Hilfe der Java-Sprachspezifikation [GJSB05] und der Java-Virtual-Machine-Spezifikation [LYBB15] behandelt. Anschließend sollen die grobe Funktionsweise von KeY im Allgemeinen und hierfür notwendige Grundlagen wie JML oder das liskovsche Substitutionsprinzip kurz erläutert werden (siehe Abschnitt 2.2 bis Abschnitt 2.5). Dies stellt keine umfassende Anleitung für KeY dar (vgl. KeY-Buch [ABB+16]), sondern soll lediglich für spätere Entscheidungen eine Grundlage bieten.

Nach den Grundlagen werden in Kapitel 3 die einzelnen Spezifikationen und Beweise mit den jeweils aufgetretenen Problemen vorgestellt. Dazu wird zunächst die

https://dx.doi.org/10.24355/dbbs.084-201801311013

Java-Version gewählt, um dann auf die nötigen Anpassungen einzugehen (siehe Abschnitt 3.1). Anschließend soll der Weg zur letztendlich erstellten Spezifikation aufgezeigt werden, um späteren Anwendern mit ähnlichen Problemen eine Hilfsstellung zu geben. Dabei wird zunächst auf die erstellten Verträge eingegangen, um anschließend gegebenenfalls auf gefundene Probleme einzugehen. Sollte ein Beweis nicht automatisch geschlossen werden können, sollen an dieser Stelle die Ursachen ergründet werden. Die im Rahmen der Spezifikation aufgefallenen Probleme und Missverständnisse werden abschließend noch einmal separat in Abschnitt 3.4 dargestellt.

Nach der Spezifikation soll auf die Arbeit mit KeY an sich eingegangen werden. Da die Parameter und Optionen in KeY eine entscheidende Rolle spielen, werden diese in Kapitel 4 vorgestellt. Bei den Parametern wird sowohl die Auswirkung auf die automatische Beweisschließung durch KeY, als auch auf den Aufwand der Beweise mit den jeweiligen Optionen eingegangen. Hierbei wird zwischen Optionen zur Strategie der Beweissuche (siehe Abschnitt 4.1), Taclet-Optionen (siehe Abschnitt 4.2) und weiteren Optionen (siehe Abschnitt 4.3) unterschieden. Ziel dieser Abschnitte ist es, zukünftige Anwender bei der Wahl der richtigen Parameter zu unterstützen. Von diesem Abschnitt sollen vor allem KeY-Anwender profitieren, da sie eventuell bei ihrer Arbeit auf ähnliche Probleme stoßen. Um dies zu unterstützen, werden aus den Beschreibungen der Parameter abstrahierende Forschungshypothesen abgeleitet. Da die Parameter und ihre Beschreibungen zwar alle auf KeY zugeschnitten sind, jedoch eventuell in ähnlicher Form auch in anderen deduktiven Verifikationswerkzeugen vorkommen, könnten auch Anwender dieser anderen Werkzeuge davon profitieren.

Im Anschluss an die Beschreibungen der Parameter und Optionen von KeY sollen in Kapitel 5 die Hypothesen untersucht werden. Hierzu werden zunächst Experimente entworfen (siehe Abschnitt 5.1), um sie anschließend auszuwerten und Rückschlüsse aus ihnen zu ziehen (siehe Abschnitt 5.3). Da die Anzahl der nötigen Experimente für die Untersuchung der Hypothesen zu groß ist, als dass sie im Rahmen der Arbeit manuell ausgeführt werden könnten, muss ein Werkzeug entwickelt werden, dass die Ausführung übernimmt. Dieses Werkzeug wird kurz in Abschnitt 5.2 vorgestellt. Die im Rahmen der Experimente gewonnen Daten sollen anschließend in Abschnitt 5.4 noch einmal mit Hilfe von *SPLConqueror* analysiert werden, um ein Modell zu erhalten, dass den Beweisaufwand in Abhängigkeit der Optionen von KeY zu beschreiben. Um die so gewonnen Informationen und Ergebnisse noch einmal kritisch zu betrachten, wird in Abschnitt 5.5 eine Diskussion der Gefahren für die Gültigkeit der Resultate durchgeführt.

Mit Hilfe der im Rahmen der Arbeit ermittelten Hypothesen und gefundenen Probleme und Lösungen sollen Vorschläge erarbeitet werden, wie die Arbeit mit KeY erleichtert werden kann (siehe Abschnitt 3.5 und Abschnitt 5.6). Die Adressaten dieses Abschnitts sind vor allem die Entwickler von KeY, da die Vorschläge zur Verbesserung genutzt werden können. Abschließend sollen verwandte Arbeiten, ein Fazit und ein Ausblick auf zukünftige Arbeiten gegeben werden (siehe Kapitel 6 bis Kapitel 8).

https://dx.doi.org/10.24355/dbbs.084-201801311013

https://dx.doi.org/10.24355/dbbs.084-201801311013

2. Grundlagen

Im weiteren Verlauf dieser Arbeit soll die deduktive Verifikation mit KeY betrachtet werden. Um auf diese eingehen zu können, soll in diesem Kapitel zunächst ein Grundlagenwissen erarbeitet werden. Mit Hilfe dieser Grundlagen sollen im Anschluss Verträge für OpenJDK erstellt und verifiziert werden.

Da es sich bei OpenJDK um eine Java-Distribution handelt, sollen zunächst einzelne Details zu Java vorgestellt werden (Abschnitt 2.1). Hierbei wird zunächst kurz auf die Geschichte von Java eingegangen, um anschließend einzelne, für diese Arbeit wichtige Sprachkonstrukte vorzustellen. Anschließend wird das Liskovsche Substitutionsprinzip vorgestellt (Abschnitt 2.2), da dies eine allgemeine Aussage über das Verhalten von Subtypen trifft, welches im weiteren Verlauf der Arbeit für die Erstellung der Verträge genutzt wird. Danach werden zunächst deduktive Verifikation und die dazu gehörenden Begrifflichkeiten eingeführt (Abschnitt 2.3), um daraufhin JML und dessen Schlüsselworte vorzustellen (Abschnitt 2.4). Nachdem diese Grundlagen zur Spezifikation gelegt wurden, wird abschließend noch KeY vorgestellt, welches für die Verifikation genutzt wird (Abschnitt 2.5).

2.1 Java

Um Aussagen über Unterschiede zwischen verschiedenen Java-Versionen zu treffen, wird zunächst die Geschichte von Java kurz vorgestellt (Abschnitt 2.1.1). Dabei ist nicht das Auftreten einzelner APIs interessant, sondern der Sprachumfang. Diese Sprachfeatures werden in Abschnitt 3.1 verwendet, um die Version auszuwählen, die als Basis dienen soll. Anschließend sollen mehrere verschiedene Sprachfeatures vorgestellt werden, welche im weiteren Verlauf der Arbeit für das Aufstellen der Verträge und die Verifikation wichtig sind. Hierzu wird das Verhalten von Ganzzahlen (Abschnitt 2.1.2) und das Verhalten von Arrays dargestellt(Abschnitt 2.1.3). Abschließend wird noch JavaDoc vorgestellt (Abschnitt 2.1.4), da dieses im weiteren Verlauf der Arbeit als Ursprung für die informellen Spezifikationen zu dem Java-Quelltext dienen.

https://dx.doi.org/10.24355/dbbs.084-201801311013

2.1.1 Geschichtlicher Hintergrund

Die Programmiersprache Java existiert seit 1995 [SRK98, S.5]. Es stehen verschiedene Java-Technologien zur Verfügung: Java SE, Java EE, Java ME, Java Embedded und Java Card [Oraf]. Der Fokus dieser Arbeit soll auf Java SE, der Standard Edition von Java [Orad], liegen. Der Vollständigkeit halber sollen die anderen Versionen jedoch kurz vorgestellt werden. Java EE bietet eine Plattform, die Hilfsmittel für die Entwicklung von Unternehmensanwendungen zur Verüfugung stellt [DS13, S.1f] und basiert auf Java SE-Technologie [DS13, S. 6]. Java ME setzt den Fokus auf Embedded-Systeme und mobile Endgeräte und liegt auch als spezielle Version für Embedded-Systeme vor [Orac]. Auch von Java SE liegt eine solche, speziell auf Embedded-Systeme zugeschnittene Fassung vor [Orae], diese ist jedoch nicht Bestandteil dieser Arbeit. Abschließend existiert auch noch Java Card, welches es ermöglichen soll, in Java geschriebene Programme auf Smart Cards auszuführen. Java Card bietet dabei allerdings nicht alle Java-Sprachfeatures, wie zum Beispiel den primitven Datentyp `long` oder einen *Garbage Collector* [Che00, S.30f].

Java SE wurde seit 1995 mehrfach überarbeitet. So wurden Java 2002 mit Version 1.4 beispielsweise mehrere APIs und das Zusicherungen-Feature hinzugefügt [j14]. 2004 gab es mit Java 5.0 die nächste Iteration, welche erneut mehrere API und ebenfalls mehrere Sprachfeatures hinzufügte [j15]. Diese umfassen Enumerationen, das Einbinden von statischen Feldern und Methoden, *for-each*-Schleifen und das automatische Packen und Entpacken von primitiven Datentypen (*Autoboxing*). Zusätzlich wurde auch ein System für das Definieren von Metadaten mit Annotationen und generische Datentypen in Java ergänzt. Im Rahmen des 2006 erschienenen Java SE 6 wurde ein Plugin-System für das mit 5.0 eingeführte Metadaten-System eingeführt und eine Vielzahl neuer APIs und Technologien hinzugefügt [j16]. Ein ähnliches Bild bietet sich auch für die mit Java SE 7 2011 eingeführten Änderungen [j17]. Jedoch wurden auch neue Annotationen erstellt und im Rahmen des „OpenJDK Project Coin“ auch ein neues Sprachfeature hinzugefügt: eine Verwaltung von Ressourcen in *try-catch*-Blöcken [jPr]. Java SE 8 führte 2014 neben erneuten APIs und Annotationen auch die neuen Lambda-Ausdrücke ein [j18].

2.1.2 Das Verhalten von Ganzzahlen

Java kennt insgesamt fünf primitive Ganzzahltypen: `byte`, `char`, `short`, `int` und `long` [GJSB05, S. 35f]. Diese unterscheiden sich durch ihren Wertebereich voneinander. Alle bis auf `char` sind vorzeichenbehaftet. Die vorzeichenbehafteten Datentypen können Werte von -2^k bis $2^k - 1$ enthalten, wobei k für `byte` 7 entspricht, für `short` 15, für `int` 31 und für `long` 63. Ein `char` kann Werte zwischen 0 und 65535 enthalten. Für die vorzeichenbehafteten Werte wird das Zweierkomplement zur internen Darstellung verwendet. Bei der Addition zweier Ganzzahlen kann es dabei zu einem Überlauf kommen [GJSB05, S. 500f]. Eine Subtraktion kann in diesem Fall auch als Addition dargestellt werden, bei der der Subtrahend als negierter Summand verwendet wird, wodurch es auch hier zu einem Überlauf kommen kann.

2.1.3 Das Verhalten von Arrays

Bei einem Array handelt es sich um ein Objekt, welches eine beliebige, aber feste Anzahl an Variablen enthalten kann. Bei einem Array-Typen handelt es sich um

https://dx.doi.org/10.24355/dbbs.084-201801311013

einen Referenztypen und nicht um einen primitiven Datentypen [GJSB05, S. 44]. Diese Anzahl an Variablen wird als Länge bezeichnet und kann 0 sein. Der Zugriff auf die Variablen erfolgt hierbei über eine Ganzzahl zwischen 0 und $n-1$, wobei n die Länge des Arrays ist [GJSB05, S. 287]. Die maximale Länge eines Arrays ist dabei nicht in der Sprachspezifikation festgelegt, es lässt sich jedoch herausfinden, dass die *Hotspot VM* in der 64-Bit Version „build 25.31-b07“ als maximale Array-Größe $2^{31} - 4$ (`Integer.MAX_VALUE -3`) unterstützt, eine solche Grenze konnte auch bei anderen virtuellen Maschinen beobachtet werden. Beim Versuch ein größeres Array zu instanzieren wird eine `java.lang.OutOfMemoryError` mit dem Grund „Requested array size exceeds VM limit“ geworfen. Dieses Verhalten konnte auch in anderen Versionen der *Java Runtime Environment* nachgewiesen werden. Der Fehler tritt also ab einem Wert von `Integer.MAX_VALUE−2` auf. Eine mögliche Erklärung für diese Grenze wäre der in der *Hotspot VM* für Arrays verwendete Drei-Maschinenwort-Kopf [Orab]. Um dies zu überprüfen, wird die Implementierung in der *Hotspot*-Implementierung des OpenJDK betrachtet. In der Implementierung der Allokierung des Arrays fällt auf, dass eine entsprechende Meldung geworfen wird, wenn die Länge eine Grenze übersteigt [opeb, src/share/vm/oops/arrayKlass.cpp - arrayKlass::allocate_arrayArray - Zeile 142]. Die Grenze wird mit einer anderen Methode berechnet, der als Parameter `T_ARRAY` übergeben wird. Die maximale Anzahl der Worte wird dort explizit aus `max_jint−header_size(type)−2` berechnet [opeb, src/share/vm/oops/arrayOop.hpp - max_array_length - Zeile 108]. Dies unterstützt diese These ebenfalls, ist jedoch implementierungsabhängig und kann sich deshalb in späteren Versionen von Java ändern.

2.1.4 JavaDoc

Bei JavaDoc handelt es sich um ein Tool, welches API-Dokumentationen aus Kommentaren im Quelltext generiert [javb]. Ein JavaDoc-Eintrag muss immer mit `/**` beginnen [java] und hebt sich damit von einem normalen mehrzeiligen Kommentar ab, der mit `/*` beginnt [GJSB05, S. 18]. Diese Kommentare werden in der Java Platform API genutzt, um die Schnittstelle möglichst gut zu dokumentieren [java]. Die Zielsetzung an dieser Stelle ist, dass das Verhalten durch die JavaDoc-Einträge oder durch Dokumente definiert werden, auf die in diesen verwiesen wird. Die so angegebenen Spezifikationen sollen hierbei nicht nur als Dokumentation, sondern eher als Vertrag angesehen werden, auf den sich die rufenden Methoden verlassen können. Sollte es zu einem Fehler kommen, kann es sich hierbei sowohl um einen Spezifikations- als auch um einen Implementierungsfehler handeln. Diese werden jedoch für gewöhnlich nicht in der Spezifikation aufgelistet.

2.2 Das Liskovsche Substitutionsprinzip

Das Liskovsche Substitutionsprinzip beschreibt das Verhalten eines Subtypen zu seinem Supertypen [LW94, S.1811ff]: „Sei $\phi(x)$ eine beweisbare Eigenschaft eines Objekts X vom Typen T. Dann sollte $\phi(y)$ wahr sein für ein Objekt y vom Typen S, wobei S ein Subtyp von T ist“ [LW94, S. 1812]. Laut diesem Prinzip muss sich der Subtyp unter der Sichtbarkeit des Supertypen wie auch der Supertyp verhalten. Diese Annahme führt dazu, dass an jeder Stelle, an der der Supertyp genutzt werden kann, auch der Subtyp eingesetzt werden kann. Da Java eine Zuweisung eines Subtypen

https://dx.doi.org/10.24355/dbbs.084-201801311013

auf das Feld eines Supertypen gestattet [GJSB05, S. 93ff], kann nicht immer mit Gewissheit gesagt werden, welchen Typ ein Objekt haben wird. In einer solchen Situation kann ohne eine solche Forderung ein Fehlverhalten auftreten.

2.3 Deduktive Verifikation

Um die Zuverlässigkeit von Software zu prüfen, stehen verschiedene Techniken zur Verfügung (siehe Abbildung 2.1). Bei bekannten Techniken wie Unittests und Code Reviews handelt es sich um informelle Vorgehensweisen [ABB+16, S. 2]. Formale Techniken umfassen beispielsweise *Assertions*, also Zusicherungen im Quelltext [Wei11, S. 3]. Dies geschieht zur Laufzeit und ist deshalb eine dynamische Technik. Eine Technik, um Software statisch, formal zu verifizieren ist deduktive Verifikation.

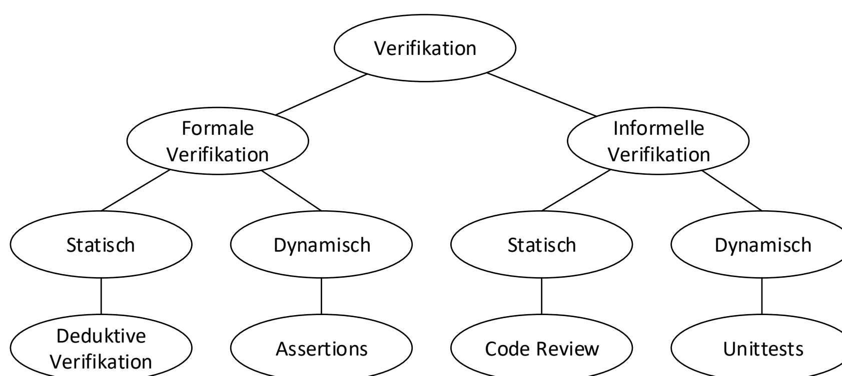

Abbildung 2.1: Darstellung verschiedener Verifikationstechniken (eigene Darstellung)

Deduktive Verifikation wandelt hierfür ein Programm in mathematische Formeln um, um diese zu beweisen [Fil11, S. 397ff]. Als Darstellung wird immer eine Logik verwendet, welche genau ist jedoch vom Verifikationswerkzeug abhängig. Hierbei kommen beispielsweise Prädikatenlogik, die Sprachen allgemeiner Beweisengines oder explizit für deduktive Verifikationen entworfene Logiken zum Einsatz. Grundsätzlich muss hierbei aus jedem Zustand, der die Vorbedingung erfüllt, unter Anwendung der zu verifizierenden Funktion immer ein Zustand folgen, der die Nachbedingung erfüllt. Hierfür wird ein Kalkül definiert, welches die Regeln enthält, mit deren Hilfe die Schlussfolgerungen auf der mathematische Formel getroffen werden [Wei11, S. 89]. Damit ein Kalkül angewendet werden kann, muss die zu verifizierende Funktion zunächst in einen Term der jeweiligen Logik umgewandelt werden [Fil11, S. 399].

Ein solches Kalkül kann vollständig und korrekt sein. Vollständigkeit, im englischen *completeness*, beschreibt, dass jede logisch wahre Aussage, die sich schlussfolgern lassen kann, auch gezeigt werden kann [Wei11, S. 89f]. Dies ist allerdings für Logiksprachen, wie zum Beispiel die in Key verwendete Sprache JavaDL, nicht möglich. Von daher wird von relativer Vollständigkeit gesprochen, wenn jede im Rahmen der

https://dx.doi.org/10.24355/dbbs.084-201801311013

Sprache beweisbare Formel bewiesen werden kann. Ein Kalkül ist genau dann korrekt, oder auch *sound*, wenn alle Regeln korrekt sind. Damit eine Regel korrekt ist, darf aus gültigen Vorbedingungen nur etwas Gültiges gefolgert werden.

Damit eine mit Hilfe eines solchen Werkzeugs verifizierte Funktion auch wirklich korrekt ist, muss auch das Kalkül korrekt sein. Wenn dies nicht der Fall ist, können auch falsche Aussagen geschlussfolgert werden. Eine möglichst hohe Vollständigkeit ist für die Aussagefähigkeit eines solchen Werkzeugs wichtig.

Ein Werkzeug zur deduktiven Verifikation ist KeY, welches Java unterstützt. Dieses wird im weiteren Verlauf dieser Arbeit behandelt. Eine alternative für C# auf der .Net-Plattform stellt beispielsweise Spec# zur Verfügung, welches ebenfalls Tools zur statischen, formalen Verifikation umfasst [BLS05, S. 53]. Unter anderem das Werkzeug *Boogie*, welches ebenfalls deduktive Verifikation nutzt [BCD+05, S. 364f].

2.4 JML

JML ist eine Behavioral Interface Specification Language [LBR06, S. 1]. Bei einer *behavioral specification* handelt es sich um eine Beschreibung des beabsichtigten Verhaltens einer Komponente oder eines Systems [HLL+12, S. 16:1]. Die formale Version einer *behavioral specification* wurde ursprünglich *interface specification* genannt. Um Verwechslungen mit dem Interface von Modulen zu vermeiden, wurden sie jedoch in *behavioral interface specification* umgetauft [HLL+12, S. 16:4]. Bei JML handelt es sich also um eine Sprache, mit deren Hilfe das Verhalten von Objekten formal beschrieben werden kann. JML versteht unter dem Verhalten einer Methode eine Menge von Veränderungen. Hierzu wird definiert, unter welchen Bedingungen ein bestimmtes Verhalten auftreten soll, was nach der Ausführung gilt, und ob Fehlerverhalten, also `Exceptions`, auftreten können. Zusätzlich lässt sich auch noch beschreiben, welche Felder ein Verhalten verändern kann [LPC+13, S. 2]. Eine Methode kann in diesem Fall als Verhalten verstanden werden.

Diese formale Beschreibung wird entweder in derselben Datei wie der Quelltext, oder einer separaten Datei hinterlegt. Ein Beispiel für eine solche, formale Spezifikation einer Methode ist in Quelltext 2.1 gegeben. Um weiterhin das Kompilieren dieser Dateien zu ermöglichen, wird die Spezifikation in einem Java-Kommentar hinterlegt [LBR06, S. 1]. Da Java-Kommentare ein- oder mehrzeilig definiert werden können [GJSB05, S. 18], gilt dies auch für JML-Spezifikationen. Für die genaue Beschreibung des Sprachumfangs von JML wird das Referenzhandbuch empfohlen [LPC+13]. An dieser Stelle sollen jedoch einige im weiteren Verlauf verwendete Sprachbestandteile erläutert werden.

2.4.1 Spezifikationsausdrücke

Beschreibung 2.1 – Model und represents: Das Schlüsselwort *model* kann in verschiedenen Kontexten eingesetzt werden. Es gibt Modell-Typen, Modellfelder und Modell-Methoden [LPC+13, S. 11, 39, 46]. In jedem dieser Fälle wird der jeweilige Kontext nicht zum Quelltext hinzugefügt, steht aber in der Spezifikation zur Verfügung. Mit Hilfe von Modellfeldern können ein oder mehrere Felder in einem

https://dx.doi.org/10.24355/dbbs.084-201801311013

```
1 /*@ public normal_behavior
2   @ requires minCapacity <= elementData.length;
3   @ ensures modCount != \old(modCount);
4   @ ensures elementData.length >= minCapacity;
5   @ ensures (\forall int i; 0 <= i && i < size;
6     \old(elementData[i]) == elementData[i]);
7   @ assignable modCount;
8   @*/
```

Quelltext 2.1: Ein Beispiel-JML-Vertrag

abstrakten Feld zusammen dargestellt werden. Um das abstrakte Feld zu spezifizieren, wird das *represents*-Schlüsselwort genutzt. Es gibt zwei verschiedene Formen von *represents*: funktionale Abstraktion und relationale Abstraktion [LPC+13, S. 60]. Bei der funktionalen Abstraktion wird ein Wert über eine Referenz zugewiesen, bei der relationalen Abstraktion wird hingegen verlangt, dass ein selbst definierter logischer Ausdruck erfüllt werden muss.

Beschreibung 2.2 – Quantifizierer [LPC+13, S. 101f]**:** JML stellt 7 verschiedene Quantifizierer zur Verfügung. Der Allquantor (`\forall`) und der Existenzquantor (`\exists`) liefern einen *Boolean* zurück, der dem Ergebnis des jeweiligen prädikatenlogischen Ausdruck entspricht. Es gibt noch Quantoren, die eine Summe (`\sum`) oder ein Produkt (`\product`) bilden oder das Maximum (`\max`) oder Minimum (`\min`) ermitteln. Die Quantoren haben dabei stets den folgenden Aufbau: „(Schlüsselwort Datentyp Variablenname; Bedingung; Ausdruck)“. Die Bedingung ist hierbei allerdings optional. Sollte sie weggelassen werden, wird sie standardmäßig mit `true` belegt. Beispielsweise würde ein Allquantor über alle Felder eines Arrays a, welcher prüft, ob die Felder aufsteigend sortiert sind, wie folgt aussehen: „(`\forall int i; 0 <= i && i < a.length -1; a[i] <= a[i+1]`)“. Der Ausdruck des Quantors wird hierbei nur ausgewertet, wenn die Bedingung erfüllt ist. Der siebte Quantifizierer zählt die Anzahl der Variablen (`\num_of`), bei denen sowohl die Bedingung wie auch der Ausdruck erfüllt ist. Sowohl für diesen als auch den All- und den Existenzquantor muss der auszuwertende Ausdruck einen *Boolean* ergeben, für die anderen Quantoren eine Zahl.

Beschreibung 2.3 – \old [LPC+13, S. 93f]**:** Das `\old`-Schlüsselwort wird genutzt, um den Inhalt eines Feldes oder das Resultat einer Methode vor der Ausführung der jeweiligen Methode zu ermitteln. Die jeweilige Methode ist dabei die Methode, in dessen Vertrag der Ausdruck verwendet wird. `\old` liefert dabei bei einem Feld allerdings immer den Inhalt zurück. Wenn es sich dabei um eine Referenz handelt, wird die Referenz zurückgegeben. Das Objekt hinter dieser Referenz kann sich während der Ausführung geändert haben, was sich auch auf das Resultat des Ausdrucks auswirkt.

Beschreibung 2.4 – \locSet / Dynamic Frames [Wei11, S. 43ff]**:** Bei `\locset` handelt es sich um einen für Spezifikationen eingeführten primitiven Datentypen. Mit Hilfe dieses Datentypens ist es möglich, in Modellfeldern eine Menge an Speicher-

https://dx.doi.org/10.24355/dbbs.084-201801311013

orten zu speichern. Felder von diesem Datentyp werden auch als *dynamic frames* bezeichnet.

Beschreibung 2.5 – \fresh [LPC+13, S. 97]**:** `\fresh` sichert zu, dass die spezifizierten Felder vor der Ausführung der jeweiligen Methode nicht existierten, es danach allerdings tun. Sollte das Objekt vor der Ausführung bereits im *Heap* existieren oder `null` sein, so ist das Resultat `false`.

Beschreibung 2.6 – \typeof und \type [LPC+13, S. 99f]**:** Um in einer Spezifikation den Typen eines Objekts zu ermitteln, kann das `\typeof`-Schlüsselwort verwendet werden. Hierbei wird immer der präziseste, dynamisch ermittelbare Datentyp eines Ausdrucks ermittelt. `\type` kann verwendet werden, um aus einem Typnamen den Typen zu ermitteln, welcher mit dem Resultat von `\typeof` verglichen werden kann.

Beschreibung 2.7 – Sichtbarkeit [LPC+13, S. 64]**:** Die Sichtbarkeit einer Spezifikation kann bei schwergewichtigen Spezifikationen explizit festgelegt werden. Wie auch bei Java stehen `private`, `protected` und `public` zur Verfügung. Es darf dabei aber für eine `private`-Methode keine `public`-Spezifikation verwendet werden. Die Sichtbarkeit regelt, auf welche Spezifikationen zugegriffen werden kann. Hierbei kann nur auf Objekte mit höherer Sichtbarkeit zugegriffen werden, wie beispielsweise `private` auf `public`, aber `public` nicht auf `private`.

Beschreibung 2.8 – Pure [LPC+13, S. 46ff]**:** Sowohl eine Methode als auch eine Klasse oder ein Interface können `pure` sein. Eine Klasse oder ein Interface, das pure ist, enthält nur Methoden die `pure` sind. Das bedeutet, dass die Methode terminieren muss und keine Zuweisungen von im Vorfeld existierenden Feldern vornehmen darf (siehe Assignable in Abschnitt 2.4.2). Dies gilt auch für nicht spezifiziertes Verhalten. Die Ausnahme stellen hierbei Konstruktoren dar. Diese dürfen schreibend auf Felder des eigenen Objekts zugreifen.

Beschreibung 2.9 – Nullable [LPC+13, S. 16f]**:** Standardmäßig wird in JML angenommen, dass Objekte nicht `null` sein dürfen. Bei Arrays gilt dies nicht nur für das Objekt an sich, sondern auch für alle Felder des Arrays. Um dieses Verhalten zu ändern, kann vor dem Datentyp des Feldes oder des Parameters oder vor dem Rückgabetypen das Schlüsselwort `nullable` verwendet werden. In diesem Fall kann das Objekt `null` sein. Bei Arrays überträgt sich das Schlüsselwort auch auf alle Felder des Arrays.

Beschreibung 2.10 – Invariant [LPC+13, S. 52f]**:** Eine Invariante wird mit dem Schlüsselwort `invariant` angegeben. Nach diesem Schlüsselwort folgt ein boolscher Ausdruck, der zu jedem sichtbaren Zustand erfüllt sein muss. Ein Zustand ist dann sichtbar, wenn er eine von sechs verschiedenen Bedingungen erfüllt. Eine dieser möglichen Bedingungen ist, dass kein Konstruktor, Finalisierer und auch keine Methode oder statische Methode von dem jeweiligen Objekt oder seiner Klasse zu diesem Zeitpunkt aufgerufen wird. Die anderen Bedingungen beziehen sich alle auf

https://dx.doi.org/10.24355/dbbs.084-201801311013

Methoden, die nicht als `helper` markiert sind. Möglich sind hierbei die folgenden Zeitpunkte:

- Nach einem Konstruktor, der das Objekt mit der Invariante initialisiert.
- Vor einem Destruktor, der vor der Löschung des Objekts aufgerufen werden soll.
- Vor dem Aufruf einer Methode des Objekts.
- Nach dem Aufruf einer Methode des Objekts.

Es lässt sich mit Hilfe der Schlüsselwörter `static` und `instance` auch festlegen, ob sich eine Invariante auf ein Objekt oder auf seine Klasse beziehen soll. Damit sich eine Klasse in einem sichtbaren Zustand befindet, muss sie bereits statisch initialisiert sein. Die Beschreibung einer Invariante wird noch wesentlich präziser in der genannten Quelle vorgenommen, dies ist für die weiteren Begründungen in dieser Arbeit aber nicht entscheidend.

Beschreibung 2.11 – Schleifeninvarianten [LPC+13, S. 111f]**:** Schleifeninvarianten können genutzt werden, um die Korrektheit einer Schleife zu zeigen. Eine Schleifenvariante ist ein Ausdruck, der nach dem Schlüsselwort `maintaining`, `maintaining_redundantly`, `loop_invariant` oder `loop_invariant_redundantly` definiert wird. Dieser Ausdruck muss in JML nach jeder Schleifenausführung gelten.

Beschreibung 2.12 – Schleifenvariantenfunktion [LPC+13, S. 112]**:** Mit Hilfe des Schlüsselworts `decreasing`, `decreasing_redundantly`, `decreases` oder `decreases_redundantly` kann eine Schleifenvariantenfunktion definiert werden. Diese wird genutzt, um zu beweisen, dass eine Schleife terminiert. Hierzu wird ein Ausdruck verwendet, der zu einem `int` oder `long` evaluiert wird. Dieser Ausdruck muss mit jedem Schleifendurchlauf sinken, darf aber nicht negativ sein.

2.4.2 Spezifikation von Methoden

Um das Verhalten einer Methode mit JML zu spezifizieren, werden formale Verträge definiert [LPC+13, S. 6]. Im Folgenden sollen kurz die Bedeutung der Schlüsselworte erläutert werden. Bei diesen Erläuterungen wird, wenn nichts anderes angegeben wird, immer von *schwergewichtigen* Spezifikationen gesprochen.

Beschreibung 2.13 – Spezifikationsfälle [LPC+13, S. 67ff]**:** Die Schlüsselworte `behavior`, `normal_behavior` und `exceptional_behavior` (das Wort `behavior` kann jeweils auch `behaviour` geschrieben werden) stellen *schwergewichtige* Spezifikationen dar. Sollte dieses Schlüsselwort weggelassen werden, ist es eine „leichtgewichtige“ Spezifikation, welche sich grundsätzlich wie eine `behavior`-Spezifikation verhält [LPC+13, S. 64]. Bei einer leichtgewichtigen Spezifikation wird die Sichtbarkeit der Methode auch für die Spezifikation angenommen. Bei anderen, fehlenden Angaben wird nichts angenommen [LBR06, S. 6]. Bei `Requires`,

https://dx.doi.org/10.24355/dbbs.084-201801311013

Ensures, Assignable und Signals wird deshalb jeweils der Wert \not_specified angenommen [LPC+13, S. 76ff, S. 83f]. Der Behavior-Spezifikationsfall stellt die allgemeine Grundform dar. Wie auch der leichtgewichtige Spezifikationsfall nimmt der behavior-Spezifikationsfall nichts an, verlangt es aber, die Sichtbarkeit zu definieren. Bei einem normal_behavior-Spezifikationsfall wird vorausgesetzt, dass keine Exception geworfen wird. Der exceptional_behavior-Spezifikationsfall erlaubt es nicht, dass Nachbedigungen definiert werden. Es wird dafür angenommen, dass ein solcher Fall niemals normal terminieren kann, sondern eine Exception geworfen werden muss.

Beschreibung 2.14 – Requires [LPC+13, S. 76]**:** Requires definiert eine Vorbedingung. Ein Vertrag kann nur dann angewendet werden, wenn alle Vorbedingungen dieses Vertrags erfüllt sind. Dabei ist es möglich, dass ein Vertrag eine beliebige Anzahl an Vorbedingungen besitzt. Sollte kein Requires angegeben sein, wird standardmäßig true angenommen, also eine immer gültige Vorbedingung.

Beschreibung 2.15 – Ensures [LPC+13, S.77]**:** Ensures stellt eine Nachbedingung dar. Nach der Ausführung der Methode müssen alle Nachbedingungen gelten, sofern keine Exception aufgetreten ist. Eine Nachbedingung bezieht sich immer auf den Zustand nach der Ausführung einer Methode, sofern nicht das \old-Schlüsselwort verwendet wird. Wenn kein Ensures definiert ist, wird standardmäßig true angenommen. Dadurch wird ausgesagt, dass grundsätzlich jeder Zustand nach der Ausführung gültig ist, so lange dies nicht der Aussage eines anderen Schlüsselworts widerspricht.

Beschreibung 2.16 – Assignable [LPC+13, S. 83]**:** Das Assignable-Schlüsselwort definiert, welche vor der Ausführung der Methode bereits existierenden Speicherbereiche beschrieben werden können. Ein Speicherbereich muss hierbei ein konkret im Speicher existierender Ort sein und kann sich nicht auf ein Modellfeld beziehen. Sollte eine Assignable-Definition weggelassen werden, wird der Wert \everything angenommen. Dies hat zur Folge, dass jede erreichbare, benennbare Position im Speicher manipuliert werden darf. Sollte eine Methode keine Modifikationen im Speicher vornehmen dürfen, kann der Wert \nothing verwendet werden.

Beschreibung 2.17 – Signals und Signals_only [LPC+13, S.77ff]**:** Mit Hilfe des Signals-Schlüsselwortes lässt sich definieren, was gegeben sein muss, damit eine bestimmte Exception auftreten darf. Sollte kein Signals definiert sein, wird standardmäßig der Fall angenommen, dass jede Exception immer geworfen werden dürfte. Das bedeutet auch, dass die Spezifikation einzelner Exceptions das Werfen anderer Exception ausschließt. Signals_only bietet eine Kurzschreibweise, mit deren Hilfe definiert werden kann, welche Exceptions ausgelöst werden dürfen, ohne die jeweiligen Fälle zu definieren, die dann gelten müssen. Der Standardfall von Signals_only gilt sowohl für leichtgewichtige, als auch schwergewichtige Spezifikationsfälle. Hierbei wird für alle Spezifikationsfälle außer normal_behavior die Java-Throws-Definition aus der Methodensignatur übernommen. In dieser Liste müssen nur *Checked*-Exceptions gelistet werden, es tauchen also weder Runtime-

https://dx.doi.org/10.24355/dbbs.084-201801311013

Exceptions noch Errors auf [GJSB05, S.221f]. Für den normal_behavior-Fall wird angenommen, dass keine Exceptions auftreten dürfen.

Beschreibung 2.18 – also [LPC+13, S. 63f]**:** Mit Hilfe des also Schlüsselworts lässt sich eine Methodenspezifikation aus mehreren einzelnen Spezifikationsfällen zusammensetzen. Dies kann entweder genutzt werden, um mehrere Spezifikationsfälle anzugeben, die alle von der Implementierung erfüllt werden sollen, oder um bei einer überschriebenen Implementierung die vorherige Spezifikation nicht zu überschreiben, sondern das Verhalten damit näher zu spezifizieren. Dadurch dass mit dem also bei einem Subtypen die Spezifikation ergänzt wird, wird das *Liskovsche Substitutionsprinzip* eingehalten (siehe Abschnitt 2.2).

2.5 KeY

Im Folgenden soll KeY kurz vorgestellt werden. Hierbei wird zunächst auf den Hintergrund und den allgemeinen Funktionsumfang eingegangen, um anschließend die Technik dahinter näher vorzustellen. Diese Vorstellung soll jedoch nicht das Vorgehen bei deduktiver Verifikation in der Tiefe behandeln, sondern für den Rest der Arbeit wichtige Begriffe erläutern. Abschließend soll die Oberfläche kurz vorgestellt werden, damit eine bessere Vorstellung des Systems gegeben ist.

Bei dem KeY-Projekt handelt es sich um ein Forschungsprojekt, welches sich seit 1998 die „Integration von formalen Softwareanalysemethode, wie [...] formale Verifikation, in das Gebiet der mainstream Softwareentwicklung“ als Ziel gesetzt hat. Aus diesem Projekt heraus entstand das KeY-Framework, welches unter anderem das KeY-System enthält. Dieses wird im Folgenden nur noch als KeY bezeichnet. KeY verwendet JML zur formalen Spezifikation von Java-Quelltext [ABB+16, S. 1f].

KeY unterstützt grundsätzlich Java Card [ABB+16, S. 3] und die APIs zu Java Card und Real-Time Java [ABB+16, S. 6]. Hierfür bietet KeY die Möglichkeit, die Erfüllung der formalen Spezifikation zu verifizieren [ABB+16, S. 1]. Der volle Sprachumfang von Java wird dabei aber nicht unterstützt. Zum Beispiel wird angenommen, dass Programme immer sequentiell ablaufen. Auch werden Sprachfeatures wie Generics oder Fließkommazahlen nicht unterstützt [ABB+16, S. 3, S. 51]. Trotz dieser Einschränkungen soll es aber auch möglich sein, komplexe Java Methoden zu verifizieren, sofern sie nur unterstützte Sprachfunktionen enthalten [ABB+16, S. 9].

Beweise setzen sich in KeY aus automatisch ausgeführten Beweisschritten und manuellen Interaktionen des Benutzers zusammen. Der Anwender kann einen Beweis beispielsweise automatisch starten und das System mit manuellen Schritten unterstützen, sofern das System stoppen sollte. Dies kann der Fall sein, wenn eine zuvor definierte maximale Beweisschrittanzahl erreicht wurde oder der automatische Beweiser keine weiteren, anwendbaren Schritte ermitteln kann [ABB+16, S.496].

Damit KeY den Java-Quelltext gegen eine in JML vorliegende Spezifikation verifizieren kann, muss diese zunächst in Logik übersetzt werden. Als Zielsprache wird Java Dynamic Logic, eine dynamische Logik, eingesetzt [ABB+16, S. 12]. Diese Umwandlung übernimmt KeY beim Laden des Problems automatisch [ABB+16, S. 496]. Der

https://dx.doi.org/10.24355/dbbs.084-201801311013

Anwender oder KeY können nun schrittweise Deduktionen auf dieser Logik ausführen, was auf einer symbolischen Ausführung basiert [ABB+16, S. 50]. Die Regeln, die für diese Schritte ausgeführt werden, nennen sich Taclets. Sie enthalten neben der Deklaration des Taclets, den logischen Schritten und dem Schema der Regel auch Hinweise, wann die Regel angewendet werden kann [ABB+16, S. 11]. Ein Taclet ändert dabei pro Anwendung höchstens eine Formel [BRR08, S. 111]. Ein Beispiel für ein solches Taclet kann in Quelltext 2.2 gesehen werden. Bei diesem Beispiel handelt es sich um eine Regel, die aus *FALSE* ein $\neg$*TRUE* macht. KeY bietet eine Vielzahl an vordefinierten Taclets (in KeY 2.6.1 sind es 1681 Stück).

```
1 false_to_not_true {
2   \schemaVar \term boolean bo;
3
4   \find ( bo = FALSE )
5   \replacewith ( !bo = TRUE )
6   \heuristics ( concrete, simplify_boolean )
7 }
```

Quelltext 2.2: Ein Beispiel Taclet: false_to_not_true (Quelle: KeY 2.6.1)

Sollte der automatische Beweisführer verwendet werden, wählt er die anzuwendenden Taclets aus. Dies geschieht anhand von globalen Strategien, die die anwendbaren Taclets priorisieren und die jeweils passendste Regel auswählen [ABB+16, S. 108]. Zusätzlich zu diesen Strategien kann KeY auch externe Entscheidungsprozeduren einsetzen, um die automatische Beweisführung zu unterstützen [ABB+16, S. 537]. Das Verhalten lässt sich dabei mit Hilfe von verschiedenen Optionen anpassen. Diese werden in Kapitel 4 näher vorgestellt.

KeY bietet zusätzlich zu dem geladenen Quelltext noch weitere Methoden- und Klassensignaturen aus einer Standardbibliothek. Dieser Quelltext ist in einer Bibliothek namens *JavaRedux* hinterlegt [BHS13, S. 31]. Sollte kein alternativer Klassenpfad angegeben sein, wird immer *JavaRedux* geladen. Die Möglichkeiten des Anwenders werden in Abbildung 2.2 dargestellt.

https://dx.doi.org/10.24355/dbbs.084-201801311013

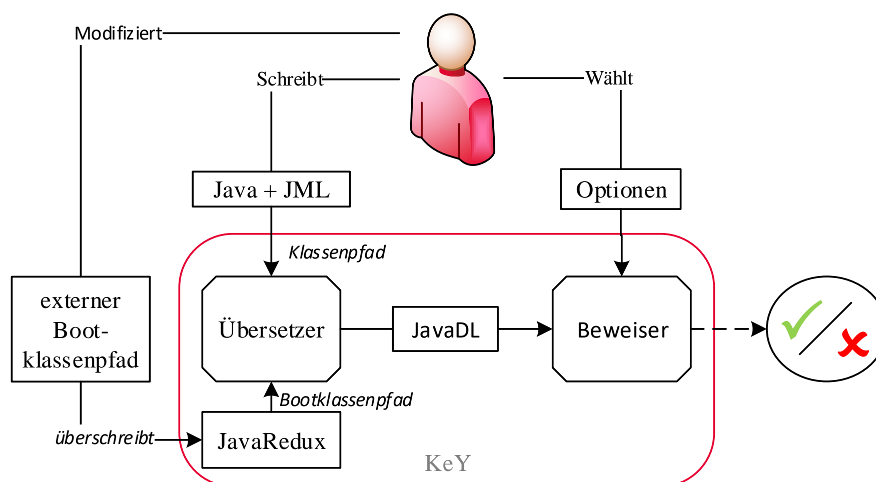

Abbildung 2.2: Einfluss auf automatische Beweise in KeY (eigene, erweiterte Darstellung, basiert auf [ABB+16, S. 496]).

https://dx.doi.org/10.24355/dbbs.084-201801311013

3. Erstellung von Verträgen für OpenJDK

In dem vorherigen Kapitel wurden Grundlagen über die Entwicklung mit Java, JML und KeY dargelegt. Mit Hilfe dieser Grundlagen sollen in diesem Kapitel formale Spezifikationen in Form von Verträgen erarbeitet werden. Der Fokus liegt in diesem Kapitel auf der Erstellung der Verträge. Die für die Verifikation in KeY benötigten Parameter werden im nächsten Kapitel betrachtet.

In diesem Kapitel geht es primär um die Erstellung der Verträge. Der Fokus liegt hierbei auf der Arbeit mit JML und dem Java-Quelltext. Als Quelltext wird die Implementierung des OpenJDK 6 genutzt. Es wird im Folgenden erklärt, wie der Quellcode für diese Arbeit verändert wurde und wie die Verträge erstellt wurden.[1] Die jeweiligen Gedanken dazu werden in diesem Kapitel dargestellt. Sollte ein Vertrag im Rahmen dieser Arbeit erstellt worden sein, welcher jedoch keinen neuen, interessanten Aspekt mit sich bringt, so befindet er sich im Abschnitt A.2.

Um diese Ziele zu erreichen, wird zunächst erläutert, wieso das OpenJDK 6 als Grundlage verwendet wird (Abschnitt 3.1). Anschließend werden die vorgenommenen Modifikationen am Quellcode und die Auswahl der Dateien für den Klassenpfad erläutert (ebenfalls Abschnitt 3.1). Für die Erstellung der Verträge wird zunächst die `Arrays.copyOf(Object[], int)` Methode vorgestellt, die von anderen Methoden an verschiedenen Stellen genutzt wird (Abschnitt 3.2). Anschließend werden exemplarisch einige Funktionen des Collection Frameworks vorgestellt (Abschnitt 3.3).

3.1 Allgemeines Vorgehen

Im Rahmen dieser Arbeit soll an Hand von Stichproben ermittelt werden, wie gut Software Entwickler mit KeY Java-Quelltext verifizieren können. Um realistischen

[1] Die Arbeit soll aus den Augen eines Softwareingenieurs erstellt werden. Wieso ich bin der Meinung bin, diese Anforderung zu erfüllen, wird in Abschnitt A.7 kurz dargestellt

https://dx.doi.org/10.24355/dbbs.084-201801311013

Quelltext als Basis für diese Untersuchung zu nutzen, soll der Quelltext der Java Platform API verwendet werden. Die Java Platform API steht jedoch in mehreren Fassungen zur Verfügung. In diesem Abschnitt soll zunächst erläutert werden weshalb die Implementierung aus dem OpenJDK 6 verwendet wird, um anschließend den allgemeinen Prozess der Erstellung der Spezifikationen vorzustellen. Anschließend wird vorgestellt, wie der Quelltext überarbeitet werden muss, damit er mit KeY verifiziert werden kann.

Auswahl der Implementierung

Für die Auswahl der Implementierung spielen vor allem zwei Faktoren eine Rolle. Als erstes Kriterium muss die Lizenz der Implementierung es ermöglichen, den Quelltext erneut zu veröffentlichen, damit er zum einen in dieser Arbeit abgedruckt werden darf und zum anderen von anderen Arbeiten als Grundlage für weitere Spezifikationen dienen kann. Das zweite Kriterium ist die Verbreitung der Implementierung: Eine weiter verbreitete Implementierung müsste zum einen stabiler und damit leichter verifizierbar sein, zum anderen ist eine Spezifikation für eine weiter verbreiterte Implementierung potenziell für mehr Nutzer interessant.

Es gibt drei zur Auswahl stehende Varianten: JavaSE, OpenJDK und GNU Classpath. Der Quelltext von JavaSE steht unter der *Oracle Binary Code License* [obl]. Diese erlaubt es leider nicht, den Quelltext erneut zu veröffentlichen [obl, Abschnitt „SOURCE CODE“], weshalb diese Implementierung ausgeschlossen wird. Seit Java 1.6 steht auch noch OpenJDK zur Verfügung. Diese wurde unter der GPLv2 veröffentlicht [opea] (siehe Abschnitt A.1). Diese Lizenz ermöglicht es, den Quelltext erneut zu veröffentlichen, sofern die Quellen und Verweise auf die Lizenz korrekt geführt, beziehungsweise übernommen werden. Einige Klassen sind explizit von dieser Lizenz ausgeschlossen. Diese Ausnahmen werden in der Lizenz als „CLASSPATH EXCEPTION“ bezeichnet. Sie erlauben es uns jedoch, Modifikationen vorzunehmen und zu veröffentlichen. GNU Classpath steht unter der *GNU General Public License* und lässt sich damit ebenfalls erneut veröffentlichen. Da es sich bei OpenJDK jedoch um eine bekanntere Variante handelt als bei GNU Classpath, wird diese verwendet.

Ein Ziel dieser Arbeit ist es, den vorliegenden Quelltext zu verifizieren, sollen möglichst wenig Änderungen an eben jenem vorgenommen werden. Dadurch wird der Quelltext weniger verfälscht und der manuelle Aufwand für die Verifikation ist geringer. Da KeY zwar grundsätzlich Java unterstützt, aber nicht alle Sprachelemente (siehe Abschnitt 2.5), spielt die Auswahl der richtigen Java-Version für die nötigen Anpassungen eine große Rolle. Es stehen auch verschiedene Versionen von OpenJDK zur Verfügung, welche auf verschiedenen Java-Versionen basieren (siehe Abschnitt 2.1). Da neue Versionen Funktionen hinzufügen, jedoch keine entfernen, wird eine ältere Java-Version und damit auch eine ältere OpenJDK-Version genutzt: OpenJDK 6.

Nötige Anpassungen am vorliegenden Quelltext

Das OpenJDK bietet den Quelltext der Java Platform API an. Dieser umfasst neben dem eigentlich Java-Quelltext auch eine Vielzahl an JavaDoc-Einträgen, welche als informelle Spezifikation der API dienen (siehe Abschnitt 2.1.4). Mit Hilfe dieser

https://dx.doi.org/10.24355/dbbs.084-201801311013

Spezifikationen werden jeweils formale JML-Spezifikationen erstellt, mit deren Hilfe anschließend KeY die jeweiligen Implementierungen verifizieren kann. Das Erstellen der JML-Spezifikationen aus den informellen JavaDoc-Einträgen heraus ist jedoch ein manueller Prozess. In diesen Einträgen wird das Verhalten in englischer Sprache informal definiert, muss also vom Entwickler interpretiert werden, wobei es zu Missverständnissen bei der Interpretation kommen kann. Dies wird zusätzlich verstärkt, da nicht immer das komplette Verhalten vollständig spezifiziert wurde. An dieser Stelle müssen implizite Annahmen mit Hilfe von Intuition identifiziert und in explizite Spezifikationen umgewandelt werden.

Trotz dieser Entscheidung müssen nach wie vor verschiedene Anpassungen am Quelltext vorgenommen werden. Um den manuellen Aufwand für die folgenden Verifikationen weiter zu reduzieren, wird die Anzahl der zu verarbeitenden Klassen möglichst gering gehalten. Dies wird erreicht, indem für die angestrebten Verifikationen unnötige Klassen, Interfaces und Methoden entfernt werden. Dadurch befinden sich im erstellten Klassenpfad nur die für die zu verifizierenden Methoden nötigen Klassen und Interfaces.

Wenn dieser möglichst kleine Ausschnitt der Java Platform API übrig ist, wird er vor der Spezifikation angepasst. Der Quelltext umfasst mehrere Elemente, mit denen KeY nicht arbeiten kann. Unseren Wissens nach gibt es keine Quelle, die beschreibt, was genau von KeY unterstützt wird und was nicht. In Abschnitt 2.5, wird eine grobe Übersicht gegeben. Da die Verarbeitung von Java Generics nicht möglich ist, werden diese entfernt. Das KeY-Plugin für Eclipse stellt eine Funktionalität zur Verfügung, um diese automatisch zu entfernen. Leider ist es für Eclipse nicht ohne weiteres möglich, die Standardklassen aus der Platform API zu bearbeiten, da diese Klassen über den Klassenpfad bereits vorhanden sind. Es ist für diese Arbeit deshalb leichter, die Generics manuell zu entfernen. Alle Klassen, die im Weiteren dieser Arbeit behandelt werden und Generics enthalten, schränken die hierbei verwendeten Klassen nicht ein, geben also nur ein Generic wie `<T>` an und nicht `<T implements X>`. Es wird also immer einfach die `Object`-Klasse verwendet. Im Rahmen dieser Arbeit sind jedoch noch weitere Einschränkungen aufgefallen. Annotationen, wie `@Deprecated` und `@SurpressWarnings`, lassen sich ebenfalls nicht verarbeiten. Sie haben allerdings keine Auswirkung auf das zu verifizierende Verhalten des Quelltexts und werden aus diesem Grund einfach gelöscht.

Der Bootklassenpfad in KeY und JavaRedux

Ein weiteres Problem stellen manche Java-Felder oder Methoden dar. Es kann dazu kommen, dass beim Verarbeiten einer Klasse in KeY eine Fehlermeldung auftritt, die von einem `UncollatedReferenceQualifier` spricht. Der Fehler kommt beispielsweise bei einem Zugriff auf das `Character.MAX_RADIX`-Feld vor, auch wenn im Klassenpfad beispielsweise die `Character`-Klasse vorhanden ist und das `MAX_RADIX`-Feld in dieser Klasse definiert ist. Dies liegt daran, dass KeY beim Verarbeiten eines Klassenpfads zunächst *JavaRedux* (siehe Abschnitt 2.5) als sogenannten *Bootklassenpfad* einbindet, sofern kein anderer angegeben wird. Die in *JavaRedux* hinterlegten Klassen lassen sich mit Hilfe der zu KeY gehörenden *key.core.jar*-Datei betrachten. Wenn man in dieser Datei den Pfad `de/uka/ilkd/key/java/JavaRedux/` öffnet, handelt es sich um den geladenen Klassenpfad. In diesem

https://dx.doi.org/10.24355/dbbs.084-201801311013

Pfad kann beispielsweise die java.lang.Character-Klasse geöffnet werden (siehe Quelltext 3.1) und es fällt auf, dass das `MAX_RADIX`-Feld nicht enthalten ist. Das Fehlen von Feldern und Methoden trifft auf mehrere Klassen zu und hängt nicht mit OpenJDK zusammen, da auch Felder und Methoden fehlen, die bereits in JavaSE 1.2 vorhanden waren. Die betroffenen Klassen könnten direkt in *JavaRedux* manipuliert werden. Das Manipulieren in der *jar*-Datei hat sich im Rahmen dieser Arbeit aber als instabile Lösung erwiesen, da *JavaRedux* teilweise nicht mehr fehlerfrei eingelesen werden konnte. Wenn beispielsweise die `java.util.Collection`-Klasse aus JavaRedux entfernt wird, kann kein Java-Quelltext mehr eingelesen werden. Es erscheint die Meldung „Position von java.util.Collection nicht gefunden". Aus diesem Grund wird eine Kopie von *JavaRedux* angelegt und in KeY als expliziter *Bootklassenpfad* verwendet. In diesem können anschließend die jeweilig nötigen Modifikationen vorgenommen werden. So kann beispielsweise das `MAX_RADIX`-Feld hinzugefügt werden, sodass KeY dieses Feld verarbeiten kann. Auch können vordefinierte Klassen, wie das Collections-Interface entfernt werden, so dass dieses aus dem *Klassenpfad* angezogen wird.

```
1 package java.lang;
2
3 /**
4  * Exists only for proving EVotingMachine
5  */
6 public final class Character extends java.lang.Object implements java
      .io.Serializable, java.lang.Comparable
7 {
8     static int digit(char ch, int radix);
9 }
```

Quelltext 3.1: Die Character-Klasse aus JavaRedux

Leider ermöglicht KeY es derzeit nicht, beim Verwenden der grafischen Benutzeroberfläche den *Bootklassenpfad* zu verändern (dies geht aus der Aussage eines KeY-Entwicklers hervor), von daher wird diese nicht für die Durchführung der Verifikation genutzt. Die Verifikation geschieht aus diesem Grund mit einer selbst geschriebenen Schnittstelle zu KeY. Da der Fokus in diesem Kapitel auf der Spezifikation liegt, wird an dieser Stelle nicht näher auf die Implementierung eingegangen. Dies erfolgt in Abschnitt 5.2.

Mit Hilfe des so erstellten Klassenpfades und dem editierbaren *Bootklassenpfad* können nun in den folgenden Abschnitten zunächst Spezifikationen für den Quelltext erstellt und diese anschließend mit KeY verifiziert werden.

3.2 Hilfsmethode Array.copyOf(Object[], int)

Die `copyOf(Object[], int)`-Methode ist kein Teil des `Collection`-Interfaces, wird aber für die Verifikation der `ArrayList` benötigt und soll deshalb an dieser Stelle behandelt werden. Da die Methode an mehreren Stellen in der Java Platform API genutzt wird, wird sie als Hilfsmethode bezeichnet. Der JavaDoc-Eintrag und die Signatur zu copyOf sind in Quelltext 3.2 zu sehen. Die im Rahmen dieser Arbeit mit JML-Spezifikationen versehene Signatur kann in Quelltext 3.3 eingesehen

https://dx.doi.org/10.24355/dbbs.084-201801311013

werden. Sollte sich im Folgenden eine Zeilenangabe auf die informelle Spezifikation beziehen, so ist sie in geschweiften Klammern geschrieben (beispielsweise $\{12 - 14\}$). Für Zeilenangaben aus der resultierenden Signatur werden spitze Klammern verwendet (beispielsweise $\langle 12 - 14 \rangle$).

Der Beschreibung zufolge dient die Funktion dazu, eine Kopie eines Arrays anzulegen. Wichtig hierbei ist jedoch, dass man der Kopie eine neue Größe zuweisen kann. Hierfür wird der Parameter `newLength` verwendet $\{12\}$. Das ursprüngliche Array wird im Parameter `original` übergeben $\{11\}$.

Zunächst sollen die Fehlerfälle formal spezifiziert werden, um im Anschluss auf den `normal_behavior`-Spezifikationsfall einzugehen. Laut dem JavaDoc-Eintrag können zwei `Exceptions` geworfen werden: eine `NegativeArraySizeException` oder eine `NullPointerException` $\{15, 16\}$. Die `NegativeArraySizeException` tritt auf, wenn `newLength` negativ ist $\langle 2 - 4 \rangle$, eine `NullPointerException` wird geworfen, wenn `original` `null` sein sollte $\langle 6 - 8 \rangle$. Diese werden in eigenen `exceptional_behavior`-Fällen abgebildet $\{15 - 23\}$. Hierbei muss beachtet werden, dass ein `signals` zur Folge hat, dass eine Exception auftreten darf, aber nicht auftreten muss (siehe Beschreibung 2.17 und Abschnitt 3.4.3).

Sollte keine dieser beiden Exceptions eintreten, soll die Methode ohne einen geworfenen Fehler ausgeführt werden $\{2 - 9\}$ $\langle 10 - 21 \rangle$. Dieser Fall soll nun näher betrachtet werden. Es ist in der informellen Spezifikation nicht ausgeschlossen, dass das `original`-Array `null`-Werte enthalten kann. Das bedeutet, dass der Parameter mit dem Schlüsselwort `nullable` (siehe Beschreibung 2.9) versehen werden muss $\langle 23 \rangle$. Eine nähere Ausführung dazu ist unter Abschnitt 3.4.1 zu finden. Durch den Einsatz von `nullable` muss jedoch explizit ausgeschlossen werden, dass `original == null` sein kann $\langle 11 \rangle$, da sonst eine Überschneidung mit dem zweiten Fehlerfall entstehen kann $\langle 6 - 8 \rangle$. Aus diesem Grund wird auch verlangt, dass der Parameter `newLength` ≥ 0 sein muss $\langle 12 \rangle$. Die `nullable`-Argumentation gilt auch für den Rückgabewert, da dieser schon per Spezifikation `null`-Werte enthalten kann $\langle 13, 23 \rangle$.

Da die Methode nur kopieren soll und von weiteren Tätigkeiten die Rede ist, wird angenommen, dass die Methode `pure` ist, also seiteneffektfrei ist und terminieren muss $\{6\}$ $\langle 25 \rangle$. Auch kann aus der Tatsache, dass eine Kopie des `original`-Arrays angelegt wird $\{2\}$, erwartet werden, dass das Ergebnis-Array ein neues ist (also `\fresh`) und damit auch ein anderes Objekt ist als `original`$\langle 14, 15 \rangle$. Außerdem ist definiert, dass das neue Array als Länge `newLength` hat $\{3\}$, das Array muss also auch ungleich `null` sein. Aus „The resulting array is of exactly the same class as the original array“$\{8 - 9\}$ folgt außerdem die Nachbedingung `\typeof(\result) == \typeof(original)` $\langle 16 \rangle$ (siehe Beschreibung 2.6).

Als abschließender Teil für die formale Spezifikation muss noch die Umsetzung der Beschreibung der Werte des Arrays $\{5 - 9\}$ definiert werden. Das neue Array soll die Länge `newLength` haben $\{12\}$. Daraus folgt, dass das `\result`-Array genau diese Länge haben muss $\langle 17 \rangle$. Für jedes Arrayfeld, dass in beiden Arrays existiert, müssen die Felder beider Arrays identisch sein $\langle 20 - 21 \rangle$. Hierfür wird mit einem Allquantor über das Array iteriert. Als Variable wird ein `int` i genutzt, welches zwischen 0 und der Länge des kleineren Arrays liegt. Dies wird erreicht, indem man verlangt, dass

https://dx.doi.org/10.24355/dbbs.084-201801311013

```
/**
Copies the specified array, truncating or padding with nulls (if
necessary) so the copy has the specified length.  For all indices
that are valid in both the original array and the copy, the two
arrays will contain identical values.  For any indices that are valid
in the copy but not the original, the copy will contain null.
Such indices will exist if and only if the specified length is
greater than that of the original array. The resulting array
is of exactly the same class as the original array.

@param original the array to be copied
@param newLength the length of the copy to be returned
@return a copy of the original array, truncated or padded with nulls
to obtain the specified length
@throws NegativeArraySizeException if <tt>newLength</tt> is negative
@throws NullPointerException if <tt>original</tt> is null
@since 1.6
*/
public static Object[] copyOf(Object[] original, int newLength)
```

Quelltext 3.2: JavaDoc und Signatur zu Arrays.copyOf()

```
/*@
  @ public exceptional_behavior
  @ requires newLength < 0;
  @ signals (NegativeArraySizeException e) true;
  @
  @ also public exceptional_behavior
  @ requires original == null;
  @ signals (NullPointerException e) true;
  @
  @ also public normal_behavior
  @ requires original != null;
  @ requires newLength >= 0;
  @ ensures \result != null;
  @ ensures \fresh(\result);
  @ ensures \result != original;
  @ ensures \typeof(\result) == \typeof(original);
  @ ensures \result.length == newLength;
  @ ensures (\forall int i; 0 <= i && i < \result.length &&
  @       i < original.length; original[i] == \result[i]);
  @ ensures (\forall int i; original.length <= i &&
  @      i < newLength; \result[i] == null);
  @*/
public static /*@ nullable pure@*/ Object[] copyOf(/*@ nullable @*/
     Object[] original, int newLength)
```

Quelltext 3.3: Spezifikation zu Arrays.copyOf()

https://dx.doi.org/10.24355/dbbs.084-201801311013

i kleiner als die Länge des Resultats und des `original`-Arrays sein muss. Für alle i, die diese Bedingung erfüllen, soll der Feldinhalt beider Arrays gleich sein. In der informellen Spezifikation ist auch beschrieben, dass alle Felder, die nur in der Kopie existieren, den Wert `null` haben {5 – 6}. Dies wird erneut mit einem Allquantor umgesetzt ⟨20⟩. Bei diesem wird als untere Grenze die Länge des `original`-Arrays und für die obere Grenze `newLength` genutzt.

Für diese Methode wird jedoch keine Verifikation durchgeführt. Die Methode verwendet die `getClass()`-Methode, für die derzeit noch kein Vertrag definiert ist. Dieses Problem lässt sich zwar noch beheben, jedoch würde es die Problematik nur verschieben. Die Methode ruft nämlich die `copyOf(Object[], int, Class)`-Methode auf, welche wiederum verifiziert werden müsste. Diese Methode verwendet zum einen das `.class`-Feld eines Objekts, was mit KeY derzeit nicht verarbeitet werden kann (das `class`-Schlüsselwort), zum anderen ruft sie jedoch die native Methode `System.arraycopy`-Methode auf. Eine native Methode kann mit KeY nicht verifiziert werden, da für diese kein Java-Code vorliegt. Von daher wird von dem Aufwand, welcher mit der Verifikation von `copyOf(Object[], int)` einhergeht abgesehen, da das Problem nur verschoben werden würde, dafür aber einen sehr hohen, manuellen Aufwand verursachen würde.

3.3 Verträge für die Java Collections

Dieser Abschnitt soll beispielhaft Teile des Java Collections Framework spezifizieren, um diese im weiteren Verlauf der Arbeit zu verifizieren. Hierbei wurde die Collections-API gewählt, da diese mit ihrer Implementierung vieler standard Datenstrukturen praxisnahe Quelltextbeispiele liefert. Da der Gesamtumfang der Collections-API jedoch sehr groß ist und es auch eine Vielzahl verschiedener Implementierungen gibt, würde das Verifizieren des gesamten Framework den Umfang der Arbeit weit überschreiten. Als Beispiele können hier verschiedene Listen, wie die `ArrayList` oder die `LinkedList`, oder auch Sets, wie das `HashSet`, genannt werden. Im Folgenden soll der Fokus auf einige, wenige Methoden der API gelegt werden und dafür die Erstellung der Verträge der Methoden ausführlicher behandelt werden. Dazu wurden folgende Methoden des `Collection`-Interfaces ausgewählt:

- `size()`
- `isEmpty()`
- `contains()`
- `toArray()`
- `add(Object)`
- `remove(Object)`
- `clear()`

https://dx.doi.org/10.24355/dbbs.084-201801311013

Wenn eine Collection als Menge von Objekten betrachtet wird, können mit diesem Funktionsumfang Elemente hinzugefügt und entfernt werden, die Kardinalität kann bestimmt werden, und es kann geprüft werden, ob die Menge der leeren Menge entspricht. Zudem können alle Elemente ermittelt werden. Diese Funktionen wurden ausgewählt, da damit ein zentraler Funktionsumfang einer `Collection` abgebildet wird.

Auch wird für die späteren Beweise eine prominente Implementierung der Collection ausgewählt, damit die Spezifikationen auch verifiziert werden können. Dies geschieht, um den Fokus dieser Arbeit nicht nur auf das Collections-Framework zu legen, sondern auch Methoden mit anderer Funktionalität zu betrachten. Da die Implementierung `ArrayList` häufig genutzt wird, soll diese exemplarisch herangezogen werden.

3.3.1 Spezifikation von Modell-Feldern

Auf Grund des Liskovschen Substitutionsprinzips (siehe Abschnitt 2.2) sollten alle Subtypen eines Typen beweisbare Eigenschaften aufrecht erhalten. Aus diesem Grund wird die Spezifikation an Hand des Interfaces und nicht der Implementierung erstellt, da alle Implementierungen sich so verhalten sollten, wie das Interface als Supertyp dies festlegt. Zur Unterstützung für die Verträge werden Felder eingeführt. Das Einführen von neuen Java-Feldern würde jedoch die Implementierung in unnötigerweise weiter verändern. Um dies zu verhindern, werden Modellfelder verwendet, die nur für die formale Spezifikation existieren (siehe Beschreibung 2.1). In diesem Unterkapitel wird die Entwicklung dieser Felder vorgestellt.

Die Spezifikationen des Collection-Frameworks führen jedoch bereits gewisse Unterscheidungen bezüglich des Funktionsumfangs von Implementierungen ein. Diese umfassen, ob das Hinzufügen von Duplikaten oder `null` erlaubt sein soll und ob die Collection eine Reihenfolge hat. In der informellen JavaDoc-Spezifikation wird dies genutzt, um das Verhalten der Methoden abhängig von der Implementierung zu definieren. Die Umsetzung dieser Unterteilung kann in Quelltext 3.4 in den Zeilen 2 bis 4 gesehen werden (im weiteren abgekürzt durch $\langle 2 - 4 \rangle$).

```
1  /*@
2    @ public model boolean supportsDuplicates;
3    @ public model boolean supportsNull;
4    @ public model boolean isOrdered;
5    @
6    @ public model instance \bigint collectionSize;
7    @ public model instance nullable Object[] elements;
8    @
9    @ public model instance \locset changeable;
10   @*/
```

Quelltext 3.4: Die definierten Modellfelder für die Java-Collections

Für die kommenden Spezifikation wird außerdem ein Modellfeld hinzugefügt, welches die Größe der Collection abbildet. Da der JavaDoc-Eintrag der size()-Methode (siehe Quelltext 3.10, Zeile 1 – 2) einen Fall für Collections mit mehr als `Integer.MAX_VALUE` Elementen definiert, wird für die Größe der JML-Datentyp `\bigint`

https://dx.doi.org/10.24355/dbbs.084-201801311013

verwendet. Da die Größe von Instanz zu Instanz variieren kann, muss es sich hierbei um ein `instance`-Feld handeln ⟨6⟩. Auch wird ein Array definiert, welches alle Elemente der Collection enthält ⟨7⟩. Hierbei wird ein Array verwendet, da dies auch der Rückgabetyp der `toArray`-Methoden der `Collection` ist. Diese beiden Felder werden genutzt, um eine Abstraktion von den Methoden `size()` und `toArray()` zu bieten, welche äquivalente Funktionalitäten zur Verfügung stellen. Diese Abstraktionen sollen später das Definieren der jeweiligen Methoden vereinfachen, da so kein Zirkelschluss entstehen soll. Ansonsten müsste beispielsweise die formale Spezifikation der `size()`-Methode von sich selbst oder der Korrektheit der `toArray()`-Methode abhängen. Außerdem kann das Resultat der `size()`-Methode von der Größe der Collection abweichen (siehe Abschnitt 3.3.3).

Die Verträge, die im Folgenden definiert werden, sollen bereits in der Lage sein, veränderbare Felder anzugeben. Um dies zu erreichen, wird ein *dynamic frame* eingesetzt (siehe Beschreibung 2.4) ⟨9⟩. In diesem wird in der jeweiligen Implementierung gespeichert, welche Felder von schreibenden Operationen verändert werden. Dadurch ist es möglich, die Verträge aufzustellen, ohne bereits im Vorfeld zu wissen, welche Variablen geändert werden können.

3.3.2 Allgemeine Spezifikation der ArrayList

In diesem Unterkapitel wird ein Teil der für die ArrayList erstellten formalen Spezifikationen vorgestellt. Die Ausnahme stellen hierbei die Verträge der Methoden dar, die in den nächsten Unterkapiteln vorgestellt werden. Für die Spezifikationen wird zunächst auf die im vorherigen Unterkapitel erstellten Modellfelder eingegangen, um im Anschluss auf die nötigen Klasseninvarianten einzugehen (siehe Beschreibung 2.10).

```
1  /*@
2    @ private represents supportsDuplicates = true;
3    @ private represents supportsNull = true;
4    @ private represents isOrdered = true;
5    @
6    @ private represents collectionSize = size;
7    @ private represents elements =
8    @         Arrays.copyOf(elementData, collectionSize);
9    @
10   @ private represents changeable =
11   @         elementData, elementData.*, modCount, size;
12   @*/
```

Quelltext 3.5: Die definierten Modellfelder der ArrayList

Die im vorherigen Unterkapitel erstellten Felder existieren jetzt zwar, haben jedoch noch keine Werte, da diese in dem `Collection`-Interface nicht bekannt sein können. Die Belegung mit Werten wird von jeder Klasse separat vorgenommen. Da die `ArrayList`-Implementierung verifiziert werden soll, können wir als Basis für diese Werte den JavaDoc-Eintrag der `ArrayList` selbst (siehe Quelltext A.30) und des `List`-Interfaces verwenden (siehe Quelltext A.29). Der Eintrag des `List`-Interfaces kann hierbei benutzt werden, da die `ArrayList` dieses Interface implementiert und

https://dx.doi.org/10.24355/dbbs.084-201801311013

dieses Verhalten vorgibt. Es werden erneut Kurzschreibweisen definiert: {ArrayList 3 – 7} steht für die Zeilen 3 bis 7 aus dem JavaDoc der `ArrayList`, {List 3 – 7} ist das Äquivalent für die `List`. Wegen des Umfangs dieses Eintrags befindet er sich im Anhang der Arbeit. Eine `ArrayList` unterstützt Duplikate {List 7}, `null` {ArrayList 2 – 3} und hat eine Reihenfolge {List 1}. Die jeweiligen Modellfelder können deshalb mit `true` belegt werden (siehe Quelltext 3.5, Zeile 2 bis 4; als Kurzschreibweise wird ⟨Modellfelder 2 – 4⟩ verwendet).

Die Anzahl der Elemente der Collection werden in dem Feld `size` gespeichert (siehe Quelltext 3.6, Zeile 8). Deshalb wird das `collectionSize`-Modellfeld einfach mit `size` gleich gesetzt ⟨Modellfelder 6⟩. Außerdem gibt es ein Object-Array, welches alle Elemente enthält: Das Feld `elementData` (siehe Quelltext 3.6, Zeile 2 – 3). Dieses kann jedoch größer sein als die Collection {ArrayList 17 – 19}. In diesem Fall enthält das Array die Daten aber immer in den ersten n Feldern, wobei n die Größe der Collection ist. Dies geht nicht direkt aus der Spezifikation hervor, dafür aber aus der Implementierung. Beispielsweise gibt die `toArray()`-Methode einfach eine Kopie des Arrays mit der Länge n zurück. Um das Array in der richtigen Größe zur Verfügung zu stellen, wird die `Arrays.copyOf(Object[], int)`-Methode verwendet (siehe Abschnitt 3.2) ⟨Modellfelder 7, 8⟩.

```
1  /**
2   * The array buffer into which the elements of the ArrayList are
      stored.
3   * The capacity of the ArrayList is the length of this array buffer.
4   */
5  private transient Object[] elementData;
6
7  /**
8   * The size of the ArrayList (the number of elements it contains).
9   *
10  * @serial
11  */
12 private int size;
```

Quelltext 3.6: Signaturen und JavaDocs der elementData- und size-Felder

```
1  public class ArrayList extends AbstractList implements RandomAccess,
      Cloneable, java.io.Serializable
```

Quelltext 3.7: Signatur der ArrayList-Klasse

Das letzte Feld, welches noch festgelegt werden muss, ist das `changeable`-Modellfeld. Wenn sich die `ArrayList`-Instanz durch eine schreibende Methode, wie `add()` oder `remove` ändert, wird auch schreibend auf das `elementData`-Array zugegriffen, welches die Elemente speichert. Bei diesen Operationen ändert sich jeweils auch die Größe des Arrays und damit das `size`-Feld. Die `ArrayList` Implementierung erbt von der abstrakten `AbstractList`-Implementierung. `ArrayList` verwendet in mehreren Methoden noch ein in der Klasse `AbstractList` definiertes Feld: `modCount`. Bei `modCount` wird die Anzahl der Modifikationen an der `ArrayList` gespeichert. Dies dient dazu, dass ein `Iterator` über die Liste

https://dx.doi.org/10.24355/dbbs.084-201801311013

```
1  The number of times this list has been <i>structurally modified</i>.
2  Structural modifications are those that change the size of the
3  list, or otherwise perturb it in such a fashion that iterations in
4  progress may yield incorrect results.
5
6  <p>This field is used by the iterator and list iterator
7  implementation returned by the {@code iterator} and {@code
8  listIterator} methods. If the value of this field changes
9  unexpectedly, the iterator (or list iterator) will throw a
10 {@code ConcurrentModificationException} [...]
```

Quelltext 3.8: Auszug des JavaDocs des modCount-Feldes der AbstractList-Klasse

bemerkt, falls eine Modifikation während des Verwendens des `Iterators` vorgenommen wurde (siehe Quelltext 3.8 Zeile 1 und 6 – 10; der ganze Eintrag kann in Quelltext A.31, gesehen werden). Bei jeder Operation, die schreibend auf die `ArrayList` zugreift, werden also diese Felder mit manipuliert und werden deshalb als Elemente des `changeable`-Modellfelds gespeichert ⟨Modellfelder 10, 11⟩.

Für die folgenden Spezifikationen der ausgewählten Methoden der `Collection` und den in einem späteren Kapitel folgenden Beweisen der `ArrayList` müssen noch Invarianten definiert werden. So wird erwartet, dass das `elementData`-Array als Länge die Kapazität der `ArrayList` hat (siehe Quelltext 3.6, Zeile 3). Die Kapazität ist dabei immer mindestens so groß, wie die Größe der `Collection`, also die `size` {ArrayList 17 – 19}. `size` darf also höchstens so groß sein, wie `elementData` lang ist (siehe Quelltext 3.9, Zeile 1; als Kurzschreibweise ⟨Invariante 1⟩). Eine negative Listengröße ergibt keinen Sinn, da eine Menge auch keine negative Kardinalität haben kann ⟨Invariante 2⟩.

```
1  /*@ invariant size <= elementData.length;
2    @ invariant size >= 0;
3    @*/
4  private int size;
5
6  /*@ invariant elementData != null;
7    @ invariant \typeof(elementData) == \type(Object[]);
8    @*/
9  private transient Object[] /*@ nullable @*/ elementData;
```

Quelltext 3.9: Die Invarianten und Signaturen der Felder der ArrayList

Da eine `ArrayList` grundsätzlich auch `null` speichern kann {ArrayList 2 – 3}, muss auch das `elementData`-Feld `nullable` sein ⟨Invariante 9⟩. Da das Array jedoch immer existieren muss, muss dies nun separat verlangt werden ⟨Invariante 6⟩. Weniger offensichtlich ist die Forderung, dass das Array immer vom Typ `Object[]` ist. Dies fiel jedoch bei dem Beweis der `add()`-Methode auf, da bei dieser ein Object in das Array geschrieben wurde, jedoch der genaue Typ des Arrays nicht definiert war (siehe auch Abschnitt 3.4.2). Theoretisch könnte das Feld mit einem anderen beliebigen Array beschrieben werden, wie beispielsweise einem `String[]`. Dies hätte zur Folge, dass nur noch `String`-Implementierungen fehlerfrei in diesem gespeichert werden können, jedoch nicht mehr beliebige `Object`-Instanzen. Deshalb muss

https://dx.doi.org/10.24355/dbbs.084-201801311013

gefordert werden, dass das `elementData`-Arrays immer genau ein `Object`-Array und keine Subklasse von diesem sein darf ⟨Invariante 7⟩.

3.3.3 Spezifikation der Collection.size()

Das einzige Ziel der size-Methode ist es, die Anzahl der Elemente in der `Collection` zu ermitteln (siehe Quelltext 3.10, Zeile 1; Kurzschreibweise {1}). Bei dieser Methode handelt es sich deshalb um eine Methode, die nur einen Wert zurückgeben soll, also als `pure` definiert werden kann (siehe Quelltext 3.11 Zeile 8; Kurzschreibweise ⟨8⟩). Da wir bereits ein Hilfsfeld definiert haben, in der die Größe der Collection definiert ist, können wir auf dieses zurückgreifen (siehe Abschnitt 3.3.1). Die Methode soll `Integer.MAX_VALUE` als obere Grenze für den Rückgabewert verwenden {2 – 3}. Das bedeutet, dass die Größe der `Collection` zurückgegeben werden soll, wenn diese höchstens `Integer.MAX_VALUE` ist ⟨1 – 3⟩. Sollte sie größer sein, muss der Rückgabewert diesem entsprechen ⟨4 – 6⟩.

```
1 /**Returns the number of elements in this collection.  If this
2 collection contains more than <tt>Integer.MAX_VALUE</tt> elements,
    returns
3 <tt>Integer.MAX_VALUE</tt>.
4
5 @return the number of elements in this collection**/
6 int size();
```

Quelltext 3.10: JavaDoc und Signatur von Collection.size()

```
1 /*@ public normal_behavior
2   @ requires collectionSize <= Integer.MAX_VALUE;
3   @ ensures \result == collectionSize;
4   @ also public normal_behavior
5   @ requires collectionSize > Integer.MAX_VALUE;
6   @ ensures \result == Integer.MAX_VALUE;
7   @*
8 int /*@pure@*/ size();
```

Quelltext 3.11: Spezifikation von Collection.size()

```
1 public int size() {
2     return size;
3 }
```

Quelltext 3.12: Implementierung von ArrayList.size()

Der erste Spezifikationsfall ⟨1 – 3⟩ ließ sich mit KeY problemlos verifizieren. Bei dem zweiten Spezifikationsfall ⟨4 – 6⟩ kommt es jedoch zu einem Problem. Aus dem Beweisbaum von KeY geht hervor, dass KeY die Forderung, dass die `collectionSize` > `Integer.MAX_VALUE` = 2.147.483.647, dahingehend umsetzt, dass `self.size` ≥ 2.147.483.648 gelten muss. Die Nachbedingung wird von `\result` =`Integer.MAX_VALUE` umgeformt zu `self.size` = 2.147.483.647 (siehe Abbildung 3.1). Dies ist jedoch ein Widerspruch zu der Tatsache, dass `self.size` ein `int` ist,

https://dx.doi.org/10.24355/dbbs.084-201801311013

```
wellFormed(heap),
self.<created> = TRUE,
java.util.ArrayList::exactInstance(self) = TRUE,
measuredByEmpty,
self.size ≥ 2147483648,
self.<inv>
==>
self = null,
self.size = 2147483647 ∧ ∀ Field f; ∀ java.lang.Object o; true
```

Abbildung 3.1: Ausschnitt aus dem Beweisbaum des Beweises des zweiten Spezifikationsfalls von ArrayList.size()

und 2.147.483.648 einem Überlauf entsprechen müsste (siehe Abschnitt 2.1.2). Dieses Verhalten wird nicht erkannt, auch wenn es sich bei der eingestellten *intRules* um *javaSemantics* handelt und ein Überlauf erkannt werden müsste (siehe Beschreibung 4.21). Ein solcher Überlauf wird durch die Invariante `size` ≥ 0 abgefangen (siehe Quelltext 3.9, Zeile 2) und ist deshalb eigentlich nicht möglich. Der Beweis dieser Methode kann deshalb nicht direkt geschlossen werden. Mit der richtigen Konfiguration schlussfolgert KeY jedoch aus der Invariante, dass `size` $\leq$ `elementData.length` sein muss, dass `elementData.length` ein `int` sein muss und deshalb `size` auch $\leq$ 2.147.483.647 sein muss. Durch den dadurch entstehenden Widerspruch zwischen `size` $\geq$ 2.147.483.648 und `size` $\leq$ 2.147.483.647 lässt sich der Beweis jedoch schließen.

```
1 public int size() {
2     if(size > Integer.MAX_VALUE)
3         return Integer.MAX_VALUE;
4     return size;
5 }
```

Quelltext 3.13: Angepasste Implementierung von ArrayList.size()

3.3.4 Spezifikation von ArrayList.ensureCapacity(int)

Diese Methode ruft die Methode Arrays.copyOf(Object[], int) auf und ist damit abhängig von dieser (siehe Abschnitt 3.2). Sie wird außerdem von der Methode ArrayList.add(Object) (siehe Abschnitt 3.3.5) genutzt.

Die `ensureCapacity`-Methode bietet die Funktionalität, die Größe des `elementData`-Arrays zu vergrößern, sofern dieses nicht mindestens `minCapacity` Elemente speichern kann. Da das Array von dieser Methode gegebenenfalls vergrößert, nicht aber gelöscht werden soll, müssen nach dem Aufruf der Methode noch alle zuvor gespeicherten Elemente an der gleichen Stelle gespeichert sein (siehe Quelltext 3.14, Zeile 1; als Kurzschreibweise wird erneut {JavaDoc 2 – 4} eingeführt). Ansonsten könnte der Aufruf dieser Methode die Reihenfolge der Elemente in der Liste manipulieren. Dabei muss das Array hinterher nicht genau `minCapacity` groß sein, sondern kann auch größer sein {JavaDoc 3 – 4, 6}.

https://dx.doi.org/10.24355/dbbs.084-201801311013

```
1  /**
2   * Increases the capacity of this <tt>ArrayList</tt> instance, if
3   * necessary, to ensure that it can hold at least the number of
4   * elements specified by the minimum capacity argument.
5   *
6   * @param   minCapacity   the desired minimum capacity
7   */
8  public void ensureCapacity(int minCapacity)
```

Quelltext 3.14: JavaDoc und Signatur zu ArrayList.ensureCapacity(int)

```
1  /*@ public normal_behavior
2    @ requires minCapacity <= elementData.length;
3    @ assignable modCount;
4    @ ensures modCount != \old(modCount);
5    @
6    @ also
7    @ public normal_behavior
8    @ requires minCapacity > elementData.length;
9    @ ensures modCount != \old(modCount);
10   @ assignable modCount;
11   @ assignable elementData;
12   @ ensures \fresh(elementData);
13   @ ensures elementData.length >= minCapacity;
14   @ ensures (\forall int i; 0 <= i && i < size;
15   @         \old(elementData[i]) == elementData[i]);
16   @*/
17 public void ensureCapacity(int minCapacity)
```

Quelltext 3.15: Formale Spezifikation von ArrayList.ensureCapacity(int)

```
1  public void ensureCapacity(int minCapacity) {
2      modCount++;
3      int oldCapacity = elementData.length;
4      if (minCapacity > oldCapacity) {
5          Object oldData[] = elementData;
6          int newCapacity = (oldCapacity * 3)/2 + 1;
7          if (newCapacity < minCapacity)
8              newCapacity = minCapacity;
9          // minCapacity is usually close to size, so this is a win:
10          elementData = Arrays.copyOf(elementData, newCapacity);
11      }
12 }
```

Quelltext 3.16: Implementierung von ArrayList.ensureCapacity(int)

https://dx.doi.org/10.24355/dbbs.084-201801311013

Die Funktionalität der Methode soll im Folgenden in zwei verschiedenen Verträgen unterteilt spezifiziert werden (siehe Quelltext 3.15, Zeile 1 – 4 und 7 – 14; Kurzschreibweise ⟨1 – 4 / 7 – 14⟩). Die beiden Verträge unterscheiden sich dadurch, dass der erste Spezifikationsfall die Schleifenbedingung (siehe Quelltext 3.16, Zeile 4; Kurzschreibweise {Implementierung 4}) nicht erfüllen kann ⟨2⟩, der zweite diese dafür immer erfüllt ⟨8⟩. Diese implementierungsspezifische Unterscheidung erscheint zwar als unsauber, da sie sich direkt auf Implementierungsdetails bezieht, ein einzelner, unabhängiger Spezifikationsfall hat jedoch nicht das gewünschte Ergebnis erzielen können. Dieses Sonderverhalten soll erst im nächsten Absatz näher betrachtet werden. In diesem Absatz soll zunächst der erste, dann der zweite Spezifikationsfall erläutert werden. Durch die Unterteilung anhand der Bedingung wird beim ersten Spezifikationsfall nur `modCount` verändert, nicht aber `elementData` ⟨3⟩. An dieser Stelle wird aufgrund eines möglichen Integer-Überlaufs nicht verlangt, dass `modCount` steigt {Implementierung 2}, sondern nur, dass es sich verändert ⟨4⟩. Diese Forderung reicht für `modCount` aus, da die Iteratoren laut informeller Spezifikation auf eine Veränderung des Wertes reagieren und nicht zwangsläufig eine Erhöhung erwarten (siehe Quelltext A.31, Zeile 8 bis 10). Da `elementData` nicht verändert wird und per Vorbedingung größer sein muss als `minCapacity` ⟨2⟩, kann auf die Nachbedingung, dass `elementData` größer als `minCapacity` sein muss, verzichtet werden. Im Fall des zweiten Spezifikationsfalls wird `modCount` wie auch im ersten Fall verändert ⟨9, 10⟩, aber zusätzlich wird auch `elementData` verändert ⟨11⟩. Um sicherzustellen, dass es sich nach der Ausführung nicht um ein bereits existierendes Array handelt, wird verlangt, dass `elementData fresh` (siehe Beschreibung 2.5) sein muss ⟨12⟩. Dies wird benötigt, damit rufende Methoden beweisen können, dass bei einem schreibenden Zugriff auf `elementData` nur das im Vorfeld als `elementData` bekannte Objekt oder ein neues Objekt modifiziert wird. Da `elementData` geändert wurde, muss nun noch verlangt werden, dass das Objekt nach der Methodenausführung groß genug ist ⟨13⟩. Auch sollen, wie bereits erläutert, alle Elemente noch an der gleichen Stelle wie vor der Methodenausführung sein. Dies wird mit Hilfe einer Quantifizierung über alle von der Liste benutzten Indizes, 0 bis `size`, umgesetzt. Für jeden dieser Werte `i` muss das Element an der Stelle `i` von `elementData` vor der Ausführung identisch zu dem Element nach der Ausführung sein ⟨14,15⟩.

```
1 /*@ public normal_behavior
2   @ ensures modCount != \old(modCount);
3   @ ensures elementData.length >= minCapacity;
4   @ ensures (\forall int i; 0 <= i && i < size; \old(elementData[i])
        == elementData[i]);
5   @ ensures \new_elems_fresh(elementData);
6   @ assignable elementData;
7   @ assignable modCount;
8   @*/
9 public void ensureCapacity(int minCapacity)
```

Quelltext 3.17: Der ursprünglich erstellte Vertrag von ArrayList.ensureCapacity(int)

https://dx.doi.org/10.24355/dbbs.084-201801311013

Der Einsatz eines einzelnen Spezifikationsfalls, der etwas unpräziser ist, wirkt im ersten Moment sinnvoller, da auch die informelle Spezifikation keine Fälle unterscheidet. Wie im vorherigen Absatz jedoch dargestellt, wurden dennoch zwei Verträge erstellt. Der Nachteil dieser Lösung ist, dass die formale Spezifikation damit Abhängigkeiten zur Implementierung aufweist. Die ursprüngliche Spezifikation für diese Methode sah nur einen Spezifikationsfall ohne Vorbedingung vor (siehe Quelltext 3.17). Bis auf die fehlende Vorbedingung unterscheidet er sich vom zweiten, letztendlich erstellten Spezifikationsfall ⟨7 – 14⟩ nur durch den Einsatz des Schlüsselwortes `\new_elems_fresh`. Das Schlüsselwort hat die Wirkung, dass alle Objekte, die von einem Objekt aus erreichbar sind und vor dem Methodenaufruf nicht vorhanden waren, neu erstellt wurden [ABB+16, S. 323]. Diese Forderung war allerdings nicht stark genug, damit aufrufende Methoden bei einem schreibenden Zugriff auf `elementData` nach dem Aufruf der `ensureCapacity`-Methode noch sicherstellen können, dass sie kein anderes Objekt als das ursprüngliche `elementData` oder ein neu erstelltes Objekt manipulieren. Der Spezifikationsfall kann auch nicht dahingehend verändert werden, dass `fresh` verwendet wird, da dies implementierungsabhängig nicht immer der Fall ist. Auf Grund der Fallunterscheidung kann dies nur angenommen werden, wenn das Array nicht groß genug war. Auch das Aufteilen in zwei Nachbedingungen, die an Hand dieser Fallunterscheidung entweder `fresh` garantieren, oder dass das Array nach wie vor dasselbe Objekt ist, haben bei den rufenden Methoden nicht ausgereicht, damit der automatische Beweiser nachweisen konnte, dass bei dem schreibenden Zugriff nach dem Methodenaufruf nur `elementData` verändert wurde. Aus diesem Fall lässt sich erkennen, dass es sich in manchen Fällen lohnen kann, die Verträge abhängig von Implementierungsdetails zu erstellen. Dies hat zwar zur Folge, dass sich die Spezifikationen eventuell häufiger ändern, kann aber das Verifizieren von Methoden stark vereinfachen.

Die Methode konnte zwar mit Hilfe dieses Vertrags verifiziert werden, kann jedoch trotzdem abstürzen. Der `minCapacity`-Parameter ist in dem Vertrag nicht begrenzt. Ein negativer Wert würde zu keinem Problem führen, da er nur zu einer Änderung der `modCount` führen würde ⟨2-4⟩, weil `elementData.length` immer größer oder gleich 0 sein muss, da ein Array keine negative Länge haben kann (siehe Abschnitt 2.1.3 Das Verhalten von Arrays). Interessant ist jedoch das Verhalten der Methode bei `minCapacity` größer als `Integer.MAX_VALUE`-3. In diesem Fall würde ein neues Array mit der entsprechenden Größe instanziiert werden, da eine Multiplikation dieses Werts mit dem Faktor $1,5$ zu einem Überlauf führen würde und das Ergebnis deshalb kleiner als `minCapacity` wäre. Da dieser Wert per Annahme größer als `Integer.MAX_VALUE - 3` ist, lässt sich das Array nicht mehr instanziieren, sondern es tritt ein `Error` auf (siehe Abschnitt 2.1.3 Das Verhalten von Arrays). An dieser Stelle kann KeY dieses Verhalten nicht erkennen, da es in der Spezifikation der nicht verifizierten Methode `Arrays.copyOf` nicht festgelegt wurde, jedoch erkennt KeY dies auch in einer Methode nicht, die das Array nur anlegt (siehe Quelltext 3.18). Das von KeY angenommene Verhalten entspricht an dieser Stelle also nicht der Implementierung aktueller Java Virtual Machines (siehe Abschnitt 2.1.3).

https://dx.doi.org/10.24355/dbbs.084-201801311013

```
1 /*@ public normal_behavior
2   @ requires arrSize >= 0;
3   @ ensures \result != null;
4   @ ensures \result.length == arrSize;
5   @*/
6 private static /*@ pure nullable @*/ Object[]
7   createArray(int arrSize){
8     return new Object[arrSize];
9 }
```

Quelltext 3.18: Verifizierbare Instanziierung eines Arrays mit beliebiger, positiver Länge im int-Bereich

3.3.5 Spezifikation von Collection.add(Object)

Diese Methode ruft die Methode ArrayList.ensureCapacity(int) auf und ist damit abhängig von dieser (siehe Abschnitt 3.3.4). Die folgenden Spezifikationen verwenden die Methode Collections.contains(Object) als Query (siehe Abschnitt A.2.1).

Das Ziel der `add(Object)`-Methode ist es, ein Element der `Collection` hinzuzufügen. Im Falle einer `Collection` bedeutet das, dass im Nachhinein das Hinzuzufügende Element in der `Collection` enthalten sein muss (siehe Quelltext 3.19, Zeile 2 – 3; Kurzschreibweise {2–3}). Bei dieser Spezifikation wird auch erstmals zwischen verschiedenen, späteren Funktionen und damit Implementierungen unterschieden (siehe Abschnitt 3.3.1). So wird in der informellen Spezifikation beispielsweise bereits zwischen Implementierungen unterschieden, die Duplikate zulassen oder nicht zulassen {3 – 4}.

Es wurden fünf verschiedene Verträge für diese Methode erstellt (siehe Quelltext A.2; Zeile 1 – 32; Kurzschreibweise ⟨1 – 34⟩). In diesem Absatz sollen zunächst die Fehlerfälle definiert werden, im nächsten der fehlerfreie Fall. Der erste Spezifikationsfall ⟨1 – 3⟩ soll verhindern, dass eine Exception auftreten kann, die nicht in der Spezifikation {23 – 32} aufgelistet ist. Die Spezifikation eines Falls für die `UnsupportedOperationException` ist zwar möglich, für diese Arbeit aber nicht interessant, da nur die `ArrayList`-Implementierung betrachtet wird und diese die Operation unterstützt und die Exception deshalb nicht wirft. Sie verhält sich äquivalent zu dem Spezifikationsfall der `NullPointerException` ⟨8 – 11⟩. Die Beschreibung zur `NullPointerException` legt fest, dass eine `NullPointerException` auftritt ⟨11⟩, wenn das übergebene Objekt `null` ⟨10⟩ ist und die `Collection` dies nicht unterstützt ⟨9⟩ {27 – 28}. Durch das Entfernen der Generics kann die `ClassCastException` leider nicht abgebildet werden, da nur im Generic hinterlegt ist, welche Objekte - von der Klasse her betrachtet - hinzugefügt werden dürfen {25 – 26}. Die anderen beiden Exceptions {29 – 32} sind sehr allgemein gehalten und es kann aus der Sicht der `Collection`-Klasse nicht spezifiziert werden, welche Eigenschaften dies beeinflussen können. Aus diesem Grund werden hierfür keine eigenen Verträge angelegt. Auch die `ArrayList`-Implementierung sieht solche Fehlerfälle nicht in der formalen Spezifikation vor.

Nachdem die Fehlerfälle spezifiziert wurden, werden nun die fehlerfreien Fälle dargestellt. In jedem Fall kann die Methode nur fehlerfrei durchlaufen werden, wenn entweder `null`-Werte unterstützt werden oder das hinzuzufügende Objekt nicht `null`

```
/**
Ensures that this collection contains the specified element (optional
operation).  Returns <tt>true</tt> if this collection changed as a
result of the call.  (Returns <tt>false</tt> if this collection does
not permit duplicates and already contains the specified element.)<p>

Collections that support this operation may place limitations on what
elements may be added to this collection.  In particular, some
collections will refuse to add <tt>null</tt> elements, and others
will impose restrictions on the type of elements that may be added.
Collection classes should clearly specify in their documentation any
restrictions on what elements may be added.<p>

If a collection refuses to add a particular element for any reason
other than that it already contains the element, it <i>must</i> throw
an exception (rather than returning <tt>false</tt>).  This preserves
the invariant that a collection always contains the specified element
after this call returns.

@param e element whose presence in this collection is to be ensured
@return <tt>true</tt> if this collection changed as a result of the
        call
@throws UnsupportedOperationException if the <tt>add</tt> operation
        is not supported by this collection
@throws ClassCastException if the class of the specified element
        prevents it from being added to this collection
@throws NullPointerException if the specified element is null and
        thiscollection does not permit null elements
@throws IllegalArgumentException if some property of the element
        prevents it from being added to this collection
@throws IllegalStateException if the element cannot be added at this
        time due to insertion restrictions
*/
boolean add(Object e);
```

Quelltext 3.19: JavaDoc und Signatur von Collection.add(Object)

```
1  /*@ public behavior
2    @ signals_only UnsupportedOperationException, ClassCastException,
3          NullPointerException, IllegalArgumentException,
4          IllegalStateException;
5    @ assignable changeable;
6    @
7    @ also
8    @ public exceptional_behavior
9    @ requires !supportsNull;
10   @ requires e == null;
11   @ signals (NullPointerException e) (true);
12   @
13   @ also
14   @ public normal_behavior
15   @ requires e != null || supportsNull;
16   @ ensures contains(e);
17   @ assignable changeable;
18   @
19   @ also
20   @ public normal_behavior
21   @ requires e != null || supportsNull;
22   @ requires (contains(e) && supportsDuplicates) || !contains(e);
23   @ ensures collectionSize == \old(collectionSize) + 1;
24   @ ensures \result;
25   @ assignable changeable;
26   @
27   @ also
28   @ public normal_behavior
29   @ requires e != null || supportsNull;
30   @ requires contains(e) && !supportsDuplicates;
31   @ ensures collectionSize == \old(collectionSize);
32   @ ensures !\result;
33   @ assignable changeable;
34   @*/
35 boolean add(/*@nullable@*/ Object e);
```

Quelltext 3.20: Formale Spezifikation von Collection.add(Object)

https://dx.doi.org/10.24355/dbbs.084-201801311013

ist ⟨15, 21, 29⟩, da es sich sonst mit dem Fehlerfall der `NullPointerException` überlagern würde ⟨9 – 10⟩. Auch wird in jedem Fall das zuvor definierte Hilfsfeld `changeable` verwendet, um zu definieren, welche Felder sich jeweils ändern dürfen ⟨17, 25, 33⟩. Der dritte Spezifikationsfall der `add(Object)`-Methode ist der erste, der keinen Fehlerfall beschreiben soll. In diesem wird die Anforderung beschrieben, dass das Element nach dem Aufruf in der `Collection` enthalten sein muss {2}. Dies wird mit Hilfe der `contains`-Methode erreicht (siehe Abschnitt A.2.1), da diese Methode `pure` ist und damit als Query genutzt werden kann und genau diese Prüfung umsetzen soll ⟨16⟩. Der vierte und fünfte Spezifikationsfall soll jeweils das Resultat der Methode abbilden. Das Resultat soll genau dann `true` sein, wenn das Element noch nicht enthalten ist oder keine Duplikate erlaubt {3 – 5}. Dieser Fall wird im vierten Spezifikationsfall umgesetzt ⟨20 – 25⟩.

```
1 public boolean add(Object e) {
2    ensureCapacity(size + 1);  // Increments modCount!!
3    elementData[size++] = e;
4    return true;
5 }
```

Quelltext 3.21: Quelltext der JavaList.add(Object)

Im weiteren Verlauf dieser Arbeit wird noch auf mehrere Parameter von KeY eingegangen. Einer dieser Parameter bezieht sich auf das Verhalten im Zusammenhang mit `int`-Feldern (siehe Beschreibung 4.21). Dieser ermöglicht es zu steuern, ob KeY einen Überlauf ignoriert, prüft oder die Java-Semantiken für `int`-Felder verwendet. Die Methode lässt sich nur verifizieren, wenn ein Überlauf ignoriert wird, da KeY sonst einen Überlauf bei dem `size`-Feld bemerkt. Wenn man dies allerdings überprüft, stellt man fest, dass es dazu nicht kommen kann, da `ensureCapacity` ein Array anlegt, dass mindestens so groß ist wie der übergebene Parameter. Aus diesem Grund müsste der Aufruf von `ensureCapacity` bereits bei Werten, die keinen Überlauf verursachen, zu Problemen führen. Dieses Problem ließe sich lösen, wenn man einer Collection erlauben würde, dass sie eine maximale Größe haben darf.

In der Collection an sich spiegelt sich diese Problematik allerdings allgemein wieder. Die `toArray`-Methode soll alle Elemente einer Collection in einem Array zurückgeben (siehe Abschnitt A.2.2). Da bereits festgestellt wurde, dass mehr Elemente in einer Collection als in einem Array enthalten sein können, kann diese Spezifikation mit keiner Implementierung erfüllt werden. Um dieses Problem zu lösen, müsste die Anforderung an die `toArray`-Methode dahingehend angepasst werden, dass sie nicht alle Elemente enthalten muss, sofern dies technisch nicht möglich ist. Die beiden Probleme bestehen auch noch in der aktuellen Version des *Java SE* [Ora17].

3.3.6 Spezifikation von Constructor(int) der ArrayList

Der `ArrayList(int)`-Konstruktor soll eine neue Collection erzeugen, welche eine vordefinierte Kapazität hat (siehe Quelltext 3.22, Zeile 2; Kurzschreibweise {2}). Diese vordefinierte Kapazität wird über den Parameter `initialCapacity` übertragen {4}, sollte jedoch nicht negativ sein, da sonst eine `IllegalArgumentException` ausgelöst werden soll {5-6}.

https://dx.doi.org/10.24355/dbbs.084-201801311013

```
1  /**
2  Constructs an empty list with the specified initial capacity.
3
4  @param   initialCapacity   the initial capacity of the list
5  @exception IllegalArgumentException if the specified initial capacity
6             is negative
7  */
8  public ArrayList(int initialCapacity)
```

Quelltext 3.22: JavaDoc und Signatur von ArrayList(int)

```
1  /*@ public normal_behavior
2    @ requires initialCapacity >= 0;
3    @ ensures elementData.length == initialCapacity;
4    @ assignable elementData;
5    @ also
6    @ public exceptional_behavior
7    @ requires initialCapacity < 0;
8    @ signals_only IllegalArgumentException;
9    @ signals (IllegalArgumentException e) true;
10   @*/
11 public ArrayList(int initialCapacity)
```

Quelltext 3.23: Formale Spezifikation von ArrayList(int)

Die Übersetzung dieser Anforderung in zwei Spezifikationsfälle ist mit den Erfahrungen aus den vorherigen Verträgen simpel. Der erste Spezifikationsfall setzt voraus, dass `initialCapacity` positiv ist (siehe Quelltext 3.23, Zeile 2; Kurzschreibweise ⟨2⟩). In diesem Fall muss die Kapazität der `ArrayList` nach der Ausführung des Konstruktors dem Parameter entsprechen ⟨3⟩, demzufolge muss auch das Array geändert werden ⟨4⟩. Sollte der Wert negativ sein ⟨7⟩, soll eine `IllegalArgumentException` geworfen werden ⟨8-9⟩.

Interessant an dieser Stelle ist allerdings, die Verifikation der Spezifikationsfälle zu betrachten. Der erste Spezifikationsfall, der die positive `initialCapacity` betrachtet, lässt sich problemlos verifizieren. Der Spezifikationsfall, der den Fehlerfall beschreibt, lässt sich jedoch nicht verifizieren. Das Problem an dieser Stelle ist, dass im Falle einer Exception das `elementData`-Feld `null` ist, da vor der Initialisierung dieses Feldes die Exception geworfen wird (siehe Quelltext 3.25, Zeile 3-5). Dadurch wird allerdings die Invariante verletzt, dass dieses Feld immer ungleich `null` ist (siehe Abschnitt 3.3.2). Diese Invariante muss überprüft werden, da Invarianten unter anderem nach der Ausführung eines Konstruktors gültig sein müssen (siehe Beschreibung 2.10). In diesem Fall wird durch das Werfen der `Exception` verhindert, dass die Referenz zurück gegeben werden kann. Das fehlerhaft initialisierte Objekt kann also nicht genutzt werden. Grundsätzlich kann ein Objekt seine Referenz aber nach dem Aufruf des Super-Konstruktors weitergeben (siehe Abschnitt 3.4.4). Dadurch könnte auf ein solches, nicht korrekt initialisiertes Objekt zugegriffen werden. Die Forderung von KeY, dass die Invariante trotz der `Exception` nach der Ausführung des Konstruktors eingehalten werden muss, ist also grundsätzlich gerechtfertigt, ergibt aber in diesem Sonderfall keinen Sinn. Eine leicht abgewandelte

https://dx.doi.org/10.24355/dbbs.084-201801311013

Implementierung des Konstruktors, der die Invariante auch im Fall eines negativen Werts in `initialCapacity`-Parameters erfüllt, ohne das Verhalten im positiv Fall zu ändern, lässt sich jedoch mit nur einer Modifikation erstellen. Hierzu muss nur ein leeres Array in `elementData` vor der Prüfung des Parameters erstellt werden (siehe Quelltext 3.25, Zeile 3). Diese Implementierung kann problemlos mit beiden Spezifikationsfällen verifiziert werden.

```
1 public ArrayList(int initialCapacity) {
2     super();
3     if (initialCapacity < 0)
4         throw new IllegalArgumentException("Illegal Capacity: "+
5                                            initialCapacity);
6     this.elementData = new Object[initialCapacity];
7 }
```

Quelltext 3.24: Implementierung von ArrayList(int)

```
1 public ArrayList(int initialCapacity) {
2     super();
3     this.elementData = new Object[0];
4     if (initialCapacity < 0)
5         throw new IllegalArgumentException("Illegal Capacity: "+
6                                            initialCapacity);
7     this.elementData = new Object[initialCapacity];
8 }
```

Quelltext 3.25: Angepasste Implementierung von ArrayList(int)

3.4 Missverständnisse bei der Spezifizierung

In den vorherigen Abschnitten dieses Kapitels wurden verschiedene Verträge erstellt. Bei der Erstellung der Verträge kam es aus dem Blickpunkt eines Softwareentwicklers zu einige Missverständnisse, bei denen sich KeY anders verhalten hat, als erwartet. Dies ist oftmals darauf zurückzuführen, dass die Intuition zu einem Sprachkonstrukt von dem letztendlichen Verhalten abweicht. Alternativ waren es fehlende Bedingungen in den Verträgen, die nicht direkt aus der informellen Spezifikation folgten. Diese Missverständnisse sollen an dieser Stelle noch einmal separat dargestellt werden, um sie näher zu erläutern und dem Leser in einem ähnlichen Fall zu helfen. In diesem Abschnitt soll deshalb noch einmal das Verhalten von Array Instanzierungen im Zusammenhang mit Nullable (sieh Abschnitt 3.4.1), die Datentypen von Arrays (siehe Abschnitt 3.4.2) und das Vorkommen mehrerer `signals`-Schlüsselworte in einem Vertrag (siehe Abschnitt 3.4.3). Abschließend wird in Abschnitt 3.4.4 noch ein Missverständnis bei der fehlerhaften Ausführung eines Konstruktors und dazu gehörenden Invarianten vorgestellt.

3.4.1 Array Instanziierung - Nullable

Im Quelltext 3.26 können zwei verschiedene Methoden mit identischen Verträgen und identischer Implementierung gesehen werden. Beide Methoden sollen `Object`-Arrays der Länge `arrSize` erzeugen. Die erste Methode (Zeile 1 – 8; Kurzschreibweise ⟨1–8⟩) lässt sich hierbei nicht verifizieren, die zweite ⟨10–17⟩ hingegen schon.

https://dx.doi.org/10.24355/dbbs.084-201801311013

Der Unterschied liegt an dieser Stelle darin, dass das Rückgabeobjekt der zweiten Methode `nullable` ist ⟨15⟩. Dies wurde jedoch nicht erwartet, da das Ergebnis nicht null sein kann und `nullable` deshalb keine Auswirkung haben sollte. Obwohl die Implementierung in beiden Fällen verbietet, dass das Rückgabeobjekt `null` ist ⟨3, 12⟩, wirkt sich der Parameter jedoch auch auf die Elemente des Arrays aus (siehe Beschreibung 2.9 Nullable). Ohne das `nullable` müssen auch alle Elemente des Arrays ungleich `null` sein. Dies ist bei neu instanziierten Arrays jedoch nur für das Array der Länge 0 der Fall, da Felder eines Arrays mit `null` initialisiert werden, sofern der Datentyp des Arrays kein *primitiver Datentyp* ist.

```
1  /*@ public normal_behavior
2    @ requires arrSize >= 0;
3    @ ensures \result != null;
4    @ ensures \result.length == arrSize;
5    @*/
6  private static Object[] cArrayNotWorking(int arrSize){
7      return new Object[arrSize];
8  }
9
10 /*@ public normal_behavior
11   @ requires arrSize >= 0;
12   @ ensures \result != null;
13   @ ensures \result.length == arrSize;
14   @*/
15 private static /*@ nullable @*/ Object[] cArrayWorking(int arrSize){
16     return new Object[arrSize];
17 }
```

Quelltext 3.26: Minimalbeispiel zur Array-Instanziierung

3.4.2 Array Zugriffe - Datentypen

Ein Array hat immer einen Komponententyp [GJSB05, S. 287]. Dieser kann ein Array-, ein Referenz- oder ein Primitivtyp sein. Wenn es sich dabei um einen Referenz- oder Primitivtyp handelt, nennt man dies den Elementtyp.

Wenn einem Feld eines Arrays ein Objekt zugewiesen werden soll, dass weder eine Instanz des Elementtyps oder ein Subtyp dieses Typs ist, wird eine `ArrayStoreException` ausgelöst [GJSB05, S. 294f]. Diese Überprüfung wird auch in KeY durchgeführt. Ein Minimalbeispiel zu dieser Problematik kann in Quelltext 3.27 dargestellt werden. Die `set`-Methode sollte instinktiv verifizierbar sein, der Beweiser bleibt jedoch mit einem `arrayStoreValid(self.arr, o)`-Hinweis stehen. Das Problem ist jedoch, dass im ersten Moment nicht erkennbar ist, wieso man in einem `Object`-Array kein `Object` speichern können sollte. Die Ursache liegt, wie auch in Abschnitt 3.3.2 dargestellt, im Typen des Arrays. Das Array `arr` ist vom Typ `Object[]`. Folglich kann `arr` jedes Array zugewiesen werden, dessen Elementtyp ein Subtyp von `Object` ist [GJSB05, S. 289]. Die Folge ist, dass das Array als Elementtyp einen anderen Subtyp von `Object` haben kann, als der Parameter `o` als Typ hat. Das Problem kann gelöst werden, indem der Typ von `arr` mit Hilfe des `typeOf`-Schlüsselworts definiert wird.

https://dx.doi.org/10.24355/dbbs.084-201801311013

```
1  public class ArrayStoreValid{
2      /*@ invariant arr != null;
3        @ invariant arr.length == 1;
4        @*/
5      private transient /*@ nullable @*/ Object arr[];
6
7      /*@ public normal_behavior
8        @ ensures true;
9        @*/
10     public void set(Object o) {
11         arr[0] = o;
12     }
13 }
```

Quelltext 3.27: Minimalbeispiel zum Datentypen bei Array-Zugriffen

3.4.3 Das Signals-Keyword in mehreren Spezifikationsfällen

In Quelltext 3.28 sind zwei Spezifikationsfälle zu sehen, die nicht disjunkt sind. Wenn `original` `null` sein sollte und `newLength` negativ, sind die Vorbedingungen beider Spezifikationsfälle gültig. Folglich gilt sowohl `signals (NullPointerException e) true`, als auch `signals (NegativeArraySizeException e) true`, was widersprüchlich wirkt, da nur eine von beiden Exceptions auftreten kann. Es lassen sich dennoch beide Spezifikationsfälle verifizieren, da `signals` nicht verlangt, dass eine bestimmte Exception auftreten muss, sondern das Auftreten der entsprechenden Exception gestattet (siehe Beschreibung 2.17 Signals und Signals_only).

```
1  /*@ public exceptional_behavior
2    @ requires original == null;
3    @ signals (NullPointerException e) true;
4    @ also
5    @ public exceptional_behavior
6    @ requires newLength < 0;
7    @ signals (NegativeArraySizeException e) true;
8    @*/
9  public static /*@pure@*/ void method(/*@ nullable @*/ Object[]
       original, int newLength){
10     if(original == null)
11         throw new NullPointerException();
12     if(newLength < 0)
13         throw new NegativeArraySizeException();
14 }
```

Quelltext 3.28: Minimalbeispiel zur Verwendung zweier, nicht disjunkter, Exceptional Behavior-Spezifikationsfälle

3.4.4 Invarianten nach Fehlerfällen in Konstruktoren

Ein Konstruktor wird in Java verwendet, um Instanzen einer Klasse zu erzeugen [GJSB05, S. 240ff]. Sollte ein Konstruktor nicht als erstes mit dem Aufruf eines

https://dx.doi.org/10.24355/dbbs.084-201801311013

Konstruktors der jeweiligen Superklasse beginnen, wird implizit der parameterlose Konstruktor der Superklasse verwendet. Das Objekt wird bereits vor der Ausführung eines Konstruktors erzeugt und liegt diesem vor, um das neue Objekt zu initialisieren [GJSB05, S. 322f]. Das Werfen einer Exception in einem Konstruktor ist grundsätzlich wie für jede andere Methode definiert und führt immer zum Abbruch der Initialisierung des Objekts. In diesem Fall wird die Referenz auf die Instanz des Objekts nicht zurückgegeben. Es wirkt auf den ersten Blick widersprüchlich, dass ein Objekt, dessen Referenz nicht bekannt ist alle Invarianten erfüllen muss. Wenn eine Referenz auf dieses Objekt jedoch bereits in den Konstruktoren zur Verfügung steht, kann sie bereits anderweitig gespeichert werden (siehe Quelltext 3.29). Wie bereits in Kapitel Abschnitt 3.3.6 dargelegt, müssen deshalb auch für ein nicht korrekt initialisiertes Objekt alle Invarianten geprüft werden.

```
1  public class ExceptionalConstructor {
2      public static final List<ExceptionalConstructor> created = new
           ArrayList<>();
3      public boolean initialized;
4
5      public ExceptionalConstructor(boolean throwing) {
6          super();
7          created.add(this);
8          if (throwing) {
9              throw new RuntimeException();
10         }
11         initialized = true;
12     }
13 }
```

Quelltext 3.29: Konstruktor, der trotz Exception die Referenz speichert

3.5 Zusammenfassung

Die Spezifikation und anschließende Verifikation der oben genannten Verträge war nicht trivial, stellte sich jedoch als möglich heraus. Es konnte beispielsweise mit Hilfe von KeY gezeigt werden, dass die `ArrayList`-Implementierung sich in den meisten betrachteteten Methoden wie spezifiziert verhielt. Es ist aber aufgefallen, dass ein grundsätzliches Problem im `Collection`-Interface bei großen Objektmengen vorliegt.

Das größte Hindernis, dass im Rahmen der Verifikation jedoch auffiel, ist, dass sich vor allem das Einlesen der gesamten Java Platform API als nicht möglich herausstellte. Nicht nur Generics, sondern eine Vielzahl verschiedener Probleme stellten sich hierbei in den Weg. Die größten Startschwierigkeiten hingen mit *JavaRedux* zusammen, da die Fehlermeldung der `UncollatedReferenceException` nicht auf *JavaRedux* zurückführte und eine Vielzahl an statischen Konstanten in *JavaRedux* nicht existieren (siehe Abschnitt 3.1). Um dieses Problem zu umgehen, hätte entweder sehr viel Aufwand in das Anpassen des Bootklassenpfads fließen müssen, gefolgt von dem vollständigen Entfernen von *Generics*. Deshalb konnten nur Teile des eigentlichen Quelltexts in KeY eingelesen werden, wodurch der Quelltext stark modifiziert werden musste. In einem Entwicklungsprojekt hätte dies zur Folge, dass

https://dx.doi.org/10.24355/dbbs.084-201801311013

ein Zweig des Projekts erzeugt werden müsste, der nur den zu verifizierenden Quelltext enthält. Selbst in diesem gab es allerdings noch einzelne Java-Schlüsselworte, wie zum Beispiel `.class`, mit denen KeY nicht arbeiten konnte.

Bei der Spezifikation der `Arrays.copyOf(Object[], int)`-Methode (siehe Abschnitt 3.2) ist bereits aufgefallen, dass auch ein wichtiger Bestandteil von Java derzeit nicht unterstützt wird: *Reflections*. Ein Problem bei der Spezifikation der *Reflections*-Funktionalität ist die hohe Flexibilität, die diese zur Verfügung stellt. Dadurch lassen sich zwar keine Methoden verifizieren, die *Reflections* benutzen, jedoch können Spezifikationen angegeben werden. Mit Hilfe dieser unverifizierten Spezifikationen können wiederum Methoden verifiziert werden, die diese nur spezifizierten Methoden aufrufen. Ihr Verhalten ist dann allerdings nur unter der Annahme verifiziert, dass sich die gerufene Methode wie spezifiziert verhält.

Im Rahmen der Verifikation von `ArrayList.ensureCapacity(int)` wurde bemerkt, dass KeY bei der Erstellung eines Arrays nicht überprüft, ob das Array zu groß ist. Dies ist darauf zurückzuführen, dass die maximale Größe eines Arrays in Java nicht spezifiziert ist. Die virtuellen Maschinen können diese Werte deshalb festlegen und es gibt keinen einheitlichen Wert auf den KeY zurückgreifen kann. Bis mit einer zukünftigen Java-Version ein solcher einheitlicher Wert festlegt werden könnte, würde zum einen lange Dauern und würde auch die Entwickler der virtuellen Maschinen in ihrer Freiheit einschränken. Deshalb wäre eine Festlegung in KeY sinnvoller. Da der Wert sich zwischen einzelnen virtuellen Maschinen unterscheiden kann, sollte der Anwender diese Grenze dynamisch festlegen können.

Dass der Funktionsumfang von KeY nicht alle Java-Möglichkeiten unterstützt, war im Vorfeld der Arbeit bekannt. Fließkommazahlen und Nebenläufigkeit wurden deshalb im Vorfeld dieser Arbeit bereits ausgeschlossen und auch *Reflections* kann man ebenfalls zu dieser Liste hinzufügen. Wenn man sich dessen bewusst ist, kann KeY dennoch theoretisch für das Verifizieren der restlichen Methoden genutzt werden. Dass KeY den Quelltext des Projekts durch nicht unterstützte Sprachkonstrukte aus moderneren Java-Versionen oder durch *JavaRedux* nicht einlesen kann, erhöht allerdings auch für theoretisch verifizierbare Methoden massiv den Aufwand. Um dieses Problem zu umgehen, wäre es empfehlenswert, KeY auf einen moderneren Java-Parser umzustellen und *JavaRedux* zu aktualisieren. Da die Konstanten einzelner Java-Klassen sich jedoch abhängig von der gewählten Java-Implementierung und Version unterscheiden können, wäre es ideal, wenn *JavaRedux* mit Hilfe der Benutzeroberfläche ausgetauscht werden könnte. Als alternative Lösung für das Problem mit dem Einlesen könnten auch Methoden, die für die Verifikation nicht benötigt werden, nicht mit eingelesen werden.

https://dx.doi.org/10.24355/dbbs.084-201801311013

4. Parameter und Optionen von KeY

Im vorherigen Abschnitt dieser Arbeit wurden Verträge für Methoden aus der Java Platform API des OpenJDK erstellt. Das jeweilige Ergebnis der Verifikation wurde bereits vorgestellt, jedoch noch nicht, wie diese genau durchgeführt wurden. Wie in den Grundlagen bereits erwähnt (siehe Abschnitt 2.5), bietet KeY eine Vielzahl verschiedener Parameter und Optionen, die eine entscheidende Rolle für den Aufwand und die Schließbarkeit eines Beweises haben, die Auswahl der richtigen Konfiguration für einen Beweis ist jedoch nicht trivial. In diesem Abschnitt werden deshalb für die Optionen Hypothesen über die Auswirkungen der Parameter auf den Aufwand der automatischen Verifikation mit KeY aufgestellt. Mit den Auswirkungen sind hierbei sowohl Auswirkungen auf die Möglichkeit, dass KeY den Beweis automatisch schließt (im weiteren Beweisbarkeit genannt), als auch Auswirkungen auf den Beweisaufwand gemeint. Diese Hypothesen sollen im nächsten Kapitel der Arbeit mit den Resultaten der Verifikation mit KeY empirisch überprüft werden.

Um die Hypothesen aufzustellen, werden zunächst die jeweiligen Parameter von KeY erläutert. Bei jedem dieser Parameter stehen zwei oder mehr Optionen zur Wahl. Diese werden zunächst mit der Kurzschreibweise $\mathit{Parameter} \in \{\mathit{Option}_1, ..., \underline{\mathit{Option}_i}, ..., \mathit{Option}_n\}$ eingeführt, wobei Option_i die Standardauswahl ist. Die Optionen müssen vor der Verifikation festgelegt werden, da wir im Rahmen dieser Arbeit vollautomatische Verifikationen betrachten wollen. Wenn KeY nicht vollautomatisch genutzt werden soll, können diese auch im Verlauf der Verifikation geändert werden, was sich positiv auf den Aufwand und die Beweisbarkeit auswirken kann. Dieses Vorgehen soll allerdings nicht im Rahmen dieser Arbeit betrachtet werden, da dies den Umfang dieser Arbeit übersteigen würde. Durch den Fokus der Arbeit wird auch an dieser Stelle auf die einzelnen Parameter nur aus Anwendersicht eingegangen. Hierbei wird jedoch nicht zu jedem Parameter eine Hypothese aufgestellt, da es bei manchen Parametern nicht sinnvoll erscheint. Sollte zu einem Parameter keine Hypothese aufgestellt werden, wird stattdessen kurz dargelegt, weshalb davon abgesehen wird.

https://dx.doi.org/10.24355/dbbs.084-201801311013

In diesem Kapitel sollen zunächst die einzelnen Optionen vorgestellt und im Rahmen dieser Vorstellung die Hypothesen erarbeitet werden. Zunächst sollen die Optionen der Strategie zur Beweissuche (Abschnitt 4.1) behandelt werden, um anschließend auf die Taclet-Optionen (Abschnitt 4.2) und abschließend auf die allgemeinen Optionen (Abschnitt 4.3) einzugehen.

4.1 Optionen zur Strategie der Beweissuche

Mit Hilfe der Optionen zur Strategie der Beweissuche, oder auch Proof-Search-Strategy-Optionen, lässt sich steuern, wie der automatische Beweiser vorgehen soll, um den jeweils nächsten Schritt im Beweis auszuwählen. Das Spektrum der von den Optionen beeinflussten Bereiche reicht hierbei von der maximalen Anzahl der Beweisschritte über die Behandlung von Methodenaufrufen bis hin zu dem Behandeln von Bedingungen und den damit auftretenden Abzweigungen im Beweis. Eine graphische Darstellung der Optionen zur Strategie der Beweissuche kann in Abbildung 4.1 gesehen werden. Diese umfasst jedoch nicht die *Max. Rules Application*, da diese als einzige keine Mehfachwahl ermöglicht, sondern eine natürliche Zahl ist. Als Darstellungsform für die Parameter und Optionen kommt ein Featurediagramm zum Einsatz [MTS+17, S. 43].

In diesem Kapitel werden alle verfügbaren Parameter vorgestellt. Sollte zu einem Parameter eine Hypothese aufgestellt werden, erfolgt dies für eine bessere Lesbarkeit direkt im Anschluss an die Beschreibung des jeweiligen Parameters. Als Quelle dient, wenn nicht anders angegeben, der KeY Quickguide [BHS13, S.28ff] und der jeweilige Tooltip zu der Option [Key].

Beschreibung 4.1 – Max. Rules Applications: *Max. Rules Applications* $\in$ {1, ..., *10.000*, ..., *1.000.000*}. Die maximale Anzahl der Beweisschritte bietet dem Anwender die Möglichkeit zu steuern, wie viele Schritte der automatische Beweiser ausführen darf, um den Beweis zu schließen, ehe er manuelle Eingriffe des Benutzers verlangt. Der Anwender kann hierbei einen beliebigen Wert zwischen 1 und 10^6 einstellen.

Da die Beweise - wenn möglich - vollautomatisch geschlossen werden sollen, wird der Wert im Folgenden immer auf das Maximum gesetzt. Für einen interaktiven Beweis können allerdings auch kleinere Werte sinnvoll sein, da der Anwender von KeY nur direkten Einfluss auf den Beweis nehmen kann, wenn der Beweiser stehen bleibt.

Beschreibung 4.2 – Stop At: *Stop At* $\in$ {*Default*, *Unclosable*}. Der Unterschied zwischen den beiden Optionen betrifft das Verhalten eines aufgespaltenen Beweises. Sollte der Anwender *Unclosable* gewählt haben, wird der automatische Beweiser die Arbeit einstellen, wenn ein Zweig des Beweises nicht automatisch geschlossen werden kann. Dies ist der Fall, wenn automatisch keine weiteren Regeln angewendet werden können. Wenn die Option *Default* gewählt wurde, wird der Beweiser weiter versuchen, die anderen Zweige des Beweises zu schließen, auch wenn ein Zweig vorher nicht geschlossen werden konnte.

https://dx.doi.org/10.24355/dbbs.084-201801311013

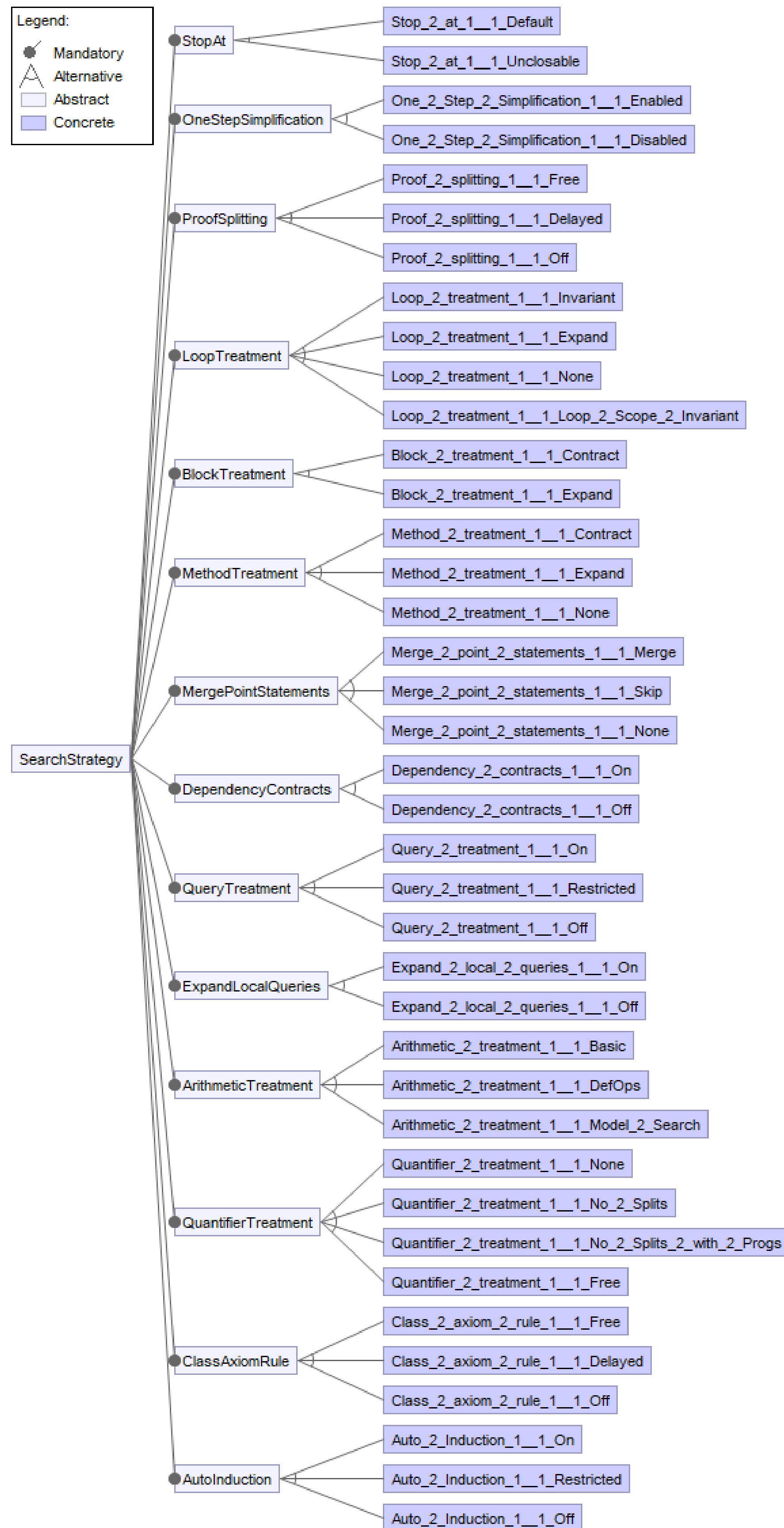

Abbildung 4.1: Darstellung der Optionen zur Strategie der Beweissuche als Feature Diagramm

https://dx.doi.org/10.24355/dbbs.084-201801311013

Wie aus Beschreibung 4.2 hervorgeht, soll *Stop At* nur Auswirkungen auf offene Beweisäste haben. Deshalb dürfte dieser Parameter keine Auswirkung auf den Aufwand eines automatisch schließbaren Beweises haben.

Hypothese 1 – Stop At - Beweisbarkeit: Der Parameter *Stop At* hat keinen Einfluss auf die Beweisbarkeit in KeY.

Hypothese 2 – Stop At - Beweisaufwand: Der Parameter *Stop At* hat keinen Einfluss auf den Aufwand automatisch schließbarer Beweise.

Beschreibung 4.3 – One Step Simplification: *One Step Simplification* ∈ {*Enabled*, *Disabled*}. Die Folge der Aktivierung dieses Parameters ist das Zusammenfassen von verschiedenen Taclets aus den Regelsätzen *concrete* und *simplify*. Das bedeutet, dass vom Beweiser statt mehreren Einzelschritten ein Schritt gemacht wird, der mehrere Regeln umfassen kann. Dies steigert allgemein die Performance des Beweisers, kann die Reihenfolge der Regelanwendung jedoch beeinflussen [ABB+16, S. 130].

Da das KeY-Buch [ABB+16] das Aktivieren dieses Parameters bereits für die Steigerung der Performance angibt, wird diese Hypothese übernommen. Außerdem wird die Vermutung aufgestellt, dass die Beweisbarkeit durch diesen Parameter nicht beeinflusst wird, da er nur mehrere Einzelschritte in einem Schritt zusammenfasst.

Hypothese 3 – One Step Simplification (Beweisbarkeit): Ein Beweis, lässt sich genau dann mit der Option *Enabled* für den *One Step Simplification*-Parameter schließen, wenn er sich mit *Disabled* schließen lässt.

Hypothese 4 – One Step Simplification (Beweisaufwand): Ein Beweis mit der Option *Enabled* für den *One Step Simplification*-Parameter einen höchstens so großen Beweisaufwand wie mit *Disabled*.

Beschreibung 4.4 – Proof Splitting: *Proof Splitting* ∈ {*Free*, *Delayed*, *Off*}. Die Option *Free* erlaubt es dem automatischen Beweiser ohne jegliche Einschränkungen den Beweis einer Formel aufzuspalten. Bei *Delayed* wird stets versucht, erst andere Regeln anzuwenden und damit das Aufspalten so lange wie möglich herauszuzögern. Das Auswählen von *Off* hat zur Folge, dass Beweise nicht aufgespaltet werden können. Dies gilt jedoch nur für Formeln, nicht für Programme.

Proof Splitting dürfte laut der Beschreibung 4.4 Proof Splitting keine Auswirkung auf Programme an sich haben, jedoch auf Formeln. Da die Programme allerdings in eine *JavaDL*-Formel übersetzt werden, kann sich die Wahl der Option dieses Parameters dennoch auf die Beweise von Java-Quelltext auswirken. Wenn ein Beweis aufgespaltet wird, so bilden sich mehrere Äste, die alle bewiesen werden müssen. Alle Schritte, die für alle entstehenden Äste identisch sind, müssen jedoch auch für jeden Ast einzeln behandelt werden, sofern diese erst nach dem Aufspalten beachtet werden. Wenn diese jedoch vorgezogen werden können, müssen sie nur einmal für alle Äste betrachtet werden. Aus diesem Grund wird vermutet, dass ein Verzögern des Aufspaltens sich positiv auf den Beweisaufwand auswirken müsste. Es könnte jedoch

https://dx.doi.org/10.24355/dbbs.084-201801311013

auch dazu führen, dass Annahmen für eigentlich zwei Äste getroffen werden, die nur auf einen Ast zutreffen. Ein Beispiel hierbei wäre die Instanziierung der Variable bei einem Quantifizierer. Deshalb wird auch die Hypothese aufgestellt, dass sich dies negativ auf das Schließen von Beweisen auswirkt.

Hypothese 5 – Proof Splitting (Beweisbarkeit): Ein Beweis, der sich mit der Option *Delayed* für den *Proof Splitting*-Parameter schließen lässt, lässt sich auch mit *Free* schließen.

Hypothese 6 – Proof Splitting (Beweisaufwand): Ein Beweis hat mit der Option *Delayed* für den *Proof Splitting*-Parameter stets einen höchstens so großen Beweisaufwand wie mit *Free*.

Sollte sich ein Beweis mit der Option *Off* automatisch schließen lassen, so enthält dieser keine Aufspaltung in der Formel, sonst hätte der automatische Beweiser an dieser Stelle stoppen müssen. Die Option müsste also von der Beweisbarkeit her schlechter als *Delayed* und *Free* sein, da diese bei Formeln ohne Aufspaltung keine Auswirkung haben. Aus diesem Grund kann es bei diesen Beweisen auch keine Auswirkung auf den Beweisaufwand geben.

Hypothese 7 – Proof Splitting (Beweisbarkeit Off): Ein Beweis, der sich mit der Option *Off* für die *Proof Splitting*-Option schließen lässt, lässt sich auch mit *Free* und *Delayed* schließen.

Hypothese 8 – Proof Splitting (Beweisaufwand Off): Ein Beweis, der sich mit der Option *Off* für die *Proof Splitting*-Option schließen lässt, hat mit den Optionen *Delayed* und *Free* einen mindestens genau so großen Beweisaufwand.

Beschreibung 4.5 – Loop Treatment: *Loop Treatment* ∈ {*Loop Scope Invariant*, <u>*Invariant*</u>, *Expand*, *None*}. Sowohl For-, als auch Do-While-Schleifen werden zunächst in While-Schleifen umgewandelt, bevor der Beweiser sie näher betrachtet und die gewählte Option eine Auswirkung hat [ABB+16, S. 76]. Für die Arbeit mit While-Schleifen stehen in KeY die Optionen *Invariant*, *Loop Scope Invariant*, *Expand* und *None* zur Auswahl. Die Option *None* hat zur Folge, dass KeY Schleifen nicht weiter behandelt. Das bedeutet, dass der automatische Beweiser einen Zweig nicht schließen kann, sofern er auf eine Schleife stößt. Sollte *Expand* gewählt werden, versucht das System mit wiederholtem Abrollen der Schleifen zu arbeiten. Deshalb ist diese Option nur praktikabel, wenn die Anzahl der Schleifeniterationen eine kleine obere Grenze hat.
In allen anderen Fällen steht die Option zur Verfügung, Schleifeninvarianten oder auch Induktion zu nutzen [ABB+16, S. 76]. Dies wird durch die Wahlmöglichkeit *Invariant* abgebildet, welche für eine maximale Automatisierung genutzt werden sollte [ABB+16, S. 553]. Für diese Option muss der Anwender Schleifenspezifikationen zu allen Schleifen angeben. Diese ist ein Dreitupel, bestehend aus einer Menge von Schleifeninvarianten (siehe Beschreibung 2.11), einer Menge LocSets (vergleichbar zu dem in Beschreibung 2.16) und einer Schleifenvariantenfunktion (siehe Beschreibung 2.12). Die Schleifenvariantenfunktion wird im KeY-Buch als Terminierungszeuge bezeichnet. Eine Schleifeninvariante muss vor der Ausführung der

https://dx.doi.org/10.24355/dbbs.084-201801311013

Schleife und nach jeder Schleifeniteration gelten [ABB+16, 101f]. Bei dem Terminierungszeugen handelt es sich entweder um die Aussage `Partial` [ABB+16, S. 271], welche bedeutet, dass die Schleife eventuell nie terminiert, oder um einen nicht negativen Ausdruck, der mit jedem Schleifendurchlauf kleiner wird [ABB+16, S. 299]. Selbst wenn Terminierungen, wie eine Exception, ein `break` oder ein `return` in einer Schleife vorkommen, können Invarianten genutzt werden, da dies von der Invariantenregel mit abgebildet wird. Sollte einer dieser Fälle auftreten, so wird der jeweilige Befehl nicht im Kontext der Schleife, sondern im Originalkontext ausgeführt [ABB+16, S. 104ff].
Seit KeY 2.7 gibt es noch eine vierte Option für *Loop Treatment*: *Loop Scope Invariant*. Diese bietet grundsätzlich die gleiche Funktionalität wie *Invariant*, fasst aber die Prüfung der Gültigkeit der Invariante während und nach der Ausführung der Schleife zusammen.

Der Einsatz von *None* für *Loop Treatment* ergibt für vollautomatische Beweise keinen Sinn, sofern Schleifen im Quelltext vorhanden sind. Auch *Expand* ist laut Beschreibung nur in Sonderfällen sinnvoll (siehe Beschreibung 4.5 Loop Treatment). Da die Funktionen der Java Platform API für viele Einsatzzwecke genutzt werden können, ist der Quelltext von dem Einsatz von Variablen geprägt. Das hat zur Folge, dass bei einer Schleife meist im Vorfeld nicht bekannt ist, wie häufig sie durchlaufen wird. Wenn also eine Schleife im Quelltext vorhanden ist, wird auch *Expand* nicht zielführend sein. Außerdem muss eine manuelle Anpassung durch das Hinzufügen einer Schleifenspezifikation erfolgen, damit *Invariant* wirklich genutzt werden kann. Ein Anwender wird für diesen Zusatzaufwand einen Grund gesehen haben, es ist also davon auszugehen, dass dies einen Vorteil mit sich bringt. Eine Untersuchung, welcher der beiden Operatoren mehr Vorteile mit sich bringt, scheint deshalb nicht sinnvoll.

Interessant ist aber der Vergleich von *Invariant* und *Loop Scope Invariant*. Da *Loop Scope Invariant* zwei Prüfungen zu einer zusammenfasst, ist davon auszugehen, dass diese Option einen geringeren Beweisaufwand als *Invariant* zur Folge hat.

Hypothese 9 – Loop Treatment (Beweisbarkeit): Jeder Beweis der sich mit der Option *Loop Treatment: Invariant* schließen lässt, lässt sich auch mit *Loop Treatment: Loop Scope Invariant* schließen.

Hypothese 10 – Loop Treatment (Beweisaufwand): Ein Beweis, der sich mit *Loop Treatment: Invariant* und *Loop Treatment: Loop Scope Invariant* schließen lässt, hat bei Verwendung von *Loop Scope Invariant* einen höchstens so großen Beweisaufwand wie mit *Invariant*.

Beschreibung 4.6 – Block Treatment: *Block Treatment* $\in$ {*Contract*, *Expand*}. Bei einem Block handelt es sich entweder um einen markierten, einen unmarkierten oder einen Try-Catch Block [ABB+16, S. 520]. Ein markierter Block ist ein unmarkierter Block mit einer Sprungmarke. Diese kann beispielsweise in *break*-Anweisungen genutzt werden, um auf ein spezifiziertes Level zu springen. Als Option für die Arbeit mit Blöcken stehen *Contract* und *Expand* zur Auswahl. Wenn *Expand* gewählt wurde, wird der Block durch die in ihm enthaltenen Befehle ersetzt, *Contract* kann stattdessen einen Kontrakt einsetzen. Hierzu wird die Gegebenheit der

https://dx.doi.org/10.24355/dbbs.084-201801311013

Vorbedingung und die Korrektheit des Blockkontrakts bewiesen. Anschließend wird die Nachbedingung des Kontrakts in den Beweis aufgenommen [ABB+16, S. 466].

Block Treatment umfasst demnach eine Funktion, bei der statt der Implementierung eines Blocks eine Spezifikation genutzt wird, welche im gleichen Zug auch verifiziert wird. Die *Contract*-Option dieses Parameters ist, wie auch *Invariant* des *Loop Treatment*-Parameters, mit einem Zusatzaufwand verbunden, da die Blockkontrakte zusätzlich erstellt werden müssen. Deshalb ist es auch bei diesem Parameter nicht sinnvoll, eine Hypothese aufzustellen. Für die Verifikationen im Rahmen dieser Arbeit wird *Contract* verwendet, da sich diese Option wie *Expand* verhält, wenn kein Vertrag angegeben ist. Sollte ein Vertrag definiert sein, würde dieser auch genutzt werden.

Beschreibung 4.7 – Method Treatment: *Method Treatment* $\in$ {*Contract*, *Expand*, *None*}. Der Parameter *Method Treatment* steuert das Verhalten bei einem Methodenaufruf. *None* hat, ähnlich wie bei *Loop Treatment*, die Folge, dass KeY den Methodenaufruf nicht automatisch bearbeitet. *Expand* sorgt dafür, dass der Aufruf der Methode durch die Implementierung ersetzt wird [ABB+16, S. 78]. Dieses Vorgehen wird im weiteren als *Inlining* bezeichnet. Alternativ kann die *Contract*-Option gewählt werden. Bei dieser wird der Aufruf nicht durch die Implementierung der Methode, sondern durch den Vertrag eben jener ersetzt. Dabei wird der Vertrag nicht validiert, sondern die Korrektheit als gegeben angesehen [ABB+16, S. 99]. Im Weiteren wird diese Behandlung von Methodenaufrufen als *Contracting* bezeichnet. Sollte eine Methode keinen Vertrag haben, wird auch bei der *Contract*-Option der Aufruf durch die Implementierung ersetzt [ABB+16, S. 529].

Method Treatment bietet eine Option, um Methodenaufrufe zu steuern. Durch den Einsatz von *Expand* kann es dazu kommen, dass jeder Methodenaufruf weitere Methoden einbindet. Auch sind einzelne Methoden `native`, werden also von der jeweiligen Java Virtual Machine direkt implementiert. Für diese liegt keine Implementierung in Java vor, weshalb diese auch nicht eingebunden werden kann. Bei der Verwendung von *Contract* kann dennoch ein Kontrakt für diese Methoden hinterlegt werden, auch wenn dieser nicht verifiziert werden kann. Auch in anderen Fällen, liegt nicht immer eine Implementierung vor. Es kann beispielsweise gegen ein Interface verifiziert werden. Der Einsatz von *Expand* ist aus diesen Gründen nicht immer möglich. Das KeY-Buch weist auch auf Probleme bezüglich der Beweisgröße beim Einsatz *Expand* hin [ABB+16, S. 530]. Dies ist jedoch vom Spezifikationsaufwand wesentlich aufwendiger, da für die jeweiligen Methoden Kontrakte angegeben werden müssen. Ist dies nicht der Fall, verhält sich *Contract* jedoch wie *Expand*, weshalb im Rahmen dieser Arbeit nur *Contract* verwendet wird.

Beschreibung 4.8 – Merge Point Statements: *Merge Point Statements* $\in$ {*Merge*, *Skip*, *None*}. Die Verwendung von *Merge Point Statements* erlaubt es KeY, verschiedene Beweisäste wieder zusammen zu fassen [SHB16, S. 57]. Hierbei stehen zwei verschiedene Operationen zur Auswahl: Das *If-Then-Else (ITE)* und das *abstraction based (PRED)* merging [SHB16, S. 65]. Welches der beiden Mergings genutzt wird, kann mit Hilfe einer JML-Spezifikation gesteuert werden [SHB16, S. 71]. Allgemein kann ein Merge nach einem Java-Block durchgeführt werden (siehe Beschreibung 4.6), welcher mit einer bestimmten Spezifikation annotiert ist:

https://dx.doi.org/10.24355/dbbs.084-201801311013

`merge_proc <Operation>`. Mit Hilfe der Option *Merge* werden diese Merge-Befehle ausgeführt, *Skip* ignoriert sie und *None* führt auch bei *Merge Point Statements* dazu, dass der automatische Beweiser stoppt.

Das eine Analyse der Auswirkungen von *Merge Point Statements* bereits in der Arbeit durchgeführt wurde, in der sie vorgestellt wurden [SHB16], wird an dieser Stelle von dem Aufstellen einer weiteren Hypothese abgesehen. Es konnte im Rahmen jener Arbeit gezeigt werden, dass sich der Beweisaufwand durch den Einsatz von *Merge Point Statements* drastisch senken lässt, jedoch gab es auch Gegenbeispiele, bei denen der Aufwand stieg. Auch die Beweisbarkeit wurde im Rahmen jener Arbeit betrachtet und es konnte festgestellt werden, dass sich vor allem der Einsatz des *abstraction based*-Mergings negativ auf die Beweisbarkeit auswirken kann.

Beschreibung 4.9 – Dependency Contracts: *Dependency Contracts* ∈ {*On*, *Off*}. Es wird gesteuert, ob die *accessible* Klausel von JML genutzt werden kann. Diese Klausel wird mit in den Methodenverträgen angegeben. Sie gibt an, welche bereits existierenden Felder von einer Methode oder einem Block gelesen werden dürfen. Damit verhält sie sich ähnlich wie die `assignable` Klausel, behandelt jedoch Lese- und keine Schreibzugriffe. [LPC+13, S. 84]. Mit Hilfe dieser Information ist bekannt, auf welchen Feldern das Ergebnis einer Methode nicht basieren kann. Es kann nicht genau gesagt werden, auf welchen Feldern das Ergebnis basiert, da es ja auch auf einer beliebigen Teilmenge der definierten Menge basieren kann [ABB+16, S. 279]. *Off* lässt KeY diese Klauseln ignorieren.

Da in den derzeitig erstellten Verträgen die `accessible` Klausel nicht verwendet wird, dürfte auch das Aktivieren von *Dependency contracts* keine Auswirkung auf den Aufwand des Beweises haben.

Hypothese 11 – Dependency Contracts (Beweisbarkeit ohne Accessible-Klauseln): Wenn ein Spezifikationsfall keine `accessible`-Klauseln angibt, hat die *Dependency Contract*-Option keine Auswirkung auf die Beweisbarkeit dieses Spezifikationsfalls.

Hypothese 12 – Dependency Contracts (Beweisaufwand ohne Accessible-Klauseln): Wenn ein Spezifikationsfall keine `accessible`-Klauseln angibt, hat die *Dependency Contract*-Option keine Auswirkung auf den Beweisaufwand dieses Spezifikationsfalls.

Beschreibung 4.10 – Query Treatment: *Query Treatment* ∈ {*On*, *Restricted*, *Off*}. Bei einer Query Methode (kurz Query), handelt es sich um eine seiteneffektfreie Methode, die in Spezifikationen aufgerufen werden kann [ABB+16, S. 405f]. *Off* stellt die Funktionalität zur Verfügung, dass Queries nicht weiter bearbeitet werden. Wenn *On* gewählt wurde, werden Queries bearbeitet. Was mit ihnen geschieht, hängt von dem Parameter *Method Treatment* ab. Sollte bei diesem die Option *Inlining* gewählt sein, wird die Query durch ihre Implementierung ersetzt. Bei *Contracting* wird, wie auch bei einem normalen Methodenaufruf, der Vertrag eingesetzt. *Restricted* stellt grundsätzlich die gleiche Funktionalität zur Verfügung, legt aber unter gewissen Bedingungen Einschränkungen fest. Dies hat Auswirkungen auf mehrere Faktoren, unter anderem auf die Reihenfolge, in der Queries ausgewertet werden

https://dx.doi.org/10.24355/dbbs.084-201801311013

und auf die Unterdrückung von Rekursionen. Sollten mehrere Queries auftreten, so wird stets die älteste Query zunächst ausgewertet. Sollte das Auswerten einer Query dazu führen, dass eine identische Query nochmals auftritt, wird sie nicht erneut ausgewertet. Dies verhindert eine direkte Rekursion. Auch werden die Parameter der Query betrachtet, um Rekursionen zu erkennen. Hierbei werden obere und untere Grenzen für Literale genutzt.

Sollte ein Vertrag keine Query enthalten, sollte der gewählte Parameter keine Rolle spielen.

Hypothese 13 – Query Treatment (Beweisbarkeit; ohne Query): Wenn ein Vertrag keine Query enthält, spielt die verwendete *Query Treatment*-Option keine Rolle im Bezug auf die Beweisbarkeit.

Hypothese 14 – Query Treatment (Beweisaufwand; ohne Query): Wenn ein Vertrag keine Query enthält, spielt die verwendete *Query Treatment*-Option keine Rolle im Bezug auf den Beweisaufwand.

Wenn eine Query enthalten ist, steigt in der Reihenfolge *Off* → *Restricted* → *On* der Umfang der Funktionalität. Die Funktionen bauen in diesem Fall aufeinander auf. Bei *Off* bleibt der Beweiser stehen, sofern die Query benötigt werden würde. An dieser Stelle kann es nach wie vor zum Schließen des Beweises kommen, wenn der Beweis mit Hilfe einer Gleichheit geschlossen werden kann. Ein Beispiel hierfür kann in Quelltext 4.1 gesehen werden. Die Methode `i` lässt sich mit Hilfe von *Method Treatment Contract* und *Query Treatment Off* verifizieren, da das Ergebnis der Methode nach dem Einsetzen des Vertrages von `complexMethod` `otherComplexMethod(j)` entspricht, was auch im Vertrag von `i` definiert ist. Das Taclet `replace_known_left`, welches diese Funktionalität zur Verfügung stellt, ist jedoch unabhänging von der gewählten *Query Treatment* und wird auch bei *Restricted* und *On* eingesetzt. Es wird deshalb für das Aufstellen der Hypothesen angenommen, dass dieses Szenario auch mit den anderen beiden Optionen identisch verifiziert werden würde. Da *Restricted* grundsätzlich die gleiche Funktionalität wie *On* zur Verfügung stellt, bei vermuteten Rekursionen jedoch abbricht, müsste sich jede mit *Restricted* verifizierbare Methode auch mit *On* verifizieren lassen. Sei $\{k\}$ die Menge der Beweise, die sich mit der *Query Treatment* Option k schließen lässt, dann müsste nach dieser Argumentation $\{Off\} \subseteq \{Restricted\} \subseteq \{On\}$ gelten.

Hypothese 15 – Query Treatment (Beweisbarkeit): Sollte sich ein Beweis mit der Option *Off* schließen, dann lässt er sich auch mit *Restricted* schließen. Wenn sich ein Beweis mit der Option *Restricted* schließen lässt, lässt er sich auch mit *On* schließen.

Auf Grund des Einflusses von *Restricted* auf die Reihenfolge der Auswertungen, wird vermutet, dass es Spezifikationsfälle gibt, bei denen sich eine Methode mit *Restricted* schneller verifizieren lässt und andere, bei denen sich mit *On* ein geringerer Aufwand erreichen lässt.

https://dx.doi.org/10.24355/dbbs.084-201801311013

```
1  public abstract /*@pure@*/ int otherComplexMethod(int a);
2
3  /*@ public normal_behavior
4    @ ensures \result == otherComplexMethod(a);
5    @*/
6  public abstract /*@pure@*/ int complexMethod(int a);
7
8  /*@public normal_behavior
9    @ ensures \result == otherComplexMethod(j);
10   @ */
11 public int i(int j){
12     return complexMethod(j);
13 }
```

Quelltext 4.1: Beispiel für eine mit Query Treatment Off verifizierbare Methode

Hypothese 16 – Query Treatment (Beweisaufwand): Es gibt Beweise, bei denen *Restricted* einen geringeren Beweisaufwand verursacht als *On* und Beweise, bei denen *On* einen geringeren Beweisaufwand verursacht als *Restricted*.

Beschreibung 4.11 – Expand Local Queries: *Expand Local Queries* $\in$ {*<u>On</u>*, *Off*}. Der Parameter ermöglicht es den *Query Treatment*-Parameter für lokale Queries zu überschreiben. Eine lokale Query ist eine Methode, die auf `self` oder einem seiner Elternobjekte aufgerufen wird. Alternativ wird sie als lokal angesehen, wenn der Typ ihres Resultats bekannt ist. Dieser Parameter hängt auf den ersten Blick eng mit dem *Query Treatment*-Parameter zusammen, ist aber unabhängig von dieser. Sollte *On* für *Expand Local Queries* gewählt werden, wird für alle lokalen Queries der *Query Treatment*- und der *Method Treatment*-Parameter ignoriert und für die Query wird *Expand* verwendet. *Off* deaktiviert diese Funktion. In diesem Fall verhalten sich die lokalen Queries wie normale Queries.

Expand Local Queries kann dazu genutzt werden, um bei einzelnen Queries das Einsetzen der Implementierung zu erzwingen. An dieser Stelle kann dieselbe Argumentation wie bei *Method Treatment* genutzt werden, um zu begründen, weshalb das Verwenden des Kontrakts oftmals schneller ist. Es wird an dieser Stelle die Annahme getroffen, dass es bei allen Queries im Rahmen dieser Arbeit schneller ist, *Contracting* zu nutzen.

Hypothese 17 – Expand local queries (Beweisaufwand): Sollte sich ein Beweis mit der Option *Off* und der Option *On* schließen lassen, so ist der Beweisaufwand bei *On* mindestens so groß wie bei *Off*.

Sollte eine Methode für einen Kontrakt verifizierbar sein, so muss dieser sich aus dem Quelltext des Vertrags schlussfolgern lassen. Die Implementierung muss also mindestens so viele Informationen enthalten, wie der Kontrakt. Sollte die Verifikation mit der Verwendung des Vertrags funktionieren, dann sollte sie grundsätzlich auch mit der Verwendung der Implementierung funktionieren, wenn die Implementierung den Kontrakt erfüllt. An dieser Stelle muss noch die Einschränkung getroffen werden, dass die Methode sich unter der Verwendung von *Method Treatment: Expand* verifizieren lässt, da wir an dieser Stelle das *Expand*-Verhalten betrachten wollen.

https://dx.doi.org/10.24355/dbbs.084-201801311013

Hypothese 18 – Expand Local Queries (Beweisbarkeit): Sollte sich ein Beweis mit der Option *Off* schließen lassen, dann lässt er sich auch mit *On* schließen, sofern die Methoden aller im Spezifikationsfall verwendeten Queries mit der *Method Treatment*-Option *Expand* verifiziert werden können.

Beschreibung 4.12 – Arithmetic Treatment: *Arithmetic Treatment* ∈ {*Basic*, *DefOps*, *Off*}. Der Umfang von *Basic* ermöglicht es, polynomiale Ausdrücke zu vereinfachen. Auch lineare Ungleichungen können mit dem *Basic*-Parameter behandelt werden. *DefOps* ermöglicht es, zusätzlich mathematische Symbole und Prädikate zu ersetzen. Diese umfassen beispielsweise Punktoperatoren wie * und / [ABB+16, S. 553], oder auch Java Dynamic Logic-Prädikate wie `addJint` oder auch `inInt`. *Model Search* bietet vor allem eine Unterstützung für nichtlineare Ungleichungsketten. Die Option konfiguriert KeY so, dass es nach einem Model sucht, welches die ursprüngliche Formel beschreibt. Sie ermöglicht es, für nicht erfüllbare Ziele, die nur polynomiale Gleichungen oder Ungleichungen enthalten, immer Gegenbeispiele zu finden. Für einige Ziele können die Beweise mit dieser Option auch geschlossen werden [ABB+16, S. 538] [Gla12, S. 456f].

DefOps erweitert demnach den Funktionsumfang von *Basic*. Daraus müsste sich schließen lassen, dass die Menge der mit *Basic* automatisch schließbaren Beweise eine Teilmenge der mit *DefOps* schließbaren Beweise darstellt.

Hypothese 19 – Arithmetic Treatment (Basic und DefOps): Sollte sich ein Beweis mit der Option *Basic* schließen lassen, so lässt er sich auch mit der Option *DefOps* schließen.

Der Parameter *Model Search* stellt jedoch eine andere Funktionalität zur Verfügung. Es sollte deshalb keine Teilmengenbeziehung zwischen *Model Search* und den anderen beiden Optionen feststellbar sein.

Hypothese 20 – Arithmetic Treatment (Model Search): Es gibt Beweise, die sich nicht mit *DefOps* oder *Basic*, aber mit *Model Search* schließen lassen. Auch gibt es Beweise, die sich mit *DefOps* oder *Basic*, aber nicht mit *Model Search* schließen lassen. Nicht jeder Beweis, der sich mit *DefOps* oder Basic schließen lässt, lässt sich auch mit *Model Search* schließen. Auch lässt sich nicht jeder Beweis, der sich mit *Model Search* schließen lässt, mit *DefOps* schließen.

Beschreibung 4.13 – Quantifier Treatment: *Quantifier Treatment* ∈ {*None*, *No Splits*, *No Splits with Progs*, *Free*}. Dieser Parameter behandelt die Instanziierung von Quantifizierern. Das bedeutet, dass eine neue Annahme erzeugt wird, bei der die Variable der Quantifizierung durch eine bestimmte Instanz ersetzt wird [ABB+16, S. 512]. Die Option *None* hat erneut die Funktionalität, dass KeY keine automatischen Schritte vornimmt, in diesem Fall im Bezug auf Quantifizierungsformeln. *Free* (in der Literatur auch als *Unrestricted* bezeichnet [BHS13, S. 11]) erlaubt es KeY beliebig zu instanziieren. KeY versucht jedoch mit Hilfe von Heuristiken die Anzahl der Quantifizierungen möglichst gering zu halten. *No Splits* führt nur Instanziierungen aus, wenn diese nicht zu einem Aufteilen des Beweisbaums führen.

https://dx.doi.org/10.24355/dbbs.084-201801311013

Die letzte Option, *No Splits with Progs*, wählt *No Splits* für Programme aus und für Formeln *Free* [ABB+16, S. 553].

Diese Option dürfte demnach nur eine Auswirkung auf Methoden haben, bei deren Beweis auch Quantifikatoren (siehe Beschreibung 2.2) verwendet werden. Ansonsten dürfte die Wahl des Parameters keine Auswirkung haben.

Hypothese 21 – Quantifier Treatment ohne Quantifizierungen (Beweisbarkeit): Sollte eine *JavaDL*-Formel keine Quantifizierer enthalten, so hat die Wahl der *Quantifier treatment* keine Auswirkung auf die Beweisbarkeit.

Hypothese 22 – Quantifier Treatment ohne Quantifizierungen (Beweisaufwand): Sollte eine *JavaDL*-Formel keine Quantifizierer enthalten, so hat die Wahl der *Quantifier treatment* keine Auswirkung auf den Beweisaufwand.

Wie auch bei *Proof Splitting* ist es wichtig zu beachten, dass Programme in *JavaDL* umgewandelt werden. Es ist also davon auszugehen, dass es auch bei Java-Programmen einen Unterschied zwischen *No Splits* und *No Splits with Progs* gibt. Da das Aufteilen des Beweises in mehrere Äste dazu führt, dass mehr Taclets auf mehreren Ästen angewendet werden, sollte der Beweisaufwand durch das Erlauben des Aufteilens steigen. Da dies jedoch für Beweise nötig sein kann, ist davon auszugehen, dass es Beweise gibt, die sich nur mit dem Aufspalten in mehrere Äste schließen lassen. Sollte dies nicht nötig sein, erzwingt das Verwenden von *Free* dies allerdings nicht. Es ist also davon auszugehen, dass die *Beweisbarkeit* mit einer höheren Freiheit für den automatischen Beweiser steigt.

Hypothese 23 – Quantifier Treatment (Beweisbarkeit): Betrachtet man den Parameter *Quantifier Treatment*, dann lässt sich jeder Beweis, der sich mit *None* schließen lässt, auch mit den anderen Optionen schließen. Jeder Beweis, der sich mit *No Splits* schließen lässt, lässt sich auch mit *No Splits with Progs* und *Free* schließen. Jeder Beweis, der sich mit *No Splits with Progs* schließen lässt, lässt sich auch mit *Free* schließen.

Hypothese 24 – Quantifier Treatment (Beweisaufwand): Betrachtet man den Parameter *Quantifier Treatment*, dann ist der Beweisaufwand mit der Option *Free* mindestens so groß wie der Aufwand mit der Option *No Splits with Progs*. Dessen Aufwand ist wiederum mindestens so groß wie mit der Option *No Splits*. Für diesen ist der Aufwand mindestens so groß ist wie der mit *None*.

Beschreibung 4.14 – Class Axiom Rule: *Class Axiom Rule* $\in$ {*Free*, *Delayed*, *Off*}. Der Tooltip besagt an dieser Stelle, dass die Axiome der Klassen, wie beispielsweise Invarianten, behandelt werden. Die Option selbst regelt, wann diese Axiome behandelt werden. *Free* erlaubt es dem automatischen Beweiser, zu beliebigen Zeitpunkten die Axiome zu bearbeiten, während *Delayed* dies so lange möglich verzögert. *Off* untersagt KeY das Bearbeiten der Axiome vollständig.

Da der Parameter *Class Axiom Rule* sich auf das Bearbeiten von Klassen-Axiomen wie beispielsweise Invarianten bezieht, sollte der Parameter nur eine Auswirkung bei Beweisen von Methoden aus Klassen mit Axiomen haben.

https://dx.doi.org/10.24355/dbbs.084-201801311013

Hypothese 25 – Class Axiom Rule (Beweisbarkeit, ohne Axiome): Sollte eine Methode bewiesen werden, die keine Invarianten und Klassenfelder benutzt, so hat der Parameter *Class Axiom Rule* keine Auswirkung auf die Beweisbarkeit.

Hypothese 26 – Class Axiom Rule (Beweisaufwand, ohne Axiome): Sollte eine Methode bewiesen werden, die keine Invarianten und Klassenfelder benutzt, so hat der Parameter *Class Axiom Rule* keine Auswirkung auf den Beweisaufwand.

Sollte es ein solches Axiom geben, muss nach der Ausführung der Methode überprüft werden, ob es eventuell verletzt wird. Damit dies der Fall sein kann, muss ein Feld verändert worden sein. Sollte ein Feld verändert werden und es gibt an dem Objekt des Felds eine Klasseninvariante, so müssen die Invarianten aufgelöst werden, um zu überprüfen, ob diese verletzt werden. Da vor dem Auflösen der Axiome nicht bekannt sein kann, welche Felder in Axiomen vorkommen, spielt es dabei keine Rolle, ob das Feld in der Klasseninvariante genutzt wird oder nicht.

Hypothese 27 – Class Axiom Rule (mit Axiomen und Schreibzugriffen): Sollte eine Methode bewiesen werden, die ein Feld beschreibt, und es gibt mindestens eine Klasseninvariante zu dem Objekt des Feldes, so kann der Beweis nicht mit *Off* geschlossen werden.

Sollte die Methode jedoch keine Schreibzugriffe auf Felder vornehmen, können die Axiom als Nachbedingung auch nicht verletzt werden. Wenn sie auch nicht als Vorbedingung für den Beweis benötigt werden, ist es auch nicht nötig diese aufzulösen.

Hypothese 28 – Class Axiom Rule (Axiome nicht benötigt): Sollte eine Methode bewiesen werden, die keine Schreibzugriffe vornimmt und die für den Beweis die Axiome nicht benötigt, kann der Beweis auch mit der Option *Off* geschlossen werden, wenn er mit einer anderen Option für *Class Axiom Rule* geschlossen werden kann.

Beschreibung 4.15 – Auto Induction: *Auto Induction* $\in$ {*On*, *Restricted*, <u>*Off*</u>}. Laut dem Tooltip von KeY behandelt der Parameter *Auto Induction* den automatischen Einsatz von Induktion für Quantifizierer. Sollte ein Ziel eine Formel der Form $\Longrightarrow$ $\backslash forall\ int\ i; 0 \leq i \rightarrow \phi$ enthalten, kann KeY eine Induktion mit der Hypothese ϕ durchführen. Dies funktioniert auch für einige andere Formeln. Auch Formeln der Form $\Longrightarrow$ $\backslash forall\ int\ i; 0 \leq i \rightarrow \phi \wedge \psi$ können mit dieser Regel zu ($\Longrightarrow$ $\backslash forall\ int\ i; 0 \leq i \rightarrow \phi)$ ==> ψ umgewandelt werden. Dies gilt erneut auch für ähnliche Formeln. Als Optionen für diesen Parameter stehen „On“, „Restricted“ und „Off“ zur Auswahl. *On* und *Off* erlauben oder verbieten den Einsatz dieser Regel in jedem Fall. *Restricted* erlaubt dies nur, wenn als Suffix der Quantifizierungsvariablen *IND* verwendet wird. Es konnte jedoch leider kein entsprechender Nachweis dazu in der Dokumentation von KeY, in dem KeY-Buch oder auf einer der Webseiten des KeY-Projekts gelisteten Quelle gefunden werden.

Da leider keine wirklich ausführliche Erklärung zu dem Parameter *Auto Induction* gefunden wurde, wird von dem Aufstellen einer Hypothese zu diesem Parameter abgesehen.

https://dx.doi.org/10.24355/dbbs.084-201801311013

Zusammenfassend kann gesagt werden, dass zu den 14 Optionen zur Beweissuche 28 Hypothesen aufgestellt wurden. Von den 14 Parametern wurden für den weiteren Verlauf der Arbeit 3 Optionen festgelegt: *Block Treatment: Contract*, *Method Treatment: Contract* und *Auto Induction: Off*. Für den Parameter *Loop Treatment* nur die Optionen *Invariant* und *Loop Scope Invariant*. Eine um diese Einschränkungen ergänzte Darstellung der Optionen kann in Abbildung 4.2 gesehen werden.

4.2 Taclet Optionen

KeY stellt neben den Optionen zur Stategie der Beweissuche auch noch Taclet-Optionen zur Verfügung. Diese steuern, welche Taclets von KeY geladen werden und welche nicht. Eine graphische Zusammensetzung wird in Abbildung 4.3 dargestellt. Ein Taclet stellt hierbei eine auf die Formel anwendbare Regel dar (siehe Abschnitt 2.5). Dadurch hat dies meist nicht nur eine Auswirkung darauf, wie der Beweis durchgeführt wird, sondern vor allem darauf, was gezeigt wird. So lässt sich beispielsweise steuern, ob ein Integer-Überlauf möglich ist, oder wie mit dem möglichen Auftreten von `RuntimeExceptions` gearbeitet wird. Aus diesem Grund werden zu den meisten Taclet-Optionen keine Hypothesen aufgestellt. Diese werden dennoch diskutiert, da sie konfiguriert werden müssen und diese das Resultat der Verifikation ändern können.

Als Quelle dient, wenn nicht anders angegeben, der Tooltip aus KeY [Key].

Beschreibung 4.16 – JavaCard: *JavaCard* ∈ {<u>*On*</u>, *Off*}. Sollte der Parameter aktiviert werden, werden spezielle JavaCard-Features vorausgesetzt (siehe Abschnitt 2.1.1). Ein Beispiel hierfür ist der in JavaCard vorhandene Transaktionsmechanismus. Dieser sichert zu, dass entweder alle oder keine Speicherupdates einer Transaktion ausgeführt werden [HP04, S. 114]. Das Verhalten von KeY ist für JavaCard korrekt, wenn der Parameter aktiviert ist. Für Java muss er allerdings deaktiviert sein, da KeY sonst falsche Annahmen über den Quelltext treffen kann.

Da in dieser Arbeit Java-Quelltext und kein JavaCard-Quelltext verifiziert werden soll, ist dieser Parameter für die Verifikation im Rahmen dieser Arbeit immer deaktiviert. Dies entspricht nicht der Standardauswahl und muss dementsprechend angepasst werden.

Beschreibung 4.17 – Strings: *Strings* ∈ {<u>*On*</u>, *Off*}. Sollte *Strings* aktiviert sein, so werden die zu der Java-Klasse String gehörenden Taclets geladen. Wenn der Parameter deaktiviert ist, werden diese Taclets nicht aktiviert. Es gibt jedoch den Hinweis, dass KeY nach dem Deaktivieren der String-Taclets unvollständig ist, also nicht alle verifizerbaren Methoden verifiziert werden können (siehe Abschnitt 2.3).

An dieser Stelle ist es interessant, ob es überhaupt einen praktischen Einsatzzweck im Rahmen dieser Arbeit gibt, den Parameter zu deaktivieren. Es wird deshalb die Behauptung aufgestellt, dass das aktiviert lassen des Taclets immer von Vorteil ist.

https://dx.doi.org/10.24355/dbbs.084-201801311013

Abbildung 4.2: Darstellung der Optionen zur Strategie der Beweissuche als Feature Diagramm (mit Einschränkungen)

https://dx.doi.org/10.24355/dbbs.084-201801311013

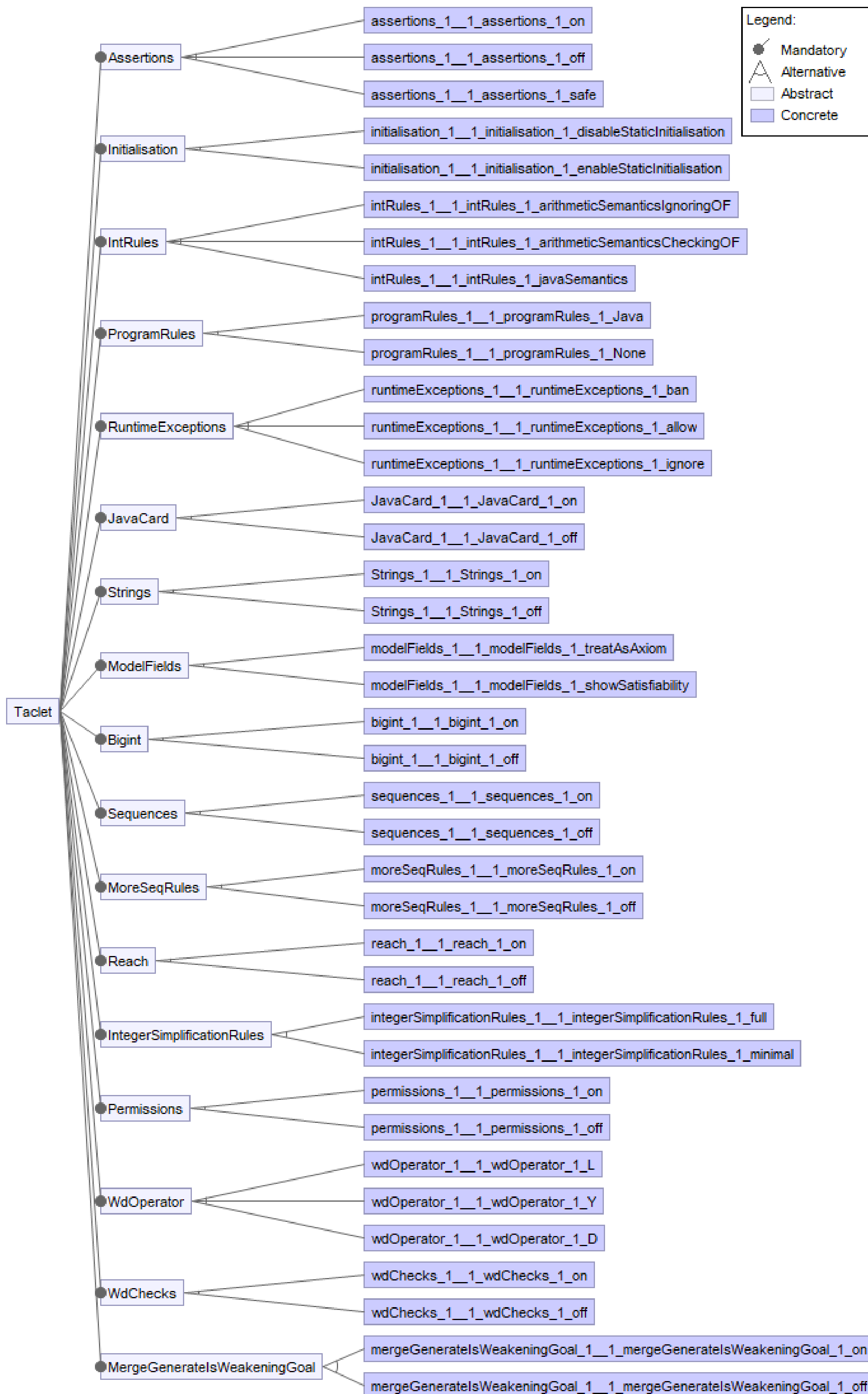

Abbildung 4.3: Darstellung der Taclet Optionen als Feature Diagramm

https://dx.doi.org/10.24355/dbbs.084-201801311013

Hypothese 29 – Strings (Beweisbarkeit): Jeder Beweis, der sich im Rahmen dieser Arbeit mit dem Parameter *Strings:Off* schließen lässt, lässt sich auch mit *Strings:On* schließen.

Hypothese 30 – Strings (Beweisaufwand): Sollte sich ein Beweis mit *Strings:On* und *Strings:Off* schließen lassen, so ist der Beweisaufwand unabhängig von der Auswahl der *Strings*-Option.

Beschreibung 4.18 – Assertions: *Assertions* ∈ {<u>*Safe*</u>, *On*, *Off*}. Bei einer Zusicherung handelt es sich um das `assert`-Statement aus Java. Sollten Zusicherungen aktiviert sein, wird ein boolscher Ausdruck evaluiert und muss `true` ergeben. Sollte dies nicht der Fall sein, wird ein `AssertionError` erzeugt und geworfen [GJSB05, S. 373ff]. Diese Zusicherungen lassen sich jedoch auch deaktivieren [jcp]. *Off* hat in KeY zur Folge, dass `assert` Statements aus Java übersprungen werden, was dem Verhalten von deaktivierten Zusicherungen entspricht. *On* führt die Statements aus und wirft eine `AssertionException`, sofern sie zu `false` ausgewertet werden.[1] Bei *Save* handelt es sich um die Standardoption. Sie verlangt, dass die Spezifikation auch ohne `assert`-Statements erfüllt werden müssen, aber zudem auch alle zugesicherten Bedingungen eingehalten werden.

Eine Zusicherung soll das Vertrauen eines Entwicklers in seinen Quelltext erhöhen, indem überprüft wird, ob seine Erwartungen an den Zustand des Systems erfüllt sind [GJSB05, S. 376]. Ein Entwickler kann die Zusicherungen zwar auch nutzen, um einen Fehler im Programmfluss zu erzeugen, jedoch sollte hierfür eher eine Abfrage und eine `Exception` verwendet werden. Der Einsatz einer Zusicherung, die falsch sein kann, ist zwar möglich, sollte in der Entwicklung allerdings vermieden werden, da es nicht zu dem Konzept der Zusicherung passt. Dies könnte also als stilistischer Programmierfehler betrachtet werden, welcher mit überprüft werden sollte. Aus diesem Grund wird *Save* verwendet.

Beschreibung 4.19 – Bigint: *Bigint* ∈ {<u>*On*</u>, *Off*}. Sollte *Bigint* deaktiviert sein, werden alle Taclets, die das JML-Konstrukt `\bigint` behandeln ebenfalls deaktiviert. Bei `\bigint` handelt es sich um einen von JML eingeführten Datentyp. Dieser ist zwar kein primitiver Datentypen, Vergleiche zwischen zwei Objekten werden aber dennoch auf Basis des Werts und nicht der Objektreferenz durchgeführt. Im Gegensatz zu `int` gibt es bei `\bigint` keinen Überlauf [LPC+13, S. 140].

`\bigint` kommt in den Spezifikationen vor, die im Rahmen dieser Arbeit selbst erstellt wurden (siehe Abschnitt 3.3). Deshalb muss der Parameter zumindest für die Beweise im Zusammenhang mit diesem Feld aktiviert werden. Die Frage ist, ob es einen Vorteil bietet, den Parameter in den anderen Beweisen zu deaktivieren. Deshalb werden Hypothesen aufgestellt, die analog zu denen des Parameters *Strings* sind.

[1] Vermutlich handelt es sich hierbei um einen Fehler im Tooltip, da die Option sich laut Tooltip wie Java verhält und demnach einen `AssertionError` erzeugen müsste, was dem tatsächlichen Verhalten und der Aussage im KeY-Buch entspricht [ABB+16, S. 261].

https://dx.doi.org/10.24355/dbbs.084-201801311013

Hypothese 31 – Bigint (Beweisbarkeit): Jeder Beweis, der sich im Rahmen dieser Arbeit mit dem Parameter *Bigint:Off* schließen lässt, lässt sich auch mit *Bigint:On* schließen.

Hypothese 32 – Bigint (Beweisaufwand): Sollte sich ein Beweis mit *Bigint:On* und *Bigint:Off* schließen lassen, so ist der Beweisaufwand unabhängig von der Auswahl der *Bigint*-Option.

Beschreibung 4.20 – Initialisation: *Initialisation* ∈ {*DisableStaticInitialisation*, *EnableStaticInitialisation*}. Die *Initialisation*-Option betrifft die statische Initialisierung von Java-Klassen. Hierbei handelt es sich um einen Java-Block, der ausgeführt wird, wenn eine Klasse initialisiert wird [GJSB05, S. 239]. Die Option, und damit die Bearbeitung von statischen Initialisierungen, lässt sich aktivieren oder deaktivieren. Sollte sie deaktiviert sein, werden statische Initialisierungen ignoriert und der Zustand zu Beginn der Methodenverifikation entspricht nicht dem der eigentlichen Ausführung des Java-Quelltexts, sofern statische Initialisierung im Quelltext vorkommt. Dafür werden andernfalls die Beweise komplexer.

Da die Auswirkungen dieses Parameters bekannt sind, wird zu diesem Parameter keine Hypothese aufgestellt. Sollte eine statische Initialisierung für die Verifikation benötigt werden, muss dieser Parameter aktiviert werden. Andernfalls entsteht durch die Aktivierung ein zusätzlicher Aufwand, da statische Initialisierungen gesucht und gegebenenfalls ausgewertet werden. Beispielsweise versucht KeY zu einem `Object`-Array statische Initialisierungen zu laden, wenn ein `Object`-Array im Beweis vorkommt. Dies schlägt jedoch fehl, da keine Initialisierung gefunden werden kann, wodurch eine `Exception` auftritt. Deshalb wird der Parameter nicht aktiviert, sofern dies für die Verifikation der jeweiligen Methode nicht notwendig ist.

Beschreibung 4.21 – IntRules: *IntRules* ∈ {*ArithmeticSemanticsIgnoringOF*, *ArithmeticSemanticsCheckingOF*, *JavaSemantics*}. Die Standardoption, *ArithmeticSemanticsIgnoringOF* prüft keinen Überlauf (siehe Abschnitt 2.1.2). Das bedeutet, dass Integer als mathematische Ganzzahlen interpretiert werden. Die Folge ist, dass sich Bedingungen oftmals leichter entkräften lassen, aber dafür auch fehlerhafte Java-Programme bewiesen werden können, KeY sich also nicht korrekt verhält. *ArithmeticSemanticsCheckingOF* prüft einen Überlauf, behandelt Integer dafür aber nach wie vor wie mathematische Ganzzahlen. Dadurch wird verhindert, dass sich KeY nicht korrekt verhält, das Modell ist dafür aber nicht komplett, da Java-Programme den Überlauf nutzen können, um Funktionalität zur Verfügung zu stellen. Die Wahlmöglichkeit *JavaSemantics* arbeitet mit Integern wie Java. Die Zahlen bewegen sich also nur im definierten Bereich. Die Bitvektoroperationen von Java werden hierbei in KeY mit der Modulo-Operation nachmodelliert. Der Vorzug dieser Möglichkeit liegt darin, dass sie sowohl korrekt, als auch vollständig ist (siehe Abschnitt 2.3). Der Nachteil liegt bei dieser Auswahl in der höheren Komplexität.

An dieser Stelle liegt erneut eine fachliche Entscheidung vor, wie mit Integer-Überläufen umzugehen ist. Hypothesen zu erstellen ergibt an dieser Stelle auf Grund der fachlichen Begründung hinter der Wahl keinen Sinn. Bei den meisten Beweisen wird *JavaSemantics* genutzt. Allerdings gibt es auch einzelne Beweise, bei denen eine

https://dx.doi.org/10.24355/dbbs.084-201801311013

Überlauf-Problematik bekannt ist und die Methode unter der Annahme verifiziert werden soll, dass kein Überlauf auftritt. Dies hat den Vorteil, dass die restliche Funktionalität verifiziert werden kann. Von der *ArithmeticSemanticsCheckingOF*-Option wird abgesehen, da ein Überlauf von Programmen genutzt werden kann, um Funktionalität zur Verfügung zu stellen (siehe Abschnitt A.2.12). Ist dies nicht der Fall, sollte der Überlauf über die Spezifikation ausgeschlossen werden.

Beschreibung 4.22 – IntegerSimplificationRules: *IntegerSimplificationRules* ∈ {<u>*Full*</u>, *Minimal*}. Dieser Parameter ermöglicht es, selten verwendete Regeln für Ganzzahlen zu deaktivieren. Hierfür muss die *Minimal*-Option gewählt werden. Diese Option sorgt allerdings dafür, dass KeY unvollständig wird. Die Auswahl von *Minimal* verhindert die Verwendung von *JavaSemantics* bei *IntRules*, da dies sonst nicht geladene Regeln benötigt. Bei *Full* werden auch die seltenen Regeln geladen. Sollte eine Regel geladen werden, wird die automatische Beweissuche die Regel normal einsetzen.

Erneut können mit Hilfe dieser Option Regeln deaktiviert werden. Es muss allerdings beachtet werden, dass das Wählen von *Minimal* den Einsatz von *IntRules:JavaSemantics* ausschließt. Wie auch bei *Strings* und *Bigint* ist es interessant, ob es bei der Verifikation eines Java-Programms einen Vorteil gibt, wenn die Option *Minimal* verwendet wird.

Hypothese 33 – IntegerSimplificationRules (Beweisbarkeit): Jeder Beweis, der sich im Rahmen dieser Arbeit mit dem Parameter *IntegerSimplificationRules:Minimal* schließen lässt, lässt sich auch mit *IntegerSimplificationRules:Full* schließen.

Hypothese 34 – IntegerSimplificationRules (Beweisaufwand): Sollte sich ein Beweis mit *IntegerSimplificationRules:Minimal* und *IntegerSimplificationRules:Full* schließen lassen, so ist der Beweisaufwand unabhängig von der Auswahl der *IntegerSimplificationRules*.

Beschreibung 4.23 – ModelFields: *ModelFields* ∈ {<u>*TreatAsAxiom*</u>, *ShowSatisfiability*}. Standardmäßig ist *TreatAsAxiom* ausgewählt. Bei dieser Auswahl wird die Erfüllbarkeit eines JML Modellfeldes nicht überprüft. Bei einem Modellfeld handelt es sich um ein gedachtes, im Speicher nicht existierendes Feld (siehe Beschreibung 2.1 Model und represents). Dies kann dazu führen, dass auch widersprüchliche Spezifikationen akzeptiert werden. Wenn *ShowSatisfiability* gewählt wurde, wird die Erfüllbarkeit überprüft. Hierbei können alle Modellfelder als Formel betrachtet werden, die genau dann erfüllbar ist, wenn es mindestens eine Belegung gibt, bei der die Formel wahr ist. Beispielsweise wären die Modellfelder `a` und `b` mit den Belegungsregeln `a=b` und b=a+1 nicht erfüllbar. Hierbei wird dies jedoch nur lokal innerhalb einer Spezifikation geprüft.

Durch die Auswahl von *ShowSatisfiability* werden zusätzliche Prüfungen aktiviert. Dies ist erneut eine fachlich motivierte Wahl und eignet sich deshalb nicht für eine Hypothese im Rahmen dieser Arbeit. Für den Rest der Arbeit, wird *TreatAsAxiom* verwendet, welches auch die Standardeinstellung ist.

https://dx.doi.org/10.24355/dbbs.084-201801311013

Beschreibung 4.24 – Sequences: *Sequences* ∈ {<u>*On*</u>, *Off*}. KeY bietet Funktionen bezüglich Sequenzen. Mit dem Begriff Sequenz ist hierbei eine endliche Sequenz gemeint, die die Funktionalität einer typsicheren Liste mit variabler Länge hat [ABB+16, S. 149f]. Die Sequenzen bieten verschiedene Optionen, so können zum Beispiel Elemente zu dieser Liste hinzugefügt und entfernt werden, diese Liste kann umgekehrt oder es kann ein Element in dieser gesucht werden. Auch bei diesem Parameter lassen sich alle Regeln deaktivieren.

Erneut stellt sich die Frage, ob der Einsatz von *Sequences:Off* im Rahmen dieser Arbeit einen Vorteil mit sich bringen würde. Deshalb wird, wie auch bei vorherigen Fällen, die Hypothese aufgestellt, dass der Parameter im Rahmen dieser Arbeit immer auf *Sequences:On* belassen werden kann, da *Sequences:Off* keinen Vorzug mit sich bringen würde.

Hypothese 35 – Sequences (Beweisbarkeit): Jeder Beweis, der sich im Rahmen dieser Arbeit mit dem Parameter *Sequences:Off* schließen lässt, lässt sich auch mit *Sequences:On* schließen.

Hypothese 36 – Sequences (Beweisaufwand): Sollte sich ein Beweis mit *Sequences:On* und *Sequences:Off* schließen lassen, so ist der Beweisaufwand unabhängig von der Auswahl der *Sequences*-Option.

Beschreibung 4.25 – MoreSeqRules: *moreSeqRules* ∈ {<u>*Off*</u>, *On*}. *moreSeqRules* bestimmt, welcher Regelsatz für Sequenzregeln geladen wird. Sollte die Option *On* gewählt werden, werden feingranularere Regeln bezüglich der Sequenzen geladen, um potenzielle Probleme mit Permutationen oder Informationsflüssen zu lösen. Worum es sich bei diesen Problemen genau handeln kann, wird leider nicht definiert. Durch die verkleinerte Granularität der Regeln müssen jedoch mehr Beweisschritte bei der Verifikation gemacht werden.

Aus dieser Beschreibung lassen sich auch direkt zwei Hypothesen schließen:

Hypothese 37 – MoreSeqRules (Beweisbarkeit): Jeder Beweis, der sich im Rahmen dieser Arbeit mit dem Parameter *MoreSeqRules:Off* schließen lässt, lässt sich auch mit *MoreSeqRules:On* schließen.

Hypothese 38 – MoreSeqRules (Beweisaufwand): Sollte sich ein Beweis mit *MoreSeqRules:Off* und *MoreSeqRules:On* schließen lassen, so ist der Beweisaufwand mit *Off* immer mindestens so groß wie mit *On*.

Beschreibung 4.26 – ProgramRules: *ProgramRules* ∈ {<u>*Java*</u>, *None*}. Mit Hilfe dieser Option lassen sich Regeln deaktivieren, die Java-Sprachkonstrukte bearbeiten. Hierfür muss die Option auf *None* gesetzt werden. Als Alternative steht *Java* zur Verfügung. *None* führt jedoch trivialerweise dazu, dass KeY für Java-Quelltext nicht vollständig ist (siehe Abschnitt 2.3).

Da in dieser Arbeit Java-Quelltext behandelt wird, wird auch die Option *Java* gewählt. Eine Hypothese ist aus diesem Grund nicht interessant.

https://dx.doi.org/10.24355/dbbs.084-201801311013

Beschreibung 4.27 – Reach: *Reach* ∈ {<u>*On*</u>, *Off*}. Mit der Wahl von *On* werden Regeln bezüglich der Erreichbarkeit aktiviert. Bei der Frage nach der Erreichbarkeit geht es darum, ob ein Objekt oder ein Feld über eine Kette von Referenzen von einem anderen Objekt aus erreicht werden kann [ABB+16, S. 251]. Dies kann über das *reach*-, oder das *reachLocs*-Schlüsselwort erreicht werden [ABB+16, S. 251f]. Sollte *Off* gewählt werden, werden diese Regeln deaktiviert.

Da die Beweise, die im Rahmen dieser Arbeit exemplarisch erzeugt wurden, keine Aussagen mit Hilfe des *reach*- oder *reachLocs*-Schlüsselworts treffen, wäre eine Untersuchung dieses Parameters nicht zielführend.

Beschreibung 4.28 – RuntimeExceptions: *RuntimeExceptions* ∈ {<u>*Ban*</u>, *Allow*, *Ignore*}. Theoretisch ist es bei Java möglich, dass bei jedem Methodenaufruf eine `RuntimeException` geworfen wird. Im Gegensatz zu allgemeinen `Exceptions`, müssen `RuntimeExceptions` nicht in der Methodensignatur angegeben oder von einem *Try-Catch*-Block behandelt werden. Dieser Parameter legt fest, wie KeY damit umgeht. *Ban* nimmt an, dass das Werfen einer nicht spezifizierten `RuntimeException` die Spezifikation verletzt. Da Java an dieser Stelle weniger restriktiv ist, ist KeY bei der Verwendung von *Ban* unvollständig (siehe Abschnitt 2.3). Im Gegensatz dazu verhält KeY sich bei der Verwendung der *Allow*-Option wie Java. Diese Benutzung von `RuntimeExceptions` ist jedoch oftmals schwieriger, als sie einfach als spezifikationsverletzenden Sonderfall zu behandeln. Die dritte Wahlmöglichkeit, *Ignore*, führt dazu, dass KeY davon ausgeht, dass keine `RuntimeExceptions` auftreten. Dies entspricht allerdings nicht dem Verhalten von Java und ist somit nicht garantiert korrekt.

RuntimeExceptions ist einer der Taclet-Parameter, welche festlegen, was mit dem Beweis gezeigt werden soll. Grundsätzlich wird im weiteren Verlauf dieser Arbeit *Ban* verwendet, da die Methoden der Java-Platform API Fehlerfälle explizit definieren und diese entsprechend behandelt werden können. Von den drei zur Verfügung stehenden Optionen ist dies die strengste Variante und findet am ehesten nicht spezifizierte Exceptions.

Beschreibung 4.29 – WdChecks: *WdChecks* ∈ {*On*, <u>*Off*</u>}. *WdChecks* steht für *welldefinedness checks*, also Wohldefiniertheitsprüfungen, und bezieht sich auf die JML Spezifikationen. Auch diese Option lässt sich entweder aktivieren oder deaktivieren. Hierbei geht es nicht nur um Methoden-Kontrakte, sondern um alle Spezifikationen. Die Kontrakte, Schleifeninvarianten und Blockverträge werden hierbei zu dem Zeitpunkt, zu dem sie verwendet werden, auf ihre Wohldefiniertheit hin geprüft. Ein Spezifikationsausdruck ist genau dann wohldefiniert, wenn er in keinem gültigen Zustand eine `Exception` auslösen kann. Damit ein Vertrag wohldefiniert ist, müssen mehrere Bedingungen erfüllt sein. So müssen zuerst die Vorbedingungen immer wohldefiniert sein. Die Klauseln müssen es in allen Zuständen sein, die auch die Vorbedingungen erfüllen und die Nachbedingungen müssen für alle Zustände wohldefiniert sein, die durch die Ausführung der jeweiligen Methode erreichbar sind [ABB+16, S. 281f].

https://dx.doi.org/10.24355/dbbs.084-201801311013

Der Parameter *WdChecks* ist erneut fachlich. Sollte eine solche zusätzliche Überprüfung benötigt sein, kann diese aktiviert werden. Im Rahmen dieser Arbeit wird allerdings von der Verwendung zusätzlicher *welldefinedness checks* abgesehen.

Beschreibung 4.30 – WdOperator: *WdOperator* $\in \{\underline{L}, Y, D\}$. Bei dieser Option wird die Wohldefiniertheitsprüfung von Formeln und Termen behandelt. Hierbei wird das Verfahren gewählt, mit dessen Hilfe die Formeln und Terme geprüft werden. L ist die Standardauswahl und basiert der McCarthy-Logik [HP04]. D basiert auf Kleene-Logik [BBM98]. Y ist äquivalent zur Option D [DMR08].

Da im Rahmen dieser Arbeit von der Verwendung von *welldefinedness checks* abgesehen wird, ergibt es keinen Sinn, die Art der nicht eingesetzten Überprüfungen zu definieren. Der Parameter wird deshalb im Rahmen dieser Arbeit nicht näher untersucht. Da allerdings eine Option gewählt werden muss, wird die Standardoption L verwendet.

Beschreibung 4.31 – MergeGenerateIsWeakeningGoal: *MergeGenerateIsWeakeningGoal* $\in \{\underline{Off}, On\}$ Der Parameter wurde im Rahmen des Merge-Features hinzugefügt (siehe Beschreibung 4.8 Merge Point Statements). Obwohl diese Regel in den Taclet-Optionen gelistet wird, handelt es sich hierbei laut eines Entwicklers des Key-Teams nicht um ein Taclet, sondern um eine sogenannte Build-In-Regel, also eine fest in Key hinterlegte Regel. Laut des Entwicklers handelt es sich hierbei um die *CloseAfterJoin*-Regel, welche zusätzliche Beweisäste hinzufügt [Sch15, S. 33f, Kapitel 4.3].

Da im Rahmen dieser Arbeit keine Merge-Statements verwendet wurden und eine entsprechende Empfehlung eines Key-Entwicklers vorliegt, wird diese Option bei der Standardeinstellung *off* belassen.

Beschreibung 4.32 – Permissions: *Permissions* $\in \{\underline{Off}, On\}$. Es liegt keine Beschreibung in KeY für diesen Parameter bereit. Auf Nachfrage beim KeY-Team wurde jedoch gesagt, dass es sich beim Aktivieren von *Permissions* um eine experimentelle Option für die Arbeit mit mehreren, parallelen Threads handelt. Der Parameter geht zurück auf *Permission accounting*, welches eine Spezifikation darstellt, die Beeinflussung von Threads unterbinden soll [ABB+16, S. 378]. Hierbei werden Programme mit Zugriffsberechtigungen annotiert, so dass spezifiziert werden kann, wie und auf welche Speicherbereiche Threads zugreifen können [HM15, 165]. Es wurde außerdem gesagt, dass diese Option derzeit stets deaktiviert sein sollte.

Auf Grund der Empfehlung eines Entwicklers des KeY-Teams und der Tatsache, dass keine parallelen Prozesse untersucht werden, wird der *Permissions*-Parameter für den Weiteren Verlauf der Arbeit mit der *Off*-Option festgelegt.

Wie auch für die Optionen zur Strategie der Beweissuche wurden die in diesem Kapitel festgelegten Taclet-Optionen in der zusammenfassenden Grafik markiert und die in den Tooltips erwähnte Einschränkung mit aufgenommen (siehe Abbildung 4.4).

https://dx.doi.org/10.24355/dbbs.084-201801311013

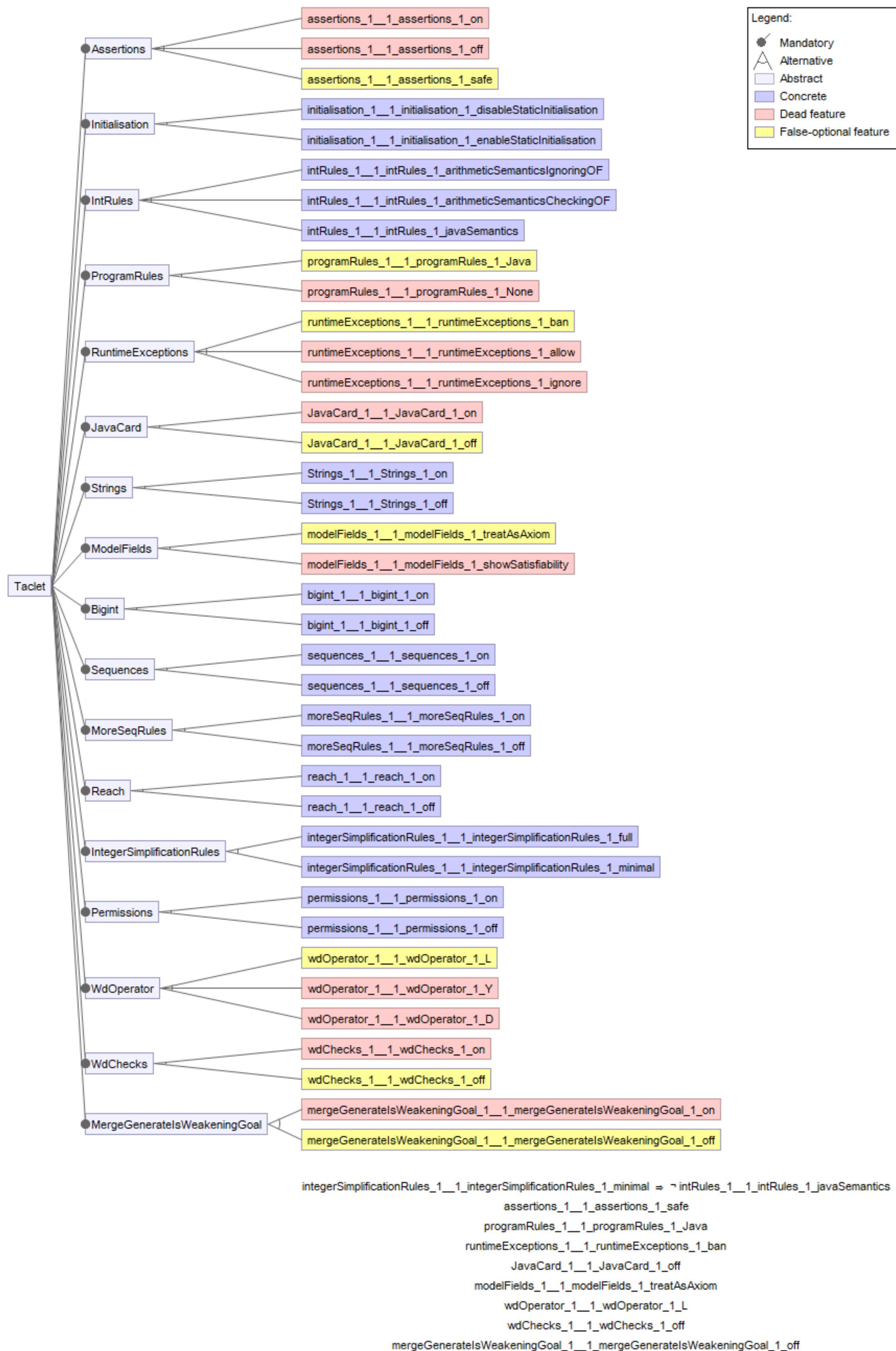

Abbildung 4.4: Darstellung der Taclet Optionen als Feature Diagramm (mit Einschränkungen)

https://dx.doi.org/10.24355/dbbs.084-201801311013

4.3 Allgemeine Optionen

Allgemeine Optionen stehen auf der Oberfläche unter *Options* zur Verfügung. Unter diesem Menüpunkt befinden sich auch die *Taclet Options*, diese wurden jedoch bereits im vorherigen Abschnitt behandelt. Es gibt zwei allgemeine Optionen, die hier vorgestellt werden sollen: *Minimize Interaction* und die *SMT-Solver-Optionen*.

Beschreibung 4.33 – Minimize Interaction: *Minimize Interaction* $\in$ {*Aktiviert*, *Deaktiviert*}. Sollte der Parameter aktiviert sein, versucht KeY die Interaktionen mit dem Benutzer zu minimieren. Dies hat zur Auswirkung, dass KeY den Benutzer bei einzelnen Punkten nicht mehr nach Unterstützung fragt, sondern das Problem selbstständig zu lösen versucht. Ein Beispiel hierfür wäre die Auswahl von Instanzen [ABB+16, S. 500].

Da im Rahmen dieser Arbeit nur vollautomatische Beweise betrachtet werden sollen, wird dieser Parameter im weiteren Verlauf aktiviert sein.

Beschreibung 4.34 – SMT-Solver-Options: Bei einem SMT-Solver handelt es sich um ein Programm, welches die Erfüllbarkeit von bestimmten Formeln überprüfen kann. Bei diesen Formeln handelt es sich um Formeln aus der *Satisfiability Modulo Theory*. Diese Theorie erweitert die allgemeine aussagenlogische Erfüllbarkeit um weitere mathematische Konstrukte [dMB08, S.1].
KeY verwendet standardmäßig den Z3-SMT-Solver[ABB+16, S. 438]. Dieser wird näher in [dMB08] erläutert und soll in dieser Arbeit nicht weiter behandelt werden. Für die SMT-Solver-Optionen stehen mehrere Punkte zur Verfügung, die beispielsweise steuern, welche Werte für Variablen instanziiert werden dürfen. Unter diesen Optionen lässt sich sogar der komplette Solver austauschen [ABB+16, S. 537].

Eine vertiefende Arbeit ermöglicht die Betrachtung des Beweises beim Einsatz eines SMT-Solvers. Hierbei handelt es sich jedoch nicht um eine Durchführung eines Beweises durch Key [ABB+16, S. 538], sodass dies nicht in dieser Arbeit behandelt werden soll.

4.4 Zusammenfassung

In diesem Kapitel wurden insgesamt 38 Hypothesen aufgestellt, welche Aussagen über die Beweisbarkeit und den Beweisaufwand mit dem Einsatz verschiedener Optionen treffen. Diese beziehen sich sowohl auf die Strategie-Optionen als auch auf die Taclet-Optionen. Ein Teil dieser Hypothesen beschreibt hierbei eine Reihenfolge zwischen den Optionen (siehe Abbildung 4.5 für die Hypothesen zur Beweisbarkeit und Abbildung 4.6 für die Hypothesen zum Beweisaufwand). Die anderen Hypothesen sagen entweder aus, dass die Optionen einen identischen Aufwand haben (siehe Tabelle 4.1), oder dass es einen Unterschied geben müsste, der aber nicht einseitig ist (dies gilt für die Hypothesen 16 bezüglich *Query Treatment On* und *Restricted* und 20 bezüglich *Arithmetic Treatment*). Nur Hypothese 27 sticht hier heraus. Diese sagt aus, dass kein Beweis für eine Methode mit Axiomen und Schreibzugriffen geschlossen werden kann, wenn *Class Axiom Rule Off* gewählt wird.

Zu den Optionen lässt sich sagen, dass es bei der Ausführlichkeit der Beschreibungen einen großen Unterschied zwischen den Optionen gibt. Einige der Optionen und

https://dx.doi.org/10.24355/dbbs.084-201801311013

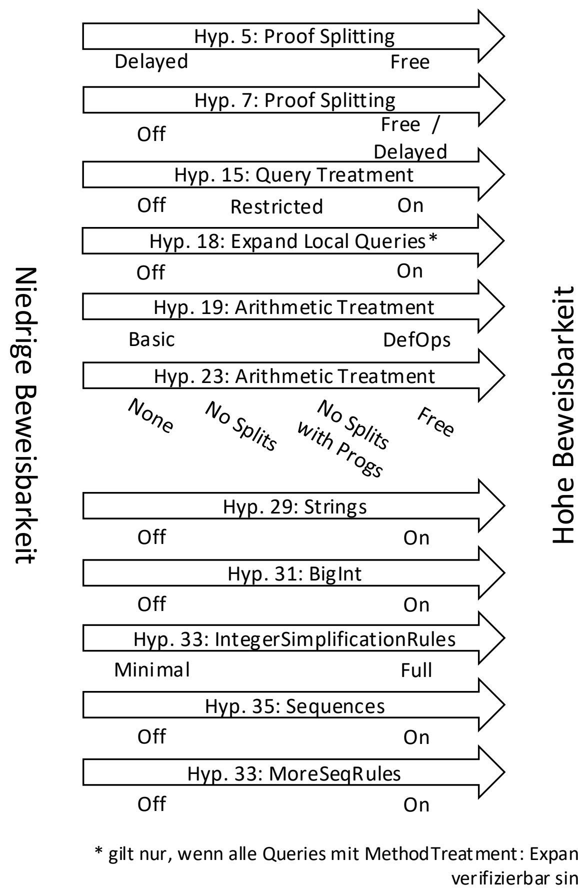

Abbildung 4.5: Hypothesen zur Beweisbarkeit (mit vermuteter Reihenfolge)

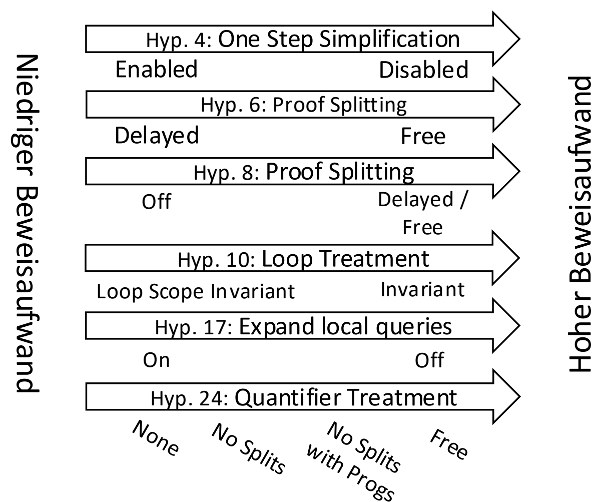

Abbildung 4.6: Hypothesen zum Beweisaufwand (mit vermuteter Reihenfolge)

https://dx.doi.org/10.24355/dbbs.084-201801311013

Hypothese	Parameter	Beweisbarkeit / Beweisaufwand	Einschränkung
3	One Step Simplification	Beweisbarkeit	
9	Loop Treatment	Beweisbarkeit	
11	Dependency Contract	Beweisbarkeit	Nur ohne accessible
12	Dependency Contracts	Beweisaufwand	Nur ohne accessible
13	Query Treatment	Beweisbarkeit	Ohne Queries
14	Query Treatment	Beweisaufwand	Ohne Queries
21	Quantifier Treatment	Beweisbarkeit	Ohne Quantifizierer
25	Class Axiom Rule	Beweisbarkeit	Methode ohne Invarianten und Klassenfeldern
26	Class Axiom Rule	Beweisaufwand	Methode ohne Invarianten und Klassenfeldern
28	Class Axiom Rule	Beweisbarkeit	Methode mit unbenötigten Axiomen ohne Schreibzugriff
29	Strings	Beweisaufwand	
31	BigInt	Beweisaufwand	
33	IntegerSimplification-Rules	Beweisaufwand	
35	Sequences	Beweisaufwand	
37	MoreSeqRules	Beweisaufwand	

Tabelle 4.1: Hypothesen mit Gleichheitsaussage

https://dx.doi.org/10.24355/dbbs.084-201801311013

Parameter sind sehr gut beschrieben, auch wenn durchaus Hintergrundwissen vorausgesetzt wird. Jedoch gibt es vor allem bei den Taclet-Optionen das Problem, dass die Qualität der Beschreibung von Parameter zu Parameter stark schwankt. Dies hängt natürlich auch damit zusammen, dass einige der Parameter noch einen eher experimentellen Hintergrund haben, jedoch könnte gerade an dieser Stelle der Anwender mit Verweisen auf einzelne Kapitel im KeY-Buch oder auf wissenschaftliche Veröffentlichungen unterstützt werden. Die Aussage, dass sich beispielsweise der Parameter *mergeGenerateIsWeakeningGoal* auf *Join Prozeduren* bezieht, gibt zwar einen Hinweis. Einen Verweis auf eine weiterführende Lektüre, was eine solche *Join Prozedur* überhaupt ist, würde die Einstiegshürde jedoch stark vereinfachen. Ein positives Beispiel an dieser Stelle ist der *wdOperator*-Parameter, bei dem genau dies in der Beschreibung getan wird. Auch gibt es einige Taclet-Optionen, die das Laden von einzelnen Regelsätzen unterbinden, jedoch wird nicht dargelegt, wieso man dies wollen sollte. Ob es einen Vorzug mit sich bringt, soll im weiteren Verlauf dieser Arbeit noch evaluiert werden. Wenn dem allerdings nicht so ist, stellt sich die Frage, wieso dem Anwender diese Optionen überhaupt angeboten werden.

https://dx.doi.org/10.24355/dbbs.084-201801311013

https://dx.doi.org/10.24355/dbbs.084-201801311013

5. Empirische Evaluierung von Key-Optionen

In Kapitel 3 wurden mehrere Verträge erstellt, die mit Hilfe von KeY verifiziert wurden. Die für die Verifikation wichtigen Parameter wurden in Kapitel 4 erläutert. Hierbei wurden insgesamt 38 Hypothesen über die Parameter aufgestellt, um eine Hilfe für die Wahl der richtigen Parameter zu stellen. Diese Hypothesen sollen in diesem Kapitel überprüft werden. Zusätzlich wird ein Modell berechnet, welches die Auswirkung der einzelnen Optionen auf den Beweisaufwand unabhängig von den Hypothesen betrachten soll. Mit Hilfe dieser Informationen sollen im Anschluss Empfehlungen für den Aufbau der Oberfläche und Bedienung von KeY und für die Standardeinstellungen getroffen werden.

Um die Hypothesen zu überprüfen, muss zunächst ein Experiment entworfen werden. Hierzu sollen statistische Tests durchgeführt werden. Für diese Tests müssen zunächst Daten ermittelt werden müssen. In diesem Kapitel sollen die Art der Experimente, die Ermittlung der Daten und die Durchführung der Tests erläutert werden. Als Grundlage hierfür werden die im Rahmen dieser Arbeit erstellten Spezifikationen genutzt, die teilweise in Kapitel 3 vorgestellt wurden. Da die dadurch entstehenden Experimente zu umfangreich sind, um diese im Rahmen dieser Arbeit manuell auszuführen, wurde ein Controller entwickelt, der KeY ansteuert. Dieser soll der Vollständigkeit halber ebenfalls kurz vorgestellt werden. Die Spezifikationen, der entwickelte KeY-Controller, die ermittelten Daten und die Skripte zur Auswertung können unter https://github.com/NeoRoy/KeYExperienceReport eingesehen werden.

Der Entwurf des Experiments wird in Abschnitt 5.1 erläutert. Dies umfasst die Identifikation der Variablen, die Umformung der jeweiligen Forschungshypothesen in statistische Hypothesen, die Auswahl des richtigen Tests und die Beschreibung der Durchführung der Experimente. Anschließend wird in Abschnitt 5.2 auf die Implementierung des Controllers eingegangen, um in Abschnitt 5.3 die Ergebnisse der Experimente vorzustellen. Die Daten werden in Abschnitt 5.4 genutzt, um ein Beweisaufwandsmodell zu erstellen. Danach soll die Gültigkeit der Resultate in

https://dx.doi.org/10.24355/dbbs.084-201801311013

Abschnitt 5.5 diskutiert werden. Abschließend werden die in diesem Kapitel gewonnenen Ergebnisse in Abschnitt 5.6 zusammengefasst betrachtet.

5.1 Entwurf der Experimente

Die Hypothesen sollen in einzelnen Experimenten untersucht werden. Sie beziehen sich dabei entweder auf die Beweisbarkeit oder den Beweisaufwand in KeY. Um das Experiment zu entwerfen, müssen zunächst die entsprechenden Variablen identifiziert werden. Anschließend sollen die Art des jeweiligen Experiments, die Art der Durchführung und die entsprechenden Rahmenbedingungen definiert werden.

Auswahl der Variablen

Bei der Beweisbarkeit handelt es sich jeweils um die Frage, ob KeY eine bestimmte Methode in einem Spezifikationsfall mit Hilfe einer gewissen Konfiguration automatisch verifizieren kann. Aus dieser Aussage können bereits die Variablen für die Experimente bezüglich der Beweisbarkeit abgeleitet werden. Die abhängige Variable, die im Weiteren betrachtet werden soll, ist hierbei die boolsche Aussage, ob der Beweis geschlossen werden konnte oder nicht. Als unabhängige Variablen können die Faktoren betrachtet werden, mit denen KeY konfiguriert wird. Dies umfasst die Kombination aus Methode und Spezifikationsfall sowie die Parameter, die untersucht werden sollen.

Bei den Aussagen über den Beweisaufwand bietet sich ein sehr ähnliches Bild. Zeit ist die relevanteste Kenngröße für die Anwendbarkeit und Skalierbarkeit eines Beweises, allerdings ist die Dauer des Beweises direkt abhängig von dem System, auf dem der Beweis ausgeführt wird und von anderen Prozessen, die auf dem System ausgeführt werden. Dadurch gäbe es viele externe Variablen, die nicht direkt kontrolliert werden können, was die Güte des Ergebnisses senkt. Das Gleiche gilt auch für die benötigte Rechenleistung, bei der zwar das ausführende System weniger Einfluss nehmen kann, jedoch hat die ausführende virtuelle Maschine noch einen externen Einfluss auf diesen Wert. Die Folge wäre, dass jeder Versuch in beiden Fällen mehrfach wiederholt werden müsste, um geeignete Mittelwerte zu erhalten. Da die Anzahl der Beweisknoten ein intern in KeY ermittelter Wert ist, spielen hier weniger externe, unerwünschte Einflüsse eine Rolle, die das Resultat verfälschen können. Der Nachteil an den Beweisknoten ist allerdings, dass sich hinter einem Knoten ein variabler, zeitlicher Aufwand verbergen kann. Um den Einfluss von Störfaktoren auf das Ergebnis der Experimente zu minimieren, wird trotzdem die Anzahl der Beweisknoten als abhängige Variable für den Beweisaufwand verwendet.

Neben den abhängigen Variablen, müssen noch die unabhängigen Variablen identifiziert werden. Alle im Rahmen dieser Arbeit aufgestellten Hypothesen beziehen sich jeweils auf einen einzelnen Parameter. Dieser kann jeweils zwei oder mehr verschiedene Optionen annehmen, allerdings beschreiben die Hypothesen meist den Zusammenhang zwischen genau zwei Optionen. Die Frage ist jeweils, wie sich die Verwendung der betrachteten Option auf einen Beweis auswirkt. Da eine Aussage über die Auswirkung der Optionen getroffen werden soll und sich diese in KeY steuern lassen, können sie für die Versuche als unabhängige Variablen dienen.

https://dx.doi.org/10.24355/dbbs.084-201801311013

Auswahl der Stichproben

Die Frage ist allerdings, wie diese im Rahmen der Experimente gewählt werden sollen. Eine Möglichkeit hierbei wäre es, eine Konfiguration zu erstellen und die zu untersuchenden Optionen jeweils einmal einzusetzen. Bei diesem Ansatz könnten jedoch Abhängigkeiten, oder auch Feature-Interaktionen, zwischen den einzelnen Parametern untergehen. Ein Parameter betrifft jeweils die Anwendung einzelner Regeln auf den Beweis. Da alle Parameter für die Verifikation eingesetzt werden, ist es naheliegend, dass die Anwendung oder das Fehlen einzelner Regeln die Anwendung von Regeln anderer Parameter beeinflussen kann. Bei der Wahl der Grundkonfiguration können ebenfalls Fehler entstehen. Diese Wahl müsste zufällig erstellt werden, da sie sonst gegen das Prinzip der Zufälligkeit für empirische Untersuchungen verstoßen würde [WRH+00, S. 94]. Und wenn diese eine, zufällig erstellte Konfiguration fehlerhaft ist, könnte dies die Resultate erneut verfälschen.

Eine Alternative wäre das Testen aller möglichen Kombinationen. Dies würde das Problem mit den Abhängigkeiten lösen, jedoch die Anzahl der benötigten Ausführungen stark steigern. Es lassen sich trotz der im Rahmen dieser Arbeit getroffenen Einschränkungen 1.990.656 gültige Konfigurationen bilden. Diese müssten allerdings nicht für jede Methode eingesetzt werden, da sie sowohl die Konfigurationen für mit *intRules:javaSemantics* zu verifizierende Methoden (663.552), als auch für mit *intRules:arithmeticSemanticsIgnoringOF* zu verifizierende Methoden (1.327.104) (siehe Beschreibung 4.21), umfassen. Dies als Grundlage zu verwenden würde den zeitlichen Rahmen dieser Arbeit überschreiten. Es muss also ein Kompromiss gefunden werden, der sowohl Abhängigkeiten erkennt, als auch eine überschaubare Anzahl an nötigen Verifikationen bietet. Einen solchen Kompromiss bietet das *t-wise-sampling*. Abhängig vom t-Wert werden hierbei entweder Abhängigkeiten höherer Ordnung erkannt (größere t-Werte) oder weniger Verifikationen benötigt (kleinere t-Werte) [MKR+16, S. 4f]. Um die Anzahl der Versuche gering zu halten, wird $t = 2$ verwendet. Mit Hilfe dieses Vorgehens konnte die Anzahl der für diese Arbeit insgesamt nötigen Konfigurationen auf insgesamt 1084 gesenkt werden (545 für *intRules:javaSemantics* und 539 für *intRules:arithmeticSemanticsIgnoringOF*).[1]

Um die zu testenden Options-Kombinationen zu ermitteln, werden Techniken aus dem Bereich der Software-Produktlinien eingesetzt. Hierzu werden die Abhängigkeiten zwischen den Optionen mit Hilfe von *FeatureIDE* als Feature-Diagramm modelliert [MTS+17, S. 43]. Die Parameter und Optionen werden hierbei als Features und die entstehenden Konfigurationen als Produkte abgebildet. Für die Erstellung der Modelle werden die in Abschnitt 4.1 und Abschnitt 4.2 erstellten Diagramme als Grundlage verwendet. Diese wurden jedoch noch weiter eingeschränkt, um möglichst wenig fehlerhafte oder unnötige Konfigurationen zu testen. So konnte im Rahmen von Vorversuchen festgestellt werden, dass KeY in der vorliegenden Version bei dem Einsatz eines `Object`-Arrays einen Fehler wirft, wenn *initialisation:enableStaticInitialisation* verwendet wird, was in allen Verträgen der `ArrayList` der Fall ist. Außerdem soll der in KeY 2.7 hinzugekommene *Merge Point Statements-*

[1] Es erscheint im ersten Moment eigenartig, dass mehr Konfigurationen für *intRules:javaSemantics* als für *intRules:arithmeticSemanticsIgnoringOF* erzeugt wurden. Dies ist allerdings darauf zurückzuführen, wie die Konfigurationen erzeugt wurden und wie viele Überschneidungen es dadurch in den Experimenten der einzelnen Hypothesentests gibt.

https://dx.doi.org/10.24355/dbbs.084-201801311013

Parameter nicht mit untersucht werden (siehe Beschreibung 4.8), weshalb dieser mit *None* festgelegt wird. Dies trifft auch auf das *Permissions*-Taclet zu (siehe Beschreibung 4.32). Da das *IntRules*-Taclet im Rahmen dieser Arbeit abhängig von der zu verifizierenden Methode festgelegt werden soll, müssen zwei verschiedene Feature-Diagramme erzeugt werden (siehe Abschnitt 3.3.5 und erneut Beschreibung 4.21). Eins setzt *IntRules: ArithmeticSemancticsIgnoringOF* voraus, das andere *IntRules: javaSemantics* (siehe Abschnitt A.4).

Die beiden so erstellten Diagramme werden für jedes Experiment kopiert und der jeweils betrachtete Parameter wird mit einer der zu untersuchenden Optionen festgelegt. Anschließend wird die *t-wise-sampling*-Funktion von *FeatureIDE* genutzt, um die Konfigurationen zu generieren [MTS+17, S. 92f].[2] Für die Generierung wird die Anzahl der Interaktionen auf 2 festgelegt, da dies dem zuvor definierten t-Wert entspricht. Die anderen Optionen werden auf dem Standardwert belassen, da diese weiteren Optionen alle den zuvor aufgestellten Forderungen entsprechen. Die so erzeugten Konfigurationen werden für jede zu betrachtende Option des Parameters kopiert, so dass jede Konfiguration mit jeder zu betrachtenden Option existiert. Dabei kann es bei dem *IntegerSimplificationRules*-Parameter zu der Erzeugung von Konfigurationen kommen, die laut dem Modell ungültig sind. Diese sollen dennoch für die Erzeugung der Daten mit genutzt werden und gegebenenfalls zu einem späteren Zeitpunkt aussortiert werden. Dadurch ist sichergestellt, dass genügend Datensätze für die Untersuchung der jeweiligen Hypothesen zur Verfügung stehen und zudem kann die Einschränkung bezüglich des Parameters (nicht *IntegerSimplificationRules:Minimal* oder nicht *IntRules:javaSemantics*) separat betrachtet werden.

Auswahl der Signifikanztests und der Testarten

Anschließend soll die Hypothese an Hand der ermittelten Daten empirisch untersucht werden. Hierzu müssen passende Tests ausgewählt werden. Da einmal ein Boolean und einmal ein Integer als abhängige Variable verwendet wird, sollen verschiedene Tests verwendet werden. Für den Integer, also den Beweisaufwand, soll ein *T-Test* durchgeführt werden. Da die gleichen Versuche immer jeweils für beide Optionen durchgeführt wurden, kann auf den *paarweisen T-Test* zurückgegriffen werden [WRH+00, S. 140f]. Für die Beweisbarkeit, also den Boolean, wären für belastbare Ergebnisse sehr viele Datensätze notwendig [Row16, S. 257]. Für kategorische Werte, wie es bei einem Boolean der Fall ist, und paarweise Daten, kann der *McNemar-Test* angewendet werden [Row16, S. 277]. Sollte sich die Hypothese nicht auf den Vergleich zwischen zwei Optionen beziehen, sondern nur eine einzelne Option beschreiben (siehe Hypothese 27), wird ein normaler *T-Test* verwendet [WRH+00, S. 138]. Hierbei wird auf einen *T-Test* zurückgegriffen, da er auch für den Test auf eine erwartete Verteilung mit definiertem Erwartungswert geeignet ist. Hierbei muss allerdings der Erwartungswert für diese Option festgelegt werden, da kein Vergleich mit einer anderen Option als Grundlage dienen kann. Ansonsten ist das Vorgehen äquivalent zu dem des *paarweisen T-Tests*. Als Signifikanzniveau wird jeweils $\alpha = 0,05$ verwendet.

Für einen Signifikanztest gibt es jeweils zwei Arten von Hypothesen. Eine Nullhypothese und eine Alternativhypothese. In der Literatur finden sich verschiedene

[2] Es wurde *FeatureIDE* Version 3.3.0. verwendet

https://dx.doi.org/10.24355/dbbs.084-201801311013

Definitionen für Nullhypothesen. Es gibt Quellen, die definieren, dass eine Nullhypothese immer besagt, dass Unterschiede zwischen den Daten rein zufällig entstanden sind [WRH+00, S. 91][Row16, S. 99]. Eine andere Quelle sagt nur aus, dass die Nullhypothese von einer gewissen Verteilung ausgeht, spezifiziert diese aber nicht [Rü10, S. 6]. Hier wird allerdings explizit erwähnt, dass Nullhypothese und Alternativhypothese nicht grundsätzlich vertauscht werden können, da für viele Hypothesen kein geeigneter Test existiert [Rü10, S.22]. Der *paarweise T-Test* nutzt als Nullhypothese die Aussage, dass der erwartete Durchschnitt der Unterschiede 0 beträgt [WRH+00, S. 144]. Auch der *McNemar-Test* geht bei seiner Nullhypothese von keinem systematischen Trend aus [Row16, S. 178].

Der Großteil der Hypothesen vergleicht zwei Optionen. Zunächst soll das Vorgehen für diese Fälle vorgestellt werden, um anschließend auf die Hypothesen einzugehen, die nur eine Option betrachten. Für die Fälle, in denen die Hypothesen von einem Unterschied zwischen den betrachteten Optionen ausgehen, wird die betrachtete Hypothese jeweils zu einer Alternativhypothese umformuliert. Als Nullhypothese wird dann die Annahme verwendet, dass keine Korrelation zwischen den zu betrachtenden Optionen und der abhängigen Variablen besteht. Sollte die Hypothese von keinem Unterschied ausgehen, muss dies sowohl beim *T-Test*, als auch beim *McNemar-Test* als Nullhypothese verwendet werden. In diesem Fall wird allerdings nur geprüft, ob die Hypothese an Hand der Stichprobe beibehalten werden darf oder abgelehnt werden muss, sie kann nicht angenommen werden.

Der *T-Test* liegt sowohl als beidseitiger, als auch als einseitiger Test vor. Dies gilt allerdings nicht für den *McNemar-Test*, da dieser nur als beidseitiger Test vorliegt. Für einen einseitigen Test wird deshalb zunächst ein beidseitiger Test auf Signifikanz gemacht und die Einseitigkeit anschließend separat betrachtet, sofern das Ergebnis signifikant ist. Hierbei wird das Resultat in einer Vierfeldertafel dargestellt, um zu prüfen, ob kein Wert aufgetreten ist, der für die Alternativhypothese spricht. Bei der Prüfung auf signifikanz des Ergebnisses muss allerdings beachtet werden, dass der P-Wert des Tests sich auf einen beidseitigen Test bezieht und deshalb halbiert werden muss [Row16, S. 85f].

Es kommt im Rahmen der Untersuchung auch vor, dass Hypothesen Aussagen verwenden, die sich durch *mindestens* oder *höchstens* ausdrücken lassen. Dies ist Beispielsweise bei Hypothese 4 der Fall. In diesem Fall wäre die korrekte Vorgehensweise, dass in diesem Fall die Nullhypothese untersucht wird, bei der die Alternativhypothese entsprechend angepasst wird. Wenn der Beweisaufwand von Option_1 *mindestens* so groß wie der von Option_2 sein soll, lautet die Alternativhypothese, dass Option_1 kleiner als Option_2 ist. Mit diesem Vorgehen kann die Hypothese jedoch nicht angenommen, sondern höchstens abgelehnt werden. Deshalb wird für solche Hypothesen zusätzlich noch eine stärkere Hypothese geprüft: Ob Option_1 einen höheren Beweisaufwand als Option_2 hat. Da diese Hypothese eine einseitige Alternativhypothese ist, könnte die Nullhypothese abgelehnt und dadurch die Alternativhypothese angenommen werden.

Die im Rahmen dieser Arbeit spezifizierten Methoden werden nun in KeY mit Hilfe der erzeugten Konfigurationen verifiziert. Hierfür wird ein selbst entwickeltes Werk-

https://dx.doi.org/10.24355/dbbs.084-201801311013

zeug eingesetzt (siehe Abschnitt 5.2). Die hier ermittelten Resultate werden anschließend mit Hilfe der Software *R* Version 3.1.2 dem jeweiligen Test unterzogen [rPr].[3]

Umwandlung der Forschungshypothesen in statistische Hypothesen

Die auszuführenden Experimente werden in Tabelle 5.1 dargestellt. Die Tabelle besteht aus sechs Spalten. Die erste und die zweite Spalte beziehen sich auf die jeweils zu prüfende Foschungshypothese: Die Hypothesennummer und die Art des Tests. Die weiteren Spalten beziehen sich auf die jeweiligen Hypothesen, die im Rahmen des statistischen Tets überprüft werden sollen. Diese lässt sich jeweils über die Experimentnummer identifizieren, die in der vierten Spalte steht. Die fünfte und sechste Spalte enthalten jeweils die überprüften Optionen. Wenn die Hypothese eine Aussage im Sinn von *mindestens* oder *höchstens* verwendt, wird die Zeile an dieser Stelle aufgeteilt. Diese Hypothesen sollen auf zwei unterschiedliche Art und Weisen untersucht werden, weshalb der jeweilige Eintrag in der siebten und achten Spalte unterteilt wird. Die siebte Spalte stellt dar, ob die Prüfung der Nullhypothese H_0 oder die Prüfung einer Alternativhypothese H_A beschrieben wird. Da für einen *McNemar-Test* nur ein beidseitiger Test durchgeführt wird, ergäbe die Unterteilung keinen Sinn. Sollte das Ergebnis eines *McNemar-Tests* signifikant sein, muss dieses separat betrachtet werden. Für einen solchen Test wird als Schreibweise $H_{\leq}$ für *höchstens* und $H_{\geq}$ für *mindestens* verwendet. In der achten Spalte wird jeweils notiert, welche Art der Alternativhypothese für den Test verwendet wird. $<>$ steht hierbei für einen beidseitigen Test, $<$ und $>$ stellen einseitige Tests dar. Bei der Verwendung von $<$ muss Option_1 kleiner als Option_2 sein, bei $>$ größer.

Hypothese	Art des Tests	Parameter	Experimentnummer	1. betroffene Option	2. betroffene Option	Art der Hypothese	Alternativhypothese
1	M	Stop At	1	Default	Unclosable	H_0	$<>$
2	T	Stop At	2	Default	Unclosable	H_0	$<>$
3	M	One Step Simplification	3	Enabled	Disabled	H_0	$<>$
4	T	One Step Simplification	4	Enabled	Disabled	H_0	$>$
						H_A	$<$
5	M	Proof Splitting	5	Delayed	Free	$H_{\leq}$	$>$
6	T	Proof Splitting	6	Delayed	Free	H_0	$>$
						H_A	$<$
7	M	Proof Splitting	7	Off	Free	$H_{\leq}$	$>$
			8	Off	Delayed	$H_{\leq}$	$>$
8	T	Proof Splitting	9	Off	Free	H_0	$>$
						H_A	$<$
			10	Off	Delayed	H_0	$>$
						H_A	$<$

[3] Es wurde *R* Version 3.1.2 verwendet.

https://dx.doi.org/10.24355/dbbs.084-201801311013

9	M	Loop Treatment	11	Invariant	Loop Scope Invariant	H_0	<>
10	T	Loop Treatment	12	Invariant	Loop Scope Invariant	H_0	<
						H_A	>
11	M	Dependency Contracts	13	On	Off	H_0	<>
12	T	Dependency Contracts	14	On	Off	H_0	<>
13	M	Query Treatment	15	On	Restricted	H_0	<>
			16	On	Off	H_0	<>
14	T	Query Treatment	17	On	Restricted	H_0	<>
			18	On	Off	H_0	<>
15	M	Query Treatment	19	Off	Restricted	$H_\leq$	>
			20	Resticted	On	$H_\leq$	>
16	T	Query Treatment	21	Restricted	On	H_A	<>
17	T	Expand local queries	22	On	Off	H_0	<
						H_A	>
18	M	Expand local queries	23	On	Off	$H_\leq$	>
19	M	Arithmetic Treatment	24	Basic	DefOps	$H_\leq$	>
20	M	Arithmetic Treatment	25	DefOps	ModelSearch	H_A	<>
21	M	Quantifier Treatment	26	None	No Splits	H_0	<>
			27	None	No Splits With Progs	H_0	<>
			28	None	Free	H_0	<>
22	T	Quantifier Treatment	29	None	No Splits	H_0	<>
			30	None	No Splits With Progs	H_0	<>
			31	None	Free	H_0	<>
23	M	Quantifier Treatment	32	None	No Splits	$H_\leq$	>
			33	No Splits	No Splits With Progs	$H_\leq$	>
			34	No Splits With Progs	Free	$H_\leq$	>
24	T	Quantifier Treatment	35	Free	No Splits With Progs	H_0	<
						H_A	>
			36	No Splits With Progs	No Splits	H_0	<
						H_A	>
			37	No Splits	None	H_0	<
						H_A	>
25	M	Class Axiom Rule	38	Free	Delayed	H_0	<>
			39	Free	Off	H_0	<>
26	T	Class Axiom Rule	40	Free	Delayed	H_0	<>
			41	Free	Off	H_0	<>
27		Class Axiom Rule	Wird separat betrachtet				
28	M	Class Axiom Rule	53	Off	Delayed	H_0	<>
29	M	Strings	42	On	Off	$H_\geq$	<
30	T	Strings	43	On	Off	H_0	<>
31	M	BigInt	44	On	Off	$H_\geq$	<

https://dx.doi.org/10.24355/dbbs.084-201801311013

32	T	BigInt	45	On	Off	H_0	<>
33	M	IntegerSimplificationRules	46	Full	Minimal	$H_\geq$	<
34	T	IntegerSimplificationRules	47	Full	Minimal	H_0	<>
35	M	Sequences	48	On	Off	$H_\geq$	<
36	T	Sequences	49	On	Off	H_0	<>
37	M	MoreSeqRules	50	On	Off	$H_\geq$	<
38	T	MoreSeqRules	51	On	Off	H_0	<>

Tabelle 5.1: Darstellung der Experimente

Die Hypothese 27 betrachtet nicht den Vergleich von zwei Hypothesen, sondern sagt aus, dass bestimmte Spezifikationsfälle nicht geschlossen werden können. Wenn ein offener Beweis als 0 und ein geschlossener Beweis als 1 betrachtet wird, wird als Erwartungswert für den *T-Test* 0 vorausgesetzt.

Einschränkung der zu verwendenden Stichproben

Nicht für alle Hypothesen sollen alle verfügbaren Daten genutzt werden. Einzelne Hypothesen schränken die beschriebenen Szenarien ein. Beispielsweise betrachten Hypothese 11 und Hypothese 12 nur Spezifikationsfälle, die keine *accessible*-Klauseln verwenden. Da diese im Rahmen dieser Arbeit nicht verwendet wurden, spielt diese Einschränkung jedoch keine Rolle. Hypothese 13 und Hypothese 14 schließen Verträge mit Queries aus. Dies trifft auf die Vertäge der `Math`-Klasse und die der Methoden `ArrayList`, `isEmpty`, `size`, `rangeCheck`, `outOfBoundsMsg` und `isEmpty` der `ArrayList` zu. Auch enthalten die leichtgewichtigen Spezifikationsfälle der `ArrayList`-Methoden `clear`, `contains`, `remove` und `add` keine Queries. Viele Spezifikationsfälle, wie die schwergewichtigen Fälle der `contains`-Methode verwenden das `elements`-Modellfeld, welches mit einer Query belegt wird und werden dadurch ausgeschlossen. Die Hypothese 15 bezieht sich nur auf Spezifikationsfälle, bei denen sich alle Queries auch mit der Option *Method Treatment Contract* schließen lassen. Dies trifft grundsätzlich auf alle Spezifikationsfälle zu, die keine Query enthalten. Der schwergewichtige Spezifikationsfall der `clear`-Methode verwendet nur die `isEmpty`-Methode und eignet sich deshalb ebenfalls. Die übrigen Queries laufen irgendwann auf `Arrays.copyOf` hinaus und eignen sich deshalb nicht. Für die Hypothese 27 können alle Spezifikationsfälle der `ArrayList` verwendet werden, da diese auf Invarianten zurückgreifen und oftmals Schreibzugriffe nutzen. Alle anderen Spezifikationsfälle können für die Überprüfung der Hypothese 28 genutzt werden. Diese Spezifikationsfälle greifen auch nicht auf Invarianten oder Klassenfelder zu und können deshalb für die Untersuchung von Hypothese 25 und Hypothese 26 eingesetzt werden.

Auch Hypothese 23 und Hypothese 24 schränken die betrachteten Spezifikationsfälle ein. Sie beziehen sich nur auf Formeln ohne Quantifizierer. Die Annahme bei der Erstellung dieser Hypothese ist, dass eine Formel dann keinen Quantifizierer erhält, wenn der Beweis mit der *Quantifier Treatment None* geschlossen werden kann. Es stellte sich jedoch heraus, dass die `assignable`-Klausel über einen Quantifizierer

https://dx.doi.org/10.24355/dbbs.084-201801311013

umgesetzt wird und deshalb jeder Beweis eine solche enthält. Die Beweise können dennoch teilweise geschlossen werden, da der Quantifizierer nicht immer ausgewertet werden muss. Der Beweis kann in diesen Fällen mit Hilfe des *allRight*-Taclets geschlossen werden. Diese Hypothesen werden deshalb abgelehnt.

5.2 Entwicklung des KeY-Controllers

Da im Rahmen dieser Arbeit eine Vielzahl einzelner Verifikationen mit Hilfe von KeY durchgeführt werden müssen, ist es nicht zielführend, dies manuell zu machen. Für die Verarbeitung von vielen Verifikationen gibt es das Werkzeug *MonKeY*, dieses dient jedoch nicht zur Ausführung der gleichen Verifikation mit verschiedenen Konfigurationen [TSAH12, S.16f]. Von daher wird im Rahmen dieser Arbeit ein eigenenes Werkzeug zur Bedienung von KeY entwickelt.

Das Werkzeug muss dafür sorgen, dass KeY die Verifikation mit unterschiedlichen Parametern unabhängig von vorherigen Verifikationen vornimmt. Sollten die Verifikationen nicht unabhängig von einander durchgeführt werden, könnte eine vorherige Verifikation die folgenden beeinflussen. Eine solche Beeinflussung könnte beispielsweise auftreten, falls KeY Informationen aus vorherigen Entscheidungen übernimmt. Außerdem wurde im Rahmen dieser Arbeit auch festgestellt, dass eine wiederholte Ausführung eines Beweises den verfügbaren Heap immer weiter befüllt, was die Leistung der virtuellen Maschine negativ beeinflusst.[4] Da nur die Anzahl der Knoten als Variable für die Experimente bezüglich des Beweisaufwandes verwendet wird, werden die Daten nicht verfälscht. Dennoch ist eine kürzere Versuchsdauer und ein geringerer Ressourcenaufwand für die Evaluierung von Vorteil, da so mehr Versuche im Rahmen dieser Arbeit durchgeführt werden können. Aus diesem Grund soll das Werkzeug nach dem Ausführen eines Beweises die virtuelle Maschine beenden.

Um mehrere Beweise parallel auszuführen wird auch auf eine Server-Client-Architektur zurückgegriffen, bei welcher der Server die Verwaltung der auszuführenden Beweise und der Ergebnisse übernimmt, während die Clients die Verifikation an sich übernehmen. Der Server liest nach dem Start einmal eine *XML*-Datei ein, in der definiert wird, welche Methoden mit welchen Konfigurationen verifiziert werden sollen. Diese werden an die Clients verteilt, wo die Verifikation ausgeführt wird. Nach der Ausführung einer einzelnen Verifikation, startet der Client einen neuen Client und beendet sich dann selbst. Durch das Beenden der virtuellen Maschine ist sie für jede Verifikation zurückgesetzt und konsekutive Ausführungen beeinflussen sich nicht direkt. Die direkte Schnittstelle zu KeY wurde ebenfalls im Rahmen dieser Arbeit implementiert und basiert auf einem Beispiel aus dem KeY-Quellcode.[5]

5.3 Auswertung der Experimente

In diesem Abschnitt sollen die Resultate der Experimente dargestellt und diskutiert werden. Hierfür werden diese zunächst gesammelt dargestellt (siehe Tabelle 5.2) um anschließend darauf einzugehen, welche Informationen sich daraus gewinnen lassen. Für die Ermittlung der Daten wurden auf einer virtuellen Maschine mit 16 CPUs und

[4]Dies lässt auf ein Speicherleck in KeY schließen (siehe Abschnitt A.5).
[5]das `key.core.example`-Projekt

https://dx.doi.org/10.24355/dbbs.084-201801311013

64 Gigabyte Arbeitsspeicher 4 Clients über 4 Wochen hinweg mit jeweils ausgeführt, die insgesamt 12505 Experimente ausgeführt haben.[6]

Die erste Spalte der Resultattabelle stellt die Nummer der zu überprüfenden Forschungshypothese dar, wie auch in Tabelle 5.1. Die Experimentnummer kann in Spalte 2 gesehen werden, gefolgt von der Art der Hypothese, die der Forschungshypothese entspricht: H_0 für die Nullhypothese, H_A für die Alternativhypothese und $H_\leq$ und $H_\geq$ für die *mindestens* und *höchstens* Sonderfälle. Der P-Wert aus R steht in der vierten Spalte. Abhängig von der Art der Hypothese steht in der letzten Spalte das Ergebnis des Resultats. Wenn die Forschungshypothese in die Nullhypothese umgewandelt wird, kann die Hypothese entweder abgelehnt werden oder sie bleibt weiter bestehen. Sollte es sich um die Alternativhypothese handeln, kann sie entweder angenommen werden oder die Hypothese bleibt bestehen.

Bei der statistischen Auswertung der Hypothesen sind vier Experimente aufgefallen, bei denen eine manuelle Prüfung des Ergebnisses nötig ist (siehe Tabelle 5.3). Aus den Vierfeldertafeln geht hervor, dass die Einseitigkeit des Tests jeweils die Hypothesen unterstützt. Da die Ergebnisse in diesen Fällen signifikant sind, können die Hypothesen angenommen werden. Insgesamt werden also 2 Forschungshypothesen abgelehnt, 5 angenommen und die übrigen 31 bleiben bestehen - zwei von ihnen konnten jedoch teilweise angenommen werden.

Hypothese 21 und Hypothese 22 wurden hierbei abgelehnt, da es keine Spezifikation für Java-Quelltext gibt, die keinen Quantifizierer enthält. Im Rahmen der Experimente ist jedoch aufgefallen, dass sich teilweise Beweise mit der Option *Quantifier Treatment None* schließen lassen. Die zusätzliche Auswahl von *Proof Splitting Off* hat die Schließbarkeit in vielen Fällen noch weiter gesenkt.

Aus Hypothese 1 und Hypothese 2 folgt, dass diese Option für einen vollautomatischen Beweis nicht beachtet werden muss. In Abbildung 5.1 wird die Differenz zwischen den Optionen *Default* und *Unclosable* des *Stop At*-Parameters dargestellt. Dies lässt vermuten, dass der Aufwand für einen nicht schließenden Beweis mit *Default* höher ist als mit *Unclosable*, weshalb *Unclosable* vorzuziehen wäre. Dies entspricht nicht der Standardauswahl des Parameters: *Default*.

Aus Hypothese 3 und Hypothese 4 lässt sich schließen, dass die Option *Disabled* des Parameters *One Step Simplification* bei einer automatischen Verifikation keinen Vorzug *Enabled* gegenüber bietet. Zu *Loop Treatment* kann angenommen werden, dass der Aufwand von *Loop Scope Invariant* geringer ist, als der von *Invariant* (siehe Hypothese 10). Die Hypothese 9 kann zwar nicht angenommen werden, kann jedoch bestehen bleiben. Sie besagt, dass die Beweisbarkeit beider Optionen identisch ist. Solche Schlussfolgerungen lassen sich auch für die Parameter *Strings* (siehe Hypothese 29 und Hypothese 30), *BigInt* (siehe Hypothese 31 und Hypothese 32), *IntegerSimplificationRules* (siehe Hypothese 33 und Hypothese 34), *Sequences* (siehe Hypothese 35 und Hypothese 36) und *MoreSeqRules* (siehe Hypothese 37 und Hypothese 38) bilden. Auch bei Dependency Contracts konnte dies festgestellt werden

[6]Jedem Client standen hierbei 14 Gigabyte Arbeitsspeicher zur Verfügung. Der Server an sich verwendet 2 Intel Xeon E5-2640 v4 Prozessoren mit einer Taktfrequenz von 2.40GHz und jeweils 10 Prozessorkernen mit Hyperthreading.

https://dx.doi.org/10.24355/dbbs.084-201801311013

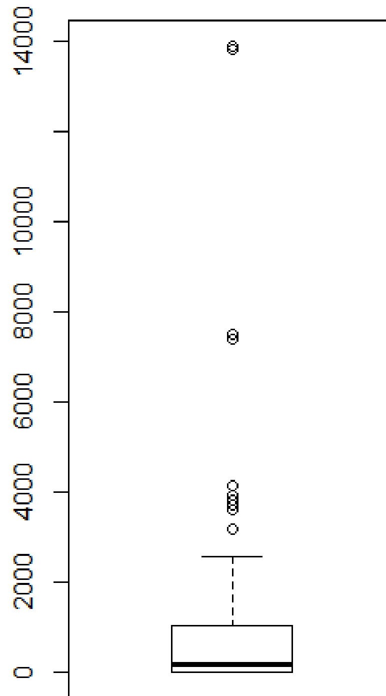

Abbildung 5.1: Boxplot der Differenzen in den Beweisaufwänden der *Stop At* Optionen (für offene Beweise)

Hypothese	Experimentnummer	Art der Hypothese	P-Wert	Forschungshypothese...
1	1	H_0	NA	bleibt bestehen
2	2	H_0	$8,2 * 10^{-2}$	bleibt bestehen
3	3	H_0	NA	bleibt bestehen
4	4	H_0	1	bleibt bestehen
		H_A	$1,187 * 10^{-7}$	**wird angenommen**
5	5	$H_{\leq}$	NA	bleibt bestehen
6	6	H_0	$9,997 * 10^{-1}$	bleibt bestehen
		H_A	$2,826 * 10^{-4}$	**wird angenommen**
7	7	$H_{\leq}$	$1,181 * 10^{-9}$	**wird angenommen***
	8	$H_{\leq}$	$1,181 * 10^{-9}$	**wird angenommen***
8	9	H_0	$7,622 * 10^{-1}$	bleibt bestehen
		H_A	$2,378 * 10^{-1}$	bleibt bestehen
	10	H_0	$2,465 * 10^{-1}$	bleibt bestehen
		H_A	$7,535 * 10^{-1}$	bleibt bestehen
9	11	H_0	NA	bleibt bestehen
10	12	H_0	$9,909 * 10^{-1}$	bleibt bestehen
		H_A	$9,122 * 10^{-3}$	**wird angenommen**

https://dx.doi.org/10.24355/dbbs.084-201801311013

11	13	H_0	NA	bleibt bestehen
12	14	H_0	$4,287 * 10^{-1}$	bleibt bestehen
13	15	H_0	NA	bleibt bestehen
	16	H_0	NA	bleibt bestehen
14	17	H_0	$9,278 * 10^{-2}$	bleibt bestehen
	18	H_0	$4,19 * 10^{-1}$	bleibt bestehen
15	19	$H_{\leq}$	$1,573 * 10^{-1}$	bleibt bestehen
	20	$H_{\leq}$	NA	bleibt bestehen
16	21	H_A	$2,935 * 10^{-2}$	**wird angenommen**
17	22	H_0	$9,34 * 10^{-1}$	bleibt bestehen
		H_A	$6,601 * 10^{-2}$	bleibt bestehen
18	23	$H_{\leq}$	$4,55 * 10^{-2}$	**wird angenommen***
19	24	$H_{\geq}$	$9,237 * 10^{-13}$	**wird angenommen***
20	25	H_A	$3,173 * 10^{-1}$	bleibt bestehen
21				ist fehlerhaft
22				ist fehlerhaft
23	32	$H_{\leq}$	$4,55 * 10^{-2}$	**wird angenommen***
	33	$H_{\leq}$	$1,573 * 10^{-1}$	bleibt bestehen
	34	$H_{\leq}$	NA	bleibt bestehen
24	35	H_0	$6,075 * 10^{-1}$	bleibt bestehen
		H_A	$3,925 * 10^{-1}$	bleibt bestehen
	36	H_0	$8,312 * 10^{-1}$	bleibt bestehen
		H_A	$1,688 * 10^{-1}$	bleibt bestehen
	37	H_0	$7,764 * 10^{-1}$	bleibt bestehen
		H_A	$2,236 * 10^{-1}$	bleibt bestehen
25	38	H_0	NA	bleibt bestehen
	39	H_0	NA	bleibt bestehen
26	40	H_0	NA	bleibt bestehen
	41	H_0	NA	bleibt bestehen
27	52	H_0	NA	bleibt bestehen
28	53	H_0	NA	bleibt bestehen
29	42	$H_{\geq}$	NA	bleibt bestehen
30	43	H_0	$3,805 * 10^{-1}$	bleibt bestehen
31	44	$H_{\geq}$	NA	bleibt bestehen
32	45	H_0	$3,49 * 10^{-1}$	bleibt bestehen
33	46	$H_{\geq}$	$5,32 * 10^{-4}$	**wird angenommen***
34	47	H_0	$4,011 * 10^{-1}$	bleibt bestehen
35	48	$H_{\geq}$	NA	bleibt bestehen
36	49	H_0	$1,629 * 10^{-1}$	bleibt bestehen
37	50	$H_{\geq}$	NA	**wird angenommen**
38	51	H_0	$1,579 * 10^{-1}$	bleibt bestehen

* nach manueller Prüfung

Tabelle 5.2: Resultate der Experimente

https://dx.doi.org/10.24355/dbbs.084-201801311013

Experiment 7 Proof Splitting		Off	
		Geschlossen	Offen
Free	Geschlossen	114	37
	Offen	0	142

Experiment 8 Proof Splitting		Off	
		Geschlossen	Offen
Delayed	Geschlossen	116	37
	Offen	0	143

Experiment 23 Expand local queries		On	
		Geschlossen	Offen
Off	Geschlossen	82	0
	Offen	4	105

Experiment 24 Query Treatment		Basic	
		Geschlossen	Offen
DefOps	Geschlossen	119	51
	Offen	0	164

Experiment 32 Quantifier Treatment		None	
		Geschlossen	Offen
No Splits	Geschlossen	132	4
	Offen	0	107

Experiment 46 IntegerSimplificationRules		Full	
		Geschlossen	Offen
Minimal	Geschlossen	122	0
	Offen	12	151

Tabelle 5.3: Darstellung der manuellen Prüfungen

(siehe Hypothese 11 und Hypothese 12), jedoch wurden nur Spezifikationsfälle ohne `accessible`-Klauseln betrachtet.

Die Hypothesen 5 bis 8 beziehen sich auf den Parameter *Proof Splitting*. Hierbei konnte jeweils eine Reihenfolge der Optionen bezüglich der Beweisbarkeit und des Beweisaufwands beschrieben werden. *Off* hat hierbei eine höchstens so gute Beweisbarkeit wie *Delayed*, welches ebenfalls eine höchstens so gute Beweisbarkeit wie *Free* hat. Für den Beweisaufwand lässt sich dies ebenfalls beobachten, jedoch in umgekehrter Reihenfolge. Die Hypothese, die den Zusammenhang bezüglich der Beweisbarkeit zwischen *Free* und *Delayed* bezieht, hat jedoch keinen bestimmbaren p-Wert gehabt. Die Daten waren also identisch.

Die Experimente zu Hypothese 15 bleiben nur bestehen, konnten also nicht angenommen werden. Hypothese 16 konnte jedoch angenommen werden. Demnach ist der Beweisaufwand bei der Verwendung von *Query Treatment Restricted* ein anderer als mit *On*.

https://dx.doi.org/10.24355/dbbs.084-201801311013

Hypothese 19 betrifft den Parameter *Arithmetic Treatment*. Laut der Hypothese ist die Beweisbarkeit von *DefOps* mindestens so groß, wie die von *Basic*. Die Hypothese zu *Model Search* (siehe Hypothese 20) bleibt nach wie vor bestehen, konnte jedoch nicht angenommen werden. Da diese in eine Alternativhypothese umgewandelt werden konnte, soll diese durch die nicht gegebene Signifikanz nicht weiter diskutiert werden.

Quantifier Treatment umfasst neben Hypothese 21 und Hypothese 22 zwei weitere Hypothesen, die nicht fehlerhaft sind. Hierbei wird jeweils eine Reihenfolge der Optionen, wie auch bei *Proof Splitting*, beschrieben. Die Reihenfolge geht hierbei von *None*, über *No Splits* und *No Splits with Progs* zu *Free*. *None* hat hierbei den geringsten Beweisaufwand, *Free* dafür die höchste Beweisbarkeit.

Unabhängig von den Hypothesen fällt auf, dass im Rahmen von Experiment 46 sehr viele Beweise geschlossen werden konnten, die *intRules:javaSemantics* und *integerSimplficationRules:Minimal* kombinieren. Das spricht dafür, dass die Einschränkung bezüglich dieser Abhängigkeit nicht benötigt wird. Allerdings kann auch kein Grund dafür erkannt werden, überhaupt *integerSimplificationRules:Minimal* zu wählen.

5.4 Berechnung eines Parameter-Modells

Im Rahmen der Experimente wurde eine große Anzahl an Datenpunkten ermittelt, die den Aufwand der Verifikation unter verschiedenen Konfigurationen darstellen. Es stellt sich die Frage, ob ein Modell gebildet werden kann, das den Beweisaufwand in Abhängigkeit der Optionen beschreibt. Das Ziel eines solchen Modells ist es, dass die Auswirkung der einzelnen Optionen auf den Verifikationsaufwand beschrieben wird. Hierbei soll ein Modell für alle Spezifikationsfälle erstellt werden und nicht eins pro Spezifikationsfall. Dies hat jedoch zur Folge, dass das Modell sehr allgemein sein wird und mit einem hohen Fehler zu rechnen ist. Eine solche Beschreibung stellt also eine grobe Orientierung und keine präzise Vorhersage des Aufwands dar. Ein Anwender soll mit diesem Modell eine Hilfestellung zu der Frage erhalten, mit der Anpassung welches Parameters er den größten Gewinn beim Beweisaufwand erzielen könnte.

Für die Erstellung des Modell soll *SPLConqueror* eingesetzt werden [SRK+12]. *SPLConqueror* wurde entwickelt, um bei Softwareproduktlinien die Auswirkungen einzelner Features auf nicht funktionale Eigenschaften zu ermitteln. Da die Konfiguration von KeY in Abschnitt 5.1 bereits als Feature-Modell dargestellt wurde, kann dieses verwendet werden. Hierbei werden die einzelnen Optionen von KeY jeweils als Feature und die Laufzeit als nicht funktionale Eigenschaft einer Konfiguration betrachtet. Eine solche Eigenschaft kann mit Hilfe von *SPLConqueror* in einem Modell abgebildet werden [SRK+12, S. 496].

Zunächst wird ein Featuremodell generiert, welches von *SPLConqueror* eingelesen werden kann. Dies ist weitestgehend äquivalent zu dem in FeatureIDE verwendeten Featuremodell, lässt allerdings eine Einschränkung außer acht: *IntRules Java_Semantics* $\Longrightarrow$ *IntegerSimplificationRules Full*. Diese kann nicht direkt übertragen werden, da die aktuelle Version von *SPLConqueror* diese Einschränkung nicht übersetzen konnte. Sie ließe sich allerdings umformen in ¬ *IntRules Java_Semantics*

https://dx.doi.org/10.24355/dbbs.084-201801311013

∨ *IntegerSimplificationRules Full*. Da *SPLConqueror* an Hand der Einschränkungen allerdings nur ungültige Datensätze aussortiert und im vorherigen Modell diese Einschränkung ohnehin angezweifelt wurde, kann von der Aufnahme der Einschränkung abgesehen werden. Außerdem werden die Parameter aus dem Modell entfernt, da die Auswirkungen der Optionen und nicht der Parameter untersucht werden sollen. Dass die Optionen eines Parameters sich gegenseitig ausschließen, bleibt jedoch Teil des Modells.

Für die Erstellung des Modells wurden zunächst alle Datensätze von nicht geschlossenen Beweisen ausgeschlossen. Um Ausreißer in dem Modell besser erkennen zu können, sollen mehrere Modelle berechnet werden. Hierzu wird *k-cross-validation* eingesetzt. Dieses Vorgehen soll normalerweise einen realistischeren Fehler für ein Modell erzeugen [K+95, S. 1138ff]. Da durch die Allgemeinheit des Modells mit einem großen Fehler zu rechnen ist, spielt der Fehler des Modells eine eher untergeordnete Rolle. Hierzu werden die Datensätze zufällig auf 10 Gruppen verteilt, von denen jeweils eine Gruppe für den Test und die andere für das Trainieren des Modells genutzt wird. Es werden 10 Gruppen gebildet, da dies die Instabilitäten zwischen den Modellen senken soll. Anschließend werden mit diesen Datensätzen in *SPLConqueror* 10 einzelne Modelle berechnet (siehe Abschnitt A.6).

Bei der Betrachtung der Modelle fällt zunächst auf, dass erwartete Optionen nicht auftauchen. Viele Optionen kommen in keinem der Modelle vor und wurden deshalb nicht mit aufgenommen. Die anderen Optionen werden in den Boxplots in Abbildung 5.2 dargestellt. Hierbei wird der jeweilige Faktor aus dem Modell auf der Y-Achse dargestellt. Die meisten dieser Optionen bieten abgesehen von dem Median 0 nur Ausreißer. Die Ausnahme stellen hierbei die Optionen *Arithmetic treatment:: Basic*, *Class axiom rule::Delayed*, *One Step Simplification::Disabled*, *Proof splitting::Delayed*, *Proof splitting::Free*, *Proof splitting::Off* und *Query treatment::-Restricted* dar.

Besonders interessant sind hierbei die Aussagen zu *Arithmetic treatment::Basic* und *Class axiom rule::Delayed*, da zu den Parametern dieser Optionen keine Hypothesen zu dem Beweisaufwand getroffen wurden. *Arithmetic treatment:: Basic* hat, bis auf einen Ausreißer, im Modell nur negative Faktoren. Dies entspricht der Aussage, dass *Basic* einen geringeren Beweisaufwand mit sich bringt, als die anderen Optionen. Der Median ist vom Betrag her allerdings kleiner als 100, was im Verhältnis zu beispielsweise *Class axiom rule::Delayed*, mit einem Median von über 1150, ein sehr geringer Wert ist. *Class axiom rule::Delayed* erhöht laut den Modellen den Beweisaufwand. Die anderen Optionen wurden bereits im Rahmen von Hypothesen betrachtet. Hierbei kann jedoch festgestellt werden, dass die Aussagen der Hypothesen sich jeweils mit den Aussagen des Modells decken (vergleiche Hypothese 4, Hypothese 6, Hypothese 8, Hypothese 16) .

Insgesamt gehen aus dem Modell also zwei neue Vermutungen hervor: Die Option *Arithmetic treatment::Basic* senkt den Beweisaufwand leicht, während die Option *Class axiom rule::Delayed* ihn stark erhöht. Außerdem gab es keine Widersprüche zu den bestehenden Hypothesen. Die Vermutungen können jedoch nicht im Rahmen dieser Arbeit untersucht werden, da sie aus der Betrachtung der Daten heraus entstanden sind.

https://dx.doi.org/10.24355/dbbs.084-201801311013

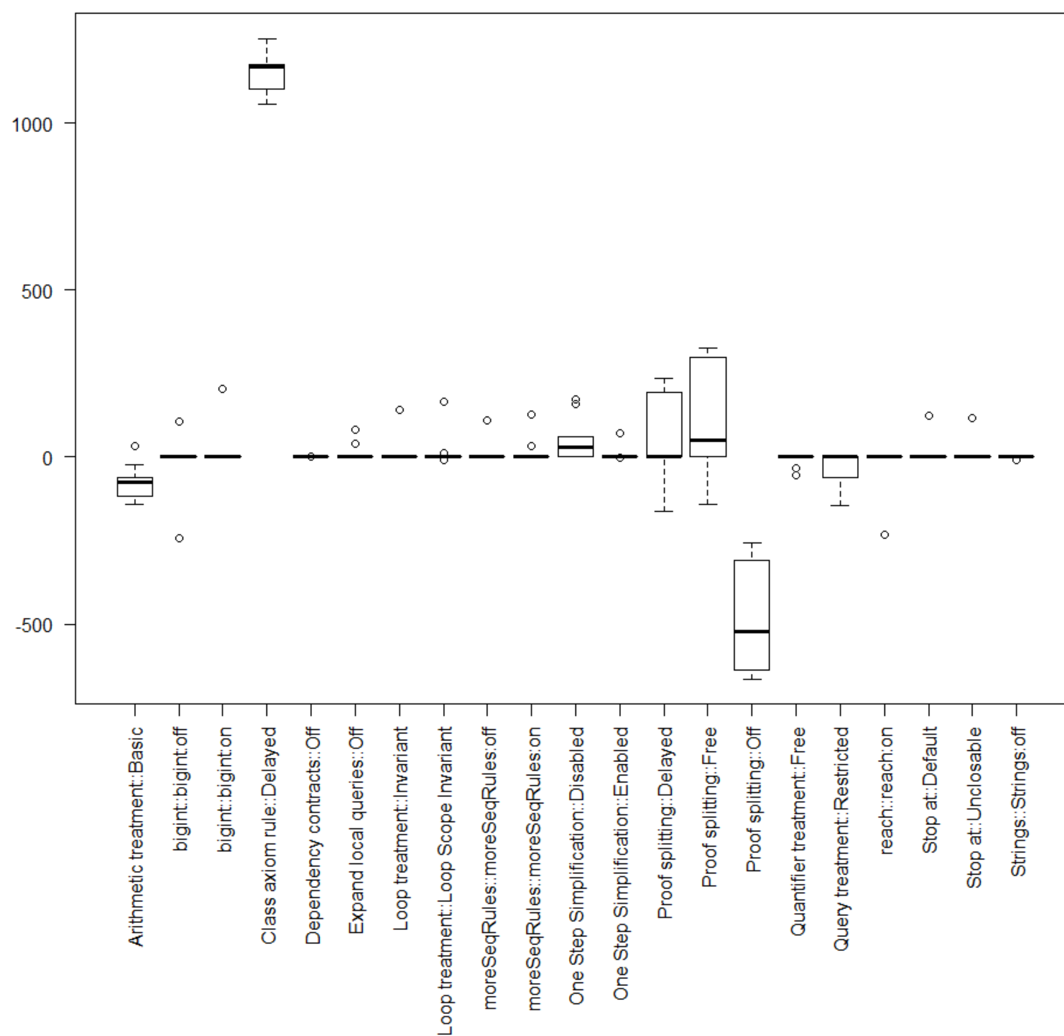

Abbildung 5.2: Auswertung der Variablen

https://dx.doi.org/10.24355/dbbs.084-201801311013

5.5 Gültigkeit der Resultate

In diesem Kapitel wurden statistische Tests durchgeführt. Diese sollen an dieser Stelle noch einmal auf ihre Gültigkeit hin diskutiert werden. Dies dient dazu, dass die Resultate der Hypothesentests noch einmal kritisch betrachtet werden und das Vorgehen retrospektiv hinterfragt wird. Um die Gültigkeit der Resultate zu betrachten, wird zunächst die interne und anschließend die externe Gültigkeit betrachtet [Onw00]. Die interne Gültigkeit bezieht sich hierbei darauf, dass die untersuchte Korrelation zwischen der abhängigen Variablen und den unabhängigen Variablen direkt sein muss und nicht von einer anderen unabhängigen Variablen herbeigeführt wird. Von externer Gültigkeit wird gesprochen, wenn sich ein Resultat generalisieren lässt.

Zunächst sollen verschiedene Gefahren für die interne Gültigkeit der Resultate betrachtet werden [Onw00, S. 15ff, S. 35ff, S. 46ff], gefolgt von Gefahren für die externe Gültigkeit [Onw00, S. 31ff, S.43ff, S. 50f]. Hierbei werden allerdings explizit keine Gefahren beschrieben, die auf einer Sortierung der Subjekte in einzelne Gruppen, einen Vortest oder die Beschreibung von Korrelationen eingehen, da keine der drei Methoden im Rahmen dieser Arbeit genutzt wurden. Außerdem werden Gefahren nicht betrachtet, die im Rahmen der Experimente nicht auftreten können. Dies gilt für die Gefahren *Prüfungsangst*, *Verhaltensvorurteil*, *Interaktion zwischen Zeit und Behandlung*, *Behandlungstrübung*, *Beschränkte Reichweite*, der *Crud-Faktor* und *Kausaler Fehler*. Gefahren, die im Rahmen dieser Arbeit auftreten können oder zumindest beachtet werden müssen, sollen an dieser Stelle diskutiert werden. Hierbei werden jeweils zunächst die auftretenden Gefahren erläutert, gefolgt von den beachteten Gefahren.

Interaktionsfreiheitsannahme: Es kann bei der Durchführung eines Experiments dazu kommen, dass die Forscher keine Interaktionen betrachten und dadurch die Komplexität des zu beschreibenden Zusammenhangs zu sehr vereinfachen. Dies wurde allerdings im Entwurf der Experimente berücksichtigt, da explizit immer mehrere Konfigurationen getestet werden, die unterschiedliche Interaktionen betrachten. Allerdings wurde *t-wise-sampling* mit $t = 2$ verwendet, also wurden höchstens paarweise Interaktionen mit betrachtet. An dieser Stelle kann also nach wie vor ein Fehler entstanden sein.

Bestätigungstendenz: Bestätigungstendenz beschreibt die Gefahr, dass eine Theorie immer noch als korrekt angesehen wird, auch wenn die ermittelten Daten eine Hypothese nicht unterstützen. Die Darstellung der im Rahmen dieser Arbeit ermittelten Resultate ist jedoch darauf ausgelegt, transparent mit den getroffenen Rückschlüssen umzugehen. Es wird explizit zwischen Hypothesen unterschieden, die angenommen werden konnten, und Hypothesen, die nicht abgelehnt werden konnten. Dies soll Transparenz bezüglich der Aussagekraft der Untersuchungen bieten. Allerdings wird in den Grafiken und in der Zusammenfassung auch von nicht angenommenen Hypothesen gesprochen. Hierbei kann ein Fehler auftreten. Durch die große Anzahl an Hypothesen, die sich allerdings nicht bestätigen lassen, wird dieser potenzielle Fehler an bei der Diskussion der Hypothesen in Kauf genommen und jeweils angemerkt. An dieser Stelle soll auch das Resultat aus der Untersuchung des *SPLConqueror*-Modells erwähnt werden. Hierbei handelt es sich explizit nur um

https://dx.doi.org/10.24355/dbbs.084-201801311013

eine unbestätigte Vermutung. Diese lässt sich im Rahmen dieser Arbeit auch nicht weiter untersuchen, da sie aus den vorliegenden Daten geschlussfolgert wurde.

Statistische Regression: Sollten die Subjekte einer Untersuchung anhand von speziellen Faktoren für ein Experiment ausgewählt worden sein, kann dies das Ergebnis verfälschen. Da alle zur Verfügung stehenden Spezifikationsfälle untersucht wurden, wurde keine Filterung vorgenommen, die nicht direkt aus der Hypothese hervorgeht. Diese Methoden wurden zwar vor dem Entwurf des Experiments ausgewählt, jedoch wurden sie gewählt, da an dieser Stelle allgemein eine Verifikation möglich war. Es wurde also nicht bewusst eine gewisse Methodenpopulation gewählt, es kann aber auch nicht nachgewiesen werden, dass bei der Auswahl der Methoden keine unbewusste Einschränkung vorgenommen wurde. Es besteht also nach wie vor das Risiko, dass die statistische Regression nicht eingehalten wurde und die Ergebnisse dadurch verfälscht wurden.

Typ-I- bis Typ-X-Fehler: Es wurden zehn verschiedene statistische Fehlerarten identifiziert [Onw00, S.39f]. Diese wurden allerdings im Rahmen dieser Arbeit nicht separat betrachtet. Es wurde jeweils nur der P-Wert angegeben, der direkt mit dem Typ-II-Fehler zusammenhängt. Der Typ-III-Fehler beschreibt eine fehlerhafte Annahme über die Richtung des Ergebnisses, was jeweils explizit definiert wurde. Der Typ-VIII-Fehler bezieht sich auf den Einsatz von beidseitigen Tests für einseitige Alternativhypothesen. Dies wurde beim Einsatz des McNemar-Tests teilweise gemacht, jedoch wurden die Ergebnisse manuell geprüft und die P-Werte in diesen Fällen so weit vom definierten Signifikanzniveau entfernt, dass dies nicht kritisch ist. Der Typ-I-Fehler beschreibt die Wahrscheinlichkeit, dass eine Nullhypothese fälschlicherweise abgelehnt wurde, was im Rahmen dieser Arbeit nicht geschehen ist. Die anderen Fehler spielen für die Experimente im Rahmen dieser Arbeit und die getroffenen Aussagen keine Rolle.

Problem der fehlerhaften Spezifizierung: Ein statistischer Test bildet ein Modell, welches eine Beobachtung beschreiben soll. Die Gefahr der fehlerhaften Spezifizierung besteht darin, dass bei einem Modell wichtige Variablen außer Acht gelassen werden. Da alle Variablen, die für die diese Untersuchung eine Rolle spielen, in KeY konfigurierbar sind, ist dieser Fehler unwahrscheinlich. An dieser Stelle sei allerdings noch einmal erwähnt, dass der Einsatz eines externen SMT-Solvers explizit nicht Teil dieser Arbeit ist. Dadurch wird allerdings ein wichtiger Parameter von KeY bei den Untersuchungen außer Acht gelassen. Da dies allerdings explizit geschieht, muss dies bei der Nutzung der Hypothesen beachtet werden, ändert aber nichts an ihrer Gültigkeit.

Geschichtliche Einflüsse: Die Gefahr des geschichtlichen Einflusses wirkt im ersten Moment entscheidender für soziologische Untersuchungen, kann jedoch auch auf Softwaresysteme angewendet werden. Sie beschreibt die Auswirkung, die der zeitliche Verlauf auf die Gültigkeit eines Ergebnisses hat. Da die KeY-Version allerdings nicht während der Ausführung der Experimente geändert wurde, können auch die Umgebungsvariablen während der Ausführung der Experimente nicht geändert worden sein.

https://dx.doi.org/10.24355/dbbs.084-201801311013

Die Reife: Die Reife ist erneut eine Gefahr, die sich vor allem auf die Arbeit mit Menschen bezieht, allerdings auf die Arbeit mit Programmen bezogen werden kann. Sie beschreibt die Auswirkung der Experimentdauer auf das Subjekt. In diesem Fall kann dies, auf Software übertragen, beispielsweise der Zustand des Speichers oder ähnlicher Systeme sein. Dies wurde jedoch explizit beim Entwurf des Experiments betrachtet. Um diese Gefahr zu umgehen, wird die virtuelle Maschine nach der Ausführung jedes Experiments beendet und neu gestartet.

Messtechnik: Durch den Einsatz einer falschen Messtechnik könnte es dazu kommen, dass die ermittelten Daten nicht konsistent genug sind. Auch diese Gefahr wurde im Vorfeld beachtet. Aus diesem Grund wird auf die verhältnismäßig konsistenten Beweisknoten zurückgegriffen und nicht beispielsweise die Beweisdauer als abhängige Variable für die Untersuchung des Beweisaufwands verwendet.

Abnutzung: Die Gefahr der Abnutzung beschreibt das Risiko, dass einzelne Subjekte im Verlauf des Experiments ausscheiden. Subjekte können in diesem Kontext zwar nicht selbst ausscheiden, jedoch wurden Datensätze bei den Betrachtungen des Beweisaufwands aussortiert. Hierbei wurden nur geschlossene Beweise betrachtet, dies wurde allerdings in den Hypothesen gefordert. Von daher wird diese Filterung nicht als Ausscheiden gewertet und die Abnutzung ist im Rahmen dieser Arbeit keine Gefahr.

Umsetzungstendenz: Umsetzungstendenz beschreibt das Problem, dass Behandlungen von einzelnen Ausführenden unterschiedlich eingesetzt werden können. Da die Ausführung der Experimente dieser Arbeit von identischen KeY-Instanzen durchgeführt wird, kann hierbei keine Verfälschung der Resultate auftreten.

Proben-Erweiterungs-Tendenz: Sollte die Menge der Subjekte im Laufe der Studie erweitert werden, kann dies das Resultat verfälschen. Die hinzugekommenen Subjekte wären nicht über die gesamte Dauer des Experiments in Behandlung gewesen und können sich dadurch anders verhalten. Da die Experimente mit KeY unabhängig voneinander ausgeführt werden, kann das spätere Hinzufügen von Subjekten das Experiment nicht verfälschen. Außerdem werden keine Subjekte hinzugefügt.

Auftragstendenz: Die Auftragstendenz beschreibt einen Lern- oder Ermüdungseffekt, der durch das wiederholte Ausführen eines Experiments mit einem Subjekt auftreten kann. Um das Problem zu umgehen, wird die virtuelle Maschine nach jeder Ausführung beendet.

Beobachtungstendenz: Wenn die Menge der für das Experiment verwendbaren Datensätze zu gering ist, kann dies einen Fehler verursachen. Dies soll allerdings grade durch die Signifikanztests beachtet werden und ist von daher kein Risiko für die Ergebnisse dieser Arbeit.

Zuordnungsbefangenheit: Sollte ein Forscher Datensätze oder Subjekte einander zuordnen, kann es passieren, dass einzelne Subjekte ausgeschlossen werden, wenn kein Partner gefunden wird. Da die Subjekte für diese Arbeit allerdings generiert werden und dadurch immer ein Partner entsteht, ist dies kein Problem.

https://dx.doi.org/10.24355/dbbs.084-201801311013

Fehler der Behandlungswiederholung: Wenn ein Experiment mit einer Gruppe von Subjekten durchgeführt wird, man aber nur einzelne Resultate in die Betrachtung aufnimmt, kann dies das Resultat verfälschen. Es werden allerdings, wenn nicht durch die Hypothese eingeschränkt, stets alle gemessenen Datensätze verwendet. Diese Gefahr ist folglich nicht gegeben.

Mehrfachbehandlungsstörung: Sollten mehrere Behandlungen an demselben Subjekt getestet werden, können sie sich gegenseitig beeinflussen. Da die virtuelle Maschine allerdings nach jeder Ausführung beendet wird, wird der Spezifikationsfall, unser Subjekt, nicht verändert. Auch diese Gefahr kann dadurch außer Acht gelassen werden.

Rückwirkende Gestaltung: Sollte ein Resultat sich dadurch verfälschen lassen, dass das Subjekt sich darüber bewusst ist, dass es untersucht wird, spricht man von rückwirkender Gestaltung. Auf die Experimente im Rahmen dieser Arbeit übertragen würde dies bedeuten, dass KeY, wenn es durch die selbst implementierte Schnittstelle angesteuert wird, sich anders verhält als sonst. Da die Oberfläche von KeY aber keine anderen Methoden verwendet als die Schnittstelle, ist diese Gefahr im Rahmen dieser Experimente nicht gegeben.

Interaktion zwischen geschichtlichen Einflüssen und der Behandlung: Diese Gefahr bezieht sich auf die Einflüsse von geschichtlichen Entwicklungen auf die Resultate des Experiments. Sollten neu erstellte Datensätze mit bereits bestehenden Datensätzen verglichen werden, können externe Entwicklungen ebenfalls Auswirkungen auf die Resultate haben. Die Experimente wurden allerdings alle mit derselben Version von KeY und derselben virtuellen Maschine erstellt. Deshalb kann angenommen werden, dass keine externen Entwicklungen die interne Gültigkeit beeinflusst haben können.

Multikollinearität: Multikollinearität beschreibt eine hohe Korrelation einzelner Variablen mit einander, wodurch eine Beobachtung mehrere Erklärungen haben kann. Da die Korrelation einzelner Optionen in KeY nicht betrachtet wurde, kann es an dieser Stelle im Rahmen der Experimente zu einem Fehler gekommen sein. Dies spielt allerdings keine Rolle, da keine Ursachenforschung betrieben werden sollte, sondern die Hypothesen Anwendern Hilfestellungen bieten sollen. Das bedeutet, dass die Richtigkeit einer Erklärung keine Rolle spielt, so lange die Beobachtung wiederholbar ist.

Effektgröße: Die Effektgröße bezieht sich auf die Interpretation eines P-Wertes und dem Unterschied zwischen statistischer Signifikanz und Signifikanz im umgangssprachlichen Sinne. Da im Rahmen dieser Arbeit nur von statistischer Signifikanz gesprochen wird, ist an dieser Stelle nicht von einem Fehler auszugehen.

Verzerrte Grafiken: Grafiken wurden im Rahmen dieser Arbeit nicht zur Untersuchung von Hypothesen genutzt, sondern nur im Rahmen der Betrachtung der Modelle aus *SPLConqueror*. An dieser Stelle wurde - auch aus anderen Gründen - explizit nicht von einer Hypothese oder Bestätigung gesprochen. Aus diesem Grund kann hierbei nicht von einem Fehler gesprochen werden.

https://dx.doi.org/10.24355/dbbs.084-201801311013

Nun sollen die externen Gefahren äquivalent zu den bereits betrachteten internen Gefahren betrachtet werden. Einige der Gefahren wiederholen sich allerdings von den internen Gefahren und werden deshalb nicht erneut genannt.

Gültigkeit der Subjektmenge: Die Gültigkeit der Subjektmenge beschreibt die Generalisierbarkeit des Resultats. Das Problem ist, dass im Rahmen dieser Arbeit nur eine geringe Anzahl verschiedener Methoden betrachtet wird, auch wenn jeweils mehrere Spezifikationsfälle betrachtet werden. Die Anzahl der Methoden lässt sich auch nicht ohne weiteres vergrößern, da es relativ aufwändig ist, weitere Methoden der Java Platform API verifizierbar zu machen. Das Risiko, dass das Ergebnis durch die geringe Anzahl an Methoden nicht hinreichend generalisierbar ist, muss also in Betracht gezogen werden.

Ökologische Richtigkeit: Mit der Umwelt sind in diesem Fall die umgebenden Einflüsse auf ein Resultat gemeint. Da KeY in einer größtenteils definierten Umgebung ausgeführt wird, sind an dieser Stelle kaum Einflüsse zu erwarten. Allerdings kann es zu einem Absturz kommen, wenn nicht genügend Arbeitsspeicher zur Verfügung steht. Dies konnte bei der Ausführung der Experimente aber nicht beobachtet werden. Da die weiteren Konfigurationen als unabhängige Variablen für die Experimente, beziehungsweise als Teil des Subjekts, betrachtet werden, ist auch hierbei nicht von einer indirekten Einflussnahme auszugehen.

Zeitliche Gültigkeit: Zeitliche Gültigkeit beschreibt die Haltbarkeit eines Ergebnisses. Wenn die verwendete KeY-Version sich ändert, kann sich theoretisch die Auswirkung der Parameter auf den Beweiser ändern. Die Experimente wurden in diesem Fall mit KeY 2.7.1430 ausgeführt und die Resultate sind damit in erster Linie auch nur für diese Version gültig. Mit einer neuen KeY-Version sollten die Hypothesen dennoch erneut untersucht und nicht einfach übernommen werden.

Genauigkeit der Variablen: Die Gefahr durch ungenaue Variablen rührt daher, dass nicht beschriebene Variablen einen Einfluss auf die Resultate nehmen können. Da die Variablen bei der Ausführung von KeY abseits der Konfiguration allerdings überschaubar sind, geht bei diesen Experimenten keine große Gefahr von der Unterspezifikation der Variablen aus.

Von den verschiedenen internen Gefahren sind nur die Gefahr der statistischen Regression, der Bestätigungstendenz und der Interaktionsfreiheitsannahme aufgefallen. Das Risiko, dass die Gefahr der statistischen Regression eintritt, scheint zwar nicht groß zu sein, ist aber vorhanden. Um die Gefahr zu widerlegen, müssten die Experimente mit einer größeren Anzahl, zufällig ausgewählter Spezifikationsfälle wiederholt werden. Da diese Arbeit allerdings nur Methoden der Java Platform API betrachtet, stehen diese noch nicht zur Verfügung, sondern müssten erst erstellt werden. Die Bestätigungstendenz folgt aus der Art der Hypothesen und der Tatsache, dass nicht annehmbare Hypothesen, die aber auch nicht abgelehnt wurden, dennoch diskutiert werden. Deshalb kann diese Gefahr auch nicht direkt behandelt werden. Sie muss dem Leser allerdings bekannt sein, damit er sie beachten kann. Die Gefahr der Interaktionsfreiheitsannahme wurde beim Entwurf der Experimente zwar bereits mit bedacht, es wurde allerdings ein Kompromiss zwischen der Komplexität

https://dx.doi.org/10.24355/dbbs.084-201801311013

der betrachteten Interaktionen und der Anzahl der nötigen Experimente getroffen. Um diese Gefahr explizit zu adressieren, müssten die für die Experimente genutzten Konfigurationen mit einem höheren t-Wert beim *t-wise-sampling* generiert werden.

Auch bei den externen Gefahren bemerkt man, dass die größte Gefahr für einen Fehler aus der niedrigen Anzahl der getesteten Methoden hervorgeht. Die größte Qualitätssteigerung der Resultate kann man demnach erreichen, in dem mehr verschiedene Methoden spezifiziert werden. Mit Hilfe dieser kann die Menge der Testsubjekte erweitert werden, wodurch die Generalisierbarkeit der Ergebnisse steigen würde. Das Risiko des geschichtlichen Einflusses ist ebenfalls gegeben. Solange keine neue KeY-Version existiert, sind die Ergebnisse dennoch gültig.

5.6 Zusammenfassung

In diesem Abschnitt wurden die in Kapitel 4 erstellten Hypothesen ausgewertet. Mit Hilfe dieser Resultate mussten zwei Hypothesen abgelehnt werden: Hypothese 21 und Hypothese 22 haben sich bei der Untersuchung als fehlerhaft formuliert herausgestellt und wurden aus diesem Grund abgelehnt. In diesen Hypothesen wurde eine Aussage zu Beweisen ohne Queries getroffen, jedoch wurde festgestellt, dass es keine Beweise von Java-Quelltext ohne Queries gibt.

Die anderen Hypothesen konnten entweder angenommen oder zumindest nicht abgelehnt werden. Aus diesen Hypothesen heraus lassen sich für Anwender, die KeY vollautomatisch verwenden wollen, direkt Empfehlungen ableiten. Wenn die Zeit oder die Speicherauslastung unkritisch sind, aber es unklar ist, ob ein Beweis automatisch geschlossen werden kann, müssen nicht alle Parameter beachtet werden. In diesem Fall ist es möglich einige Optionen für jede Verifikation festzulegen. Beispielsweise können *Quantifier Treatment* und *Proof Splitting* in einem solchen Szenario den Hypothesen nach mit der Option *Free* belegt werden. Auch ergibt es unter diesem Blickpunkt keinen Sinn, *Basic* als *Arithmetic Treatment* zu wählen oder *Stop At* zu ändern. So kann die Anzahl der zu betrachtenden Parameter im ersten Schritt gesenkt werden, was die Einarbeitung in KeY vereinfachen kann. In Abbildung 5.3 wurde die Abhängikeit dieser Optionen zum Beweisaufwand und der Beweisbarkeit zusammengefasst.

Bei einigen Parametern lässt sich aus den jeweiligen Hypothesen ableiten, dass der Beweisaufwand zwischen den jeweiligen Optionen identisch ist, die eine Option jedoch eine bessere Beweisbarkeit mit sich bringt. Als Anwender stellt sich die Frage, weshalb die Oberfläche die Auswahlmöglichkeit überhaupt anbietet. Das hat zur Folge, dass mehr Optionen vom Anwender betrachtet werden müssen, obwohl sie für ihn eigentlich keinen Mehrwert bieten. Es wäre also empfehlenswert, eine alternative Ansicht anzubieten, bei der einige Optionen ausgeblendet werden. Dies ließe sich beispielsweise als „Einfacher Modus“ und „Expertenmodus“ umsetzen. Im einfachen Modus stünden weniger Optionen zur Auswahl, sodass KeY im einfachen Modus für den Anwender den Blick auf das für ihn Wesentliche lenkt. Dies umfasst die Parameter *BigInt*, *Strings*, *IntegerSimplificationRules*, *Sequences MoreSeqRules*, *One Step Simplification* und *Stop At*. Bei *Loop Treatment* lässt sich dies ebenfalls aus der Hypothese ableiten, allerdings muss auch beachtet werden, dass ein Anwender

https://dx.doi.org/10.24355/dbbs.084-201801311013

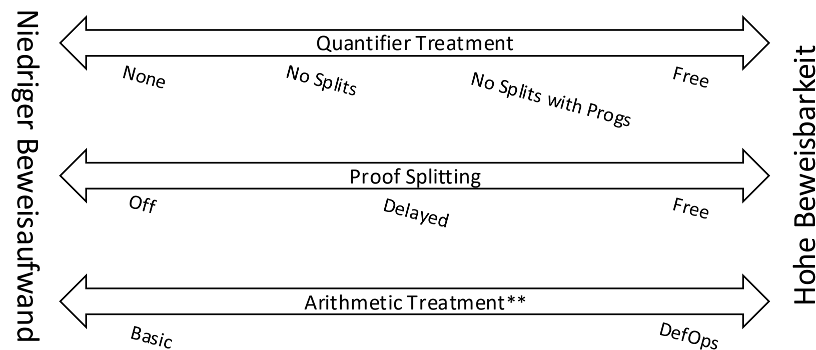

Abbildung 5.3: Abhängigkeiten ausgewählter Optionen zu Beweisbarkeit und Beweisaufwand

eventuell *Expand* nutzen möchte, welches in dieser Arbeit nicht mit betrachtet wurde (siehe Beschreibung 4.5).

In diesem Modus könnte KeY für Anwender auch Tipps zur Verfügung stellen. Wenn sich ein Beweis nicht schließen lässt, einzelne Optionen sich jedoch durch Optionen ersetzen lassen, die eine höhere Beweisbarkeit haben. Dies trifft beispielsweise auf *Proof Splitting Off* oder *IntegerSimplificationRules Minimal*. In diesen Fällen könnte *Proof Splitting Free*, beziehungsweise *Integer SimplificationRules Full*, vorgeschlagen werden.

Zusätzlich zu diesen Informationen ist im Rahmen dieser Arbeit eine Software entstanden, mit der die Ausführung einzelner Spezifikationsfälle auf verschiedene Systeme verteilt werden kann. Diese kann beispielsweise auch genutzt werden, um in einem Rechencluster eine komplexere Software zu verifizieren. Die Voraussetzung hierfür wäre allerdings, dass die Beweise vollautomatisch ausgeführt werden können. Für umfangreiche Systeme, die regelmäßig verifiziert werden sollen, ist dies für viele Spezifikationsfälle allerdings ohnehin notwendig, da der Verifikationsaufwand bei manuellen Verifikationen zu groß.

https://dx.doi.org/10.24355/dbbs.084-201801311013

https://dx.doi.org/10.24355/dbbs.084-201801311013

6. Verwandte Arbeiten

Dieses Kapitel soll verwandte Arbeiten vorstellen. Im Rahmen dieser Arbeit wurde Java-Quelltext mit Hilfe von KeY verifiziert, um anschließend detailliert auf die Parameter und Optionen von KeY einzugehen. Dadurch gibt es vor allem in zwei Bereichen ähnliche Arbeiten: Werkzeuge zur deduktiven Verifikation und die Spezifikation und Verifikation von Java-Quelltext.

Werkzeuge zur deduktiven Verifikation

Eine der wichtigsten Quellen für diese Arbeit ist das KeY-Buch [ABB+16]. Dieses bietet einen umfangreichen Einblick in die Ideen und Technologien, die in KeY eingesetzt werden. Allerdings gibt das KeY-Buch auch eine Einführung in die Arbeit mit KeY als Anwender und bietet auch einige Beispiele, wie Verträge aufzubauen sind. Darüber hinaus stellt es allerdings nicht nur KeY, sondern auch andere Tools aus dem KeY-Projekt dar. Diese Arbeit ergänzt sich hierbei mit dem KeY-Buch, das zwar auch Beispiele für Spezifikationen und die Arbeit mit KeY gibt, aber den Fokus auf die Technik und den Stand der Forschung legt. Im Rahmen dieser Arbeit liegt der Fokus aber mehr auf der Anwendersicht und dient sich dadurch eher für den Einstieg in die Arbeit mit KeY. Eine kurze Zusammenfassung des KeY-Projekts existiert ebenfalls [ABB+14].

KeY ist allerdings nicht das einzige System, welches deduktive Verifikation zur Verfügung stellt. Eine Alternative ist *Boogie*. *Boogie* ermöglicht die Verifikation von C#-Quelltext [BCD+05] und ist Teil des Spec#-Projekts [BLS05]. Im Rahmen des Hypervisor Verification Projekts wurde *VCC* entwickelt [CDH+09]. Hierbei handelt es sich um mehrere Werkzeuge, von denen eines statische Verifikation ermöglicht. Diese sollen an dieser Stelle allerdings für eine Übersicht genannt werden. Durch den Fokus auf die Systeme werden allerdings jeweils die eingesetzten Techniken und einzelne Features beschrieben und keine ausführliche Anleitung für die Spezifikationen gegeben.

Spezifikation und Verifikation von Java-Quelltext

Das OpenJML-Projekt bietet bereits umfangreiche Spezifikationen für die Java Platform API, diese lassen sich allerdings leider nicht von KeY einlesen [Cok11][Cok14].

https://dx.doi.org/10.24355/dbbs.084-201801311013

OpenJML stellt außerdem eine Reihe verschiedener Werkzeuge zur Verfügung. Eines davon führt auch eine statische Verifikation mit Hilfe von *Boogie* zur Verfügung. In diesen Arbeiten geht es allerdings um den Funktionsumfang von OpenJML und nicht direkt um Erfahrungen aus der Arbeit mit ihm. Die existierenden Verträge werden auch nicht erläutert, sondern als gegeben angesehen.

Andere Projekte spezifizieren einzelne Softwarebereiche, wie zum Beispiel den Sortieralgorithmus von `Collections` des OpenJDK [dGRdB+15] oder Garbage-Collection-Algorithmen [SSZ+15]. Gouw et al. gehen hierbei ausführlich auf die Spezifikation und das von Ihnen bemerkte Problem in dem Sortieralgorithmus ein und erwähnen KeY und dessen Einsatz eher am Rande. Die darauf aufbauende Arbeit von 2017 geht weitaus ausführlicher auf die Arbeit mit KeY ein und stellt in diesem Kontext die Arbeit mit *Merge Point Statements* vor (siehe Beschreibung 4.8) [dGdBB+17]. Auch wird im Rahmen dieser Arbeit die Taclet-Option *intRules* ausführlich diskutiert. Sun et al. gehen in ihrer Arbeit vor allem auf ihre Verträge ein, jedoch nicht auf die Arbeit mit dem Verifikationswerkzeug. Keine dieser Arbeiten geht allerdings auf die Arbeit mit dem jeweiligen Werkzeug direkt in der Form ein, wie es in dieser Arbeit geschieht. Teilweise stellen sie zwar die verwendeten Verträge vor, sprechen aber nur in Ausnahmen von den verwendeten Parametern und den Gründen für die jeweiligen Entscheidungen. Auch wird zwar das jeweilige Vorgehen vorgestellt, jedoch wird nur in Ausnahmen über die Erfahrungen mit dem Verifikationswerkzeug gesprochen.

Beckert et al. haben die nachträgliche Validierung von bestehenden Systemen betrachtet [BBG16]. Thematisch sind sie also sehr dicht an dieser Arbeit, der Fokus ist allerdings anders. Die Grundannahme bei ihnen ist, dass Quelltext verändert werden darf und sie beschreiben in diesem Zusammenhang das Vorgehen. Sie behandeln dabei allerdings nicht die Arbeit mit KeY an sich, auch wenn sie ebenfalls Vorschläge für Änderungen an den Verifikationswerkzeugen ableiten. Diese beziehen sich jedoch auf beobachtete Probleme bei der allgemeinen Arbeit mit bestehenden Systemen.

KeY unterstützt explizit auch JavaCard. Hierfür steht auch eine verifizierte API-Referenzimplementerung zur Verfügung [Mos07]. Ihr Ziel war es, dass die Beweise geschlossen werden müssen, hierbei wurde kein Fokus auf Automatisierung gelegt. Es wurde allerdings festgestellt, dass eine automatische Verifikation im Großteil der Fälle möglich war. Insgesamt beschrieben sie durch den anderen Fokus andere Probleme, vor allem in der Anwendung von Regeln und den daraus resultierenden, benötigten Anpassungen. Sie gehen hierbei allerdings natürlich nur auf JavaCard ein und diskutieren keine Probleme die bei der Arbeit mit Java-Quelltext auftreten.

Baumann et al. haben mit Hilfe von VCC einen Mikrokernel verifiziert und ihre Erfahrungen in diesem Projekt diskutiert [BBBB12]. Ihr Fazit war, dass die betrachteten Werkzeuge mittlerweile effizient auch für größere Projekte eingesetzt werden können, was sich nicht mit den Resultaten dieser Arbeit deckt. Der größte Aufwand, den sie identifiziert haben, lag bei ihnen in der Spezifikation, während er in dieser Arbeit eher auf der Vorbereitung des Quelltextes lag.

https://dx.doi.org/10.24355/dbbs.084-201801311013

7. Zusammenfassung

Diese Arbeit sollte ermitteln, wie leicht Softwareentwickler Java-Quelltext mit KeY verifizieren können. Hierfür wurden im Rahmen dieser Arbeit mit Hilfe der bestehenden, informellen Spezifikation 21 Methoden der Java Platform API des OpenJDK mit Hilfe von JML formal spezifiziert (siehe Kapitel 3). Im Rahmen dieser Arbeit konnte dadurch eine Abweichung des Verhaltens der `ArrayList` von der Spezifikation der `Collection` festgestellt werden (siehe Abschnitt 3.3.5). Selbst in den Spezifikationen der `Collection` befindet sich bereits ein auf dem selben Problem beruhender Widerspruch. Dieser Widerspruch besteht jeweils auch noch in der aktuellen Version 9.0.1 von *Java SE*.

Dieses Fehlverhalten der `ArrayList` geht auf das Verhalten von Arrays zurück. Dies ist jedoch nicht direkt von KeY bemerkt worden, sondern im Rahmen der Verifikation aufgefallen. Java wirft bei der Erstellung eines Arrays einen `OutOfMemoryError`, wenn die Größe einen gewissen Wert überschreitet (siehe Abschnitt 2.1.3). Dieses Verhalten wird jedoch nicht in KeY abgebildet (siehe Abschnitt 3.5). Java definiert in der Sprachspezifikation außerdem keine maximale Arraygröße. Die Vermutung liegt nahe, dass dies eine Freiheit ist, die den jeweiligen virtuellen Maschinen überlassen wird. Da eine Anpassung in Java unwahrscheinlich und langwierig wäre, wurde vorgeschlagen, dass KeY angepasst werden kann. Hierbei könnte der Anwender im Rahmen einer Taclet-Option eine maximale Array-Größe festlegen.

Im Rahmen der formalen Spezifikation konnte das Vorgehen exemplarisch dargestellt werden (siehe Abschnitt 3.1). Anhand dieser gewählten Beispiele wurden ein paar Missverständnisse bemerkt (siehe Abschnitt 3.4). Die Missverständnisse konnten jeweils auf ein Minimalbeispiel zurückgeführt und so verallgemeinert werden. Anhand dieser Missverständnisse können Anwender, die auf ein ähnliches Problem stoßen, dieses leichter lösen oder es als Grundlage für eigene Spezifikationen nutzen.

Die Verifikation an sich wurde allerdings vor allem durch zwei Faktoren erschwert (siehe Abschnitt 3.1). Eines der Hauptprobleme in dieser Arbeit war *JavaRedux*, einer Sammlung verschiedener Methodensignaturen der Java Platform API. Es hat sich in seiner aktuellen Form als problematisch herausgestellt, da viele Konstan-

https://dx.doi.org/10.24355/dbbs.084-201801311013

tenfelder von Java Platform API-Klassen fehlen und mit Hilfe der Benutzeroberfläche auch nicht ersetzt werden kann. Ein manuelles Festlegen des zu verwendenden Bootklassenpfads, auch abseits von *JavaRedux*, würde die Arbeit bereits stark vereinfachen. Das langfristige, wünschenswerte Ziel wäre es, für eine Vielzahl an Java-Versionen ein vollständiges, auch spezifiziertes *JavaRedux* zur Verfügung zu haben. Der zweite Faktor war das Einlesen des Quelltextes an sich. Einige Java-Sprachkonstrukte, wie ein Teil der Annotationen, wurden nicht unterstützt und mussten deshalb manuell entfernt werden. Um dieses Problem zu lösen, wäre eine Aktualisierung des verwendeten Java-Parsers nötig. Die neuen Java-Funktionalitäten müssten dabei nicht einmal unbedingt von KeY unterstützt werden, da sie für die Verifikation von Methoden nicht unbedingt benötigt werden. Ein Beispiel hierfür wäre die Annotation `@SurpressWarnings`.[1] Es würde die Arbeit mit KeY allerdings stark vereinfachen, wenn der Quelltext ohne aufwändige Modifikationen eingelesen werden kann.

Während der ursprünglichen Erstellung der in Kapitel 3 vorgestellten Kontrakte war eines der größten Probleme nach dem Einlesen des Quelltextes die Wahl der richtigen Parameter. Teilweise schwankt der Beweisaufwand zwischen den Konfigurationen sehr stark, weshalb für wiederkehrende Verifikationen nach einer Anpassung eine Konfiguration mit möglichst geringem Aufwand genutzt werden sollte, diese kann bei anderen Spezifikationsfällen die Verifikation allerdings behindern. Sich die richtige Konfiguration zu merken oder sie erneut zu finden war demotivierend, da unklar war, ob es an einer geänderten Spezifikation oder der Konfiguration lag. Um dieses Problem zu lösen, wurden 38 Hypothesen auf Basis der Beschreibungen und verfügbaren Quellen aufgestellt und untersucht (siehe Abschnitt 4.1 und Abschnitt 4.2). Hierbei mussten nur zwei Hypothesen abgelehnt werden, was für die Güte der jeweiligen Beschreibungen der Parameter spricht. Allerdings waren die Beschreibungen gerade für Taclet-Optionen sehr verstreut und teilweise nicht ausführlich (siehe beispielsweise Beschreibung 4.32). Zu allen Parametern konnte jedoch eine Beschreibung aus Anwendersicht zusammengestellt werden und mit Hilfe der Hypothesen kann durch diese Arbeit eine Hilfestellung für die Anwender zur Verfügung gestellt werden (siehe Abschnitt 5.6).[2] So ist beispielsweise aufgefallen, dass einige der Taclet-Optionen für einen vollautomatischen Beweis nicht ausgewählt werden sollten. Aus diesem Grund wird vorgeschlagen, eine Art „einfache“ Benutzeroberfläche zur Verfügung zu stellen, bei der diese Optionen nicht mehr angeboten werden. Dadurch wird der Blick des Anwenders mehr auf die für ihn relevanten Optionen und Parameter gelenkt und die ersten Schritte in KeY fallen dadurch leichter. Auch könnte die Benutzeroberfläche bei einem Beweis, der nicht geschlossen werden konnte, dem Anwender bei bestimmten Konfigurationen einen Vorschlag machen, wie er eventuell doch geschlossen werden kann.

Zum Abschluss dieser Arbeit wurden die in Kapitel 3 vorgestellten Kontrakte noch einmal mit der Oberfläche von KeY verifiziert, um sicher zu stellen, dass sich im Laufe der Bearbeitung keine Fehler eingeschlichen haben. Ein persöhnlicher Erfolg

[1] Diese wurden mit Java 5.0 im Jahr 2004 hinzugefügt (siehe Abschnitt 2.1.1)

[2] Diese Resultate basieren teilweise auf Hypothesen, die durch Ihren Aufbau niemals angenommen werden können. Sie wurden im Rahmen dieser Arbeit untersucht und konnten nicht widerlegt werden, weshalb sie weiterhin bestehen bleiben dürfen.

https://dx.doi.org/10.24355/dbbs.084-201801311013

war es, dass die abschließende Validierung mit Hilfe der in Kapitel 4 aufgestellten Hypothesen und Beschreibungen wesentlich reibungsloser von statten ging. Da nur noch einmal die Beweisbarkeit der Spezifikationsfälle überprüft wurde, konnte dank der im Rahmen dieser Arbeit aufgestellten Hypothesen sehr viele Optionen von vorn herein außer acht gelassen werden, was den Fokus auf die Auswahl der übrigen Parameter gelenkt hat.

Zusammenfassend lassen sich einige positive Ergebnisse dieser Arbeit hervorheben. So wurde ein Problem in der informellen Spezifikation der *Collection* gefunden und ein Lösungsvorschlag entwickelt. Außerdem konnte festgestellt werden, dass die Verifikation von Methoden der Java Platform API realistisch ist, allerdings wurden hierbei auch mehrere Punkte identifiziert, die die Arbeit mit KeY unnötig erschweren. Zu diesen Punkten ist es jeweils gelungen, Lösungsvorschläge zu entwickeln. Abschließend wurde eine benutzerfreundliche Gestaltung in den Fokus genommen. Hier wurden mit Hilfe der Analyse der Parameter einige Vorschläge erarbeitet. Die Oberfläche von KeY kann dadurch noch nutzerfreundlicher und übersichtlicher gestaltet werden. Durch die Umsetzung könnte vielen Nutzern die Arbeit mit KeY deutlich erleichtert werden.

https://dx.doi.org/10.24355/dbbs.084-201801311013

https://dx.doi.org/10.24355/dbbs.084-201801311013

8. Ausblick

Im Rahmen dieser Arbeit wurden grundlegend drei Themengebiete behandelt. Zunächst wurde eine Betrachtung der Spezifikation von Java-Methoden behandelt. Anschließend wurde eine qualitative Analyse der Parameter und Optionen von KeY durchgeführt, um diese daraufhin in einer quantitativen Untersuchung näher zu betrachten. Zu dem Thema der Spezifikationen und der quantitativen Untersuchung bieten sich noch interessante weitere Ansatzpunkte für zukünftige Arbeiten. Auch können die vorgeschlagenen Anpassungen in KeY noch einmal ausführlicher betrachtet und ausgearbeitet werden.

Erstellung weiterer Spezifikationen

Auf Grund des nötigen Aufwands, um die Spezifikation einer Methode der Java Platform API verifizieren zu können, liegt der Umfang derzeit bei 21 Methoden. Da diese Spezifikationen auch als Grundlage für die Erweiterung von *JavaRedux* dienen können, sollte die Arbeit an dieser Stelle fortgesetzt werden. Die so entstandenen Spezifikationen können nicht nur einen weiteren Einblick in wünschenswerte Anpassungen von KeY geben, sondern auch anderen Softwareentwicklungsprojekten als Grundlage für die Verifikationen des eigenen Quelltextes dienen. Auch sind Teilgebiete der Platform API aufgefallen, die sich nicht ohne Weiteres spezifizieren lassen, aber an vielen anderen Stellen genutzt werden. Dies trifft beispielsweise auf die *Reflections*-API und das Klassensystem in Java zu. Für weitere Verifikationen im Bereich der Platform API wäre eine vorherige Betrachtung solcher Elemente wichtig.

Bereinigung der Gefahren zur statistischen Analyse

Auch wurde die Subjektmenge als eine der Gefahren für die Gültigkeit der Resultate der quantitativen Untersuchung identifiziert. Durch das Hinzufügen weiterer Methoden aus verschiedenen Gebieten der Platform API könnte die Subjektmenge der Experimente erweitert werden und dadurch das Vertrauen in die Hypothesen weiter erhöht werden oder es könnten fehlerhafte Hypothesen auffallen. Insgesamt haben viele der Hypothesen einen Aufbau, der eine Annahme im Rahmen eines Signifikanztests nicht möglich macht. Es wäre auch interessant, die Experimente mit

https://dx.doi.org/10.24355/dbbs.084-201801311013

anderen Datenbasen zu wiederholen, die durch den Fokus dieser Arbeit auf alltäglichen Quelltext, und damit auch auf die Java Platform API, nicht verwendet wurden. Dadurch könnte die Subjektmenge erweitert werden, ohne den Aufwand weiterer, neuer Spezifikationen vorauszusetzen. Hierbei muss allerdings darauf geachtet werden, dass es sich bei den Methoden und Verträgen um Quellcode handeln kann, der auch in der Industrie vorkommt. Der Einsatz existierender Beispiele bringt die Gefahr mit sich, dass es sich hierbei um Sonderfälle handeln kann, die in der Form selten oder im schlimmsten Fall garnicht vorkommen.

Eine weitere Einschränkung, die bei den Experimenten durch den zeitlichen Rahmen dieser Arbeit vorgenommen werden musste, war die Einschränkung der getesteten Konfigurationen. Aus diesem Grund konnte auch die Gefahr der Interaktionsfreiheitsannahme nicht ausgeschlossen werden. Um dieses Risiko zu minimieren wäre es interessant, eine wesentlich größere Anzahl verschiedener Konfigurationen zu testen, und zu untersuchen, ob Hypothesen abgelehnt werden müssen oder weitere Hypothesen angenommen werden können. Alle Kombinationen zu testen wäre zwar unrealistisch, allerdings wäre es gut, die Experimente mit höheren Interaktionen zu wiederholen und beispielsweise ein *4*- oder *6-wise-sampling* zu verwenden.

Überprüfung neuer Annahmen

Durch die Erstellung der *SPLConqueror*-Modelle sind außerdem neue Vermutungen bezüglich des Beweisaufwands der *Arithmetic Treatment Basic* und *Class Axiom Rule Delayed* Optionen entstanden (siehe Abschnitt 5.4). Diese müssten im Rahmen einer separaten Arbeit untersucht werden, da sie mit den Daten in dieser Arbeit gebildet wurden. Ansonsten würde diese Annahme bestätigt werden, ohne dass sie unabhängig untersucht wurde.

Vorhersagen über die Schließbarkeit / den Aufwand eines Beweises

Eine weitere Frage, die im Rahmen dieser Arbeit aufgekommen ist, ob es wohl möglich ist, ein Modell aufzustellen, dass die Beweisbarkeit oder den Beweisaufwand beschreibt. Hierbei würden nicht nur die Parameter als Variablen genutzt werden, sondern auch Eigenschaften des Spezifikationsfall und der Methode: etwa ob eine Schleife vorkommt, oder eine Query im Spezifikationsfall. Hierzu müssten allerdings im Vorfeld noch passende Experimente entwickelt werden, um die Datensätze für ein solches Modell zu bilden. Eine Alternative dazu wäre eine Heuristik oder ein Expertensystem, dass zu einer gegebenen Methode eine möglichst optimale Konfiguration vorschlägt.

Fokussierte Arbeiten für einzelne Parameter

Es gibt für einzelne Parameter bereits Arbeiten, die sich auf diese fokussieren - ein Beispiel hierfür wäre *Merge Point Statements* [SHB16]. Es wäre interessant andere, wichtige Parameter im Rahmen einer solchen Arbeit zu betrachten. Als Beispiel soll hierbei eine Untersuchung zu *Method Treatment* genannt werden, bei der betrachtet wird, ab welcher Methodengröße sich der Einsatz von *Contracting* wirklich lohnt - auch in Hinsicht auf den dadurch steigenden Verifikationsaufwand. Dies ist natürlich auch für andere Parameter möglich - auf Grund der Auswirkung der Paramater

https://dx.doi.org/10.24355/dbbs.084-201801311013

bieten sich hierbei vor allem die Optionen zur Strategie der Beweissuche an, da sie laut den Beschreibungen oftmals komplexere Auswirkungen auf den Beweis an sich haben und seltener fachliche Entscheidungen bezüglich der zu zeigenden Aussage betreffen.

Anpassungen an KeY

Im Rahmen dieser Arbeit wurden mehrere, mögliche Verbesserungen an KeY vorgeschlagen. Dies umfasst die Überprüfung der maximalen Array-Größe und die dafür nötige Anpassung in KeY, die Aktualisierung des verwendeten Parsers auf eine neue Version, die Ersetzung von *JavaRedux* über die Benutzeroberfläche, den „einfachen“ Modus auf der Benutzeroberfläche und das Vorschlagen von anderen Kofigurationen. Wenn die Benutzeroberfläche Konfigurationen vorschlägt, wäre es eventuell auch möglich, dass KeY dieses testen von Konfigurationen automatisiert. All diese Verbesserungsvorschläge sollten noch einmal ausgearbeitet werden, um sie eventuell in KeY umzusetzen.

Wiederholung in anderen Systemen

Diese Arbeit bezieht sich auf das Vorgehen mit Java-Quelltext und KeY. Interessant wäre eine Vergleichsarbeit mit einem anderen System - beispielsweise *Boogie* für C# aus dem System *Spec#* [BLS05][BCD+05]. Ziel einer solchen Arbeit wäre es, die Arbeit mit *Boogie* an sich vorzustellen, um anschließend die Prozesse der Systeme mit einander zu vergleichen.

https://dx.doi.org/10.24355/dbbs.084-201801311013

https://dx.doi.org/10.24355/dbbs.084-201801311013

A. Anhang

A.1 OpenJDK GPLv2

Die folgende Lizenz wurde kopiert von [Oraa]:

The GNU General Public License (GPL)

Version 2, June 1991

Copyright (C) 1989, 1991 Free Software Foundation, Inc. 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software–to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

https://dx.doi.org/10.24355/dbbs.084-201801311013

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The „Program“, below, refers to any such program or work, and a „work based on the Program“ means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term „modification“.) Each licensee is addressed as „you“.

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications

https://dx.doi.org/10.24355/dbbs.084-201801311013

or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

https://dx.doi.org/10.24355/dbbs.084-201801311013

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the so-

https://dx.doi.org/10.24355/dbbs.084-201801311013

le purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and „any later version“, you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM „AS IS“ WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PAR-

https://dx.doi.org/10.24355/dbbs.084-201801311013

TY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the „copyright“ line and a pointer to where the full notice is found.

One line to give the program's name and a brief idea of what it does.

Copyright (C) <year> <name of author>

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

Gnomovision version 69, Copyright (C) year name of author Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type 'show w'. This is free software, and you are welcome to redistribute it under certain conditions; type 'show c' for details.

The hypothetical commands 'show w' and 'show c' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than 'show w' and 'show c'; they could even be mouse-clicks or menu items–whatever suits your program.

https://dx.doi.org/10.24355/dbbs.084-201801311013

You should also get your employer (if you work as a programmer) or your school, if any, to sign a „copyright disclaimer“ for the program, if necessary. Here is a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright interest in the program 'Gnomovision' (which makes passes at compilers) written by James Hacker.

signature of Ty Coon, 1 April 1989

Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

„CLASSPATH“ EXCEPTION TO THE GPL

Certain source files distributed by Oracle America and/or its affiliates are subject to the following clarification and special exception to the GPL, but only where Oracle has expressly included in the particular source file's header the words „Oracle designates this particular file as subject to the ‚Classpath' exception as provided by Oracle in the LICENSE file that accompanied this code.“

Linking this library statically or dynamically with other modules is making a combined work based on this library. Thus, the terms and conditions of the GNU General Public License cover the whole combination.

As a special exception, the copyright holders of this library give you permission to link this library with independent modules to produce an executable, regardless of the license terms of these independent modules, and to copy and distribute the resulting executable under terms of your choice, provided that you also meet, for each linked independent module, the terms and conditions of the license of that module. An independent module is a module which is not derived from or based on this library. If you modify this library, you may extend this exception to your version of the library, but you are not obligated to do so. If you do not wish to do so, delete this exception statement from your version.

https://dx.doi.org/10.24355/dbbs.084-201801311013

A.2 Weitere erstellte Verträge

A.2.1 Spezifikation von Collection.contains(Object)

Diese Methode ruft die Methode Collection.indexOf(Object) auf und ist damit abhängig von dieser (siehe Abschnitt A.2.5). Sie wird außerdem von der formalen Spezifikation der Methode Collection.add(Object) als Query verwendet (siehe Abschnitt 3.3.5) genutzt.

```
1  Returns <tt>true</tt> if this collection contains the specified
2  element. More formally, returns <tt>true</tt> if and only if this
3  collection contains at least one element <tt>e</tt> such that
4  <tt>(o==null ? e==null : o.equals(e))</tt>.
5
6  @param o element whose presence in this collection is to be tested
7  @return <tt>true</tt> if this collection contains the specified
8          element
9  @throws ClassCastException if the type of the specified element
10         is incompatible with this collection (optional)
11 @throws NullPointerException if the specified element is null and
12         this collection does not permit null elements (optional)
```

Quelltext A.1: Informelle Spezifikation von Collection.add(Object)

```
1  /*@ public normal_behavior
2    @ requires o == null;
3    @ ensures \result <==> (\exists int i;
4    @       0 <= i && i < elements.length; elements[i] == null);
5    @
6    @ also
7    @ public normal_behavior
8    @ requires o != null;
9    @ ensures \result <==> (\exists int i;
10   @       0 <= i && i < elements.length; elements[i].equals(o));
11   @
12   @ also
13   @ public behavior
14   @ signals_only ClassCastException, NullPointerException;
15   @ signals (NullPointerException e) (!supportsNull && o == null);
16   @*/
17 boolean /*@pure@*/ contains(/*@nullable@*/ Object o);
```

Quelltext A.2: Formale Spezifikation von Collection.add(Object)

A.2.2 Spezifikation von Collection.toArray()

```
1  Returns an array containing all of the elements in this collection.
2  If this collection makes any guarantees as to what order its elements
3  are returned by its iterator, this method must return the elements in
4  the same order.
5
6  <p>The returned array will be "safe" in that no references to it are
7  maintained by this collection.  (In other words, this method must
8  allocate a new array even if this collection is backed by an array).
9  The caller is thus free to modify the returned array.
10
```

https://dx.doi.org/10.24355/dbbs.084-201801311013

```
11 <p>This method acts as bridge between array-based and collection-
12 based APIs.
13
14 @return an array containing all of the elements in this collection
```

Quelltext A.3: Informelle Spezifikation von Collection.toArray()

```
1 /*@ public normal_behavior
2   @ ensures \result != null;
3   @ ensures (\forall int i; 0 <= i && i < elements.length;
4   @       (\exists int j; 0 <= j && j < \result.length;
5   @       elements[i] == \result[j]));
6   @*/
7 /*@nullable@*/ Object[] /*@pure@*/ toArray();
```

Quelltext A.4: Formale Spezifikation von Collection.toArray()

A.2.3 Spezifikation von Collection.isEmpty()

```
1 Returns <tt>true</tt> if this collection contains no elements.
2
3 @return <tt>true</tt> if this collection contains no elements
```

Quelltext A.5: Informelle Spezifikation von Collection.isEmpty()

```
1 /*@ public behavior
2   @ ensures \result == (collectionSize == 0);
3   @*/
4 boolean /*@pure@*/ isEmpty();
```

Quelltext A.6: Formale Spezifikation von Collection.toArray()

A.2.4 Spezifikation von Collection.clear()

```
1 Removes all of the elements from this collection(optional operation).
2 The collection will be empty after this method returns.
3
4 @throws UnsupportedOperationException if the <tt>clear</tt> operation
5          is not supported by this collection
```

Quelltext A.7: Informelle Spezifikation von Collection.clear()

```
1 /*@ public normal_behavior
2   @ ensures isEmpty();
3   @
4   @ also
5   @ public behavior
6   @ signals_only UnsupportedOperationException;
7   @ signals (UnsupportedOperationException e) !supportsClear;
8   @*/
9 void clear();
```

Quelltext A.8: Formale Spezifikation von Collection.clear()

https://dx.doi.org/10.24355/dbbs.084-201801311013

A.2.5 Spezifikation von List.indexOf(Object)

Diese Methode wird von der ArrayList.contains(Object) (siehe Abschnitt A.2.1) genutzt.

Die Methode `ArrayList.indexOf` konnte nicht vollautomatisch mit Hilfe des angegeben Vertrags angegeben werden. Wenn die Implementierung angepasst wird, sodass nach dem Objekt selbst und nicht nach einem gleichen Objekt gesucht wird. In diesem Fall kann der Vertrag geschlossen werden, dies passt jedoch nicht zum Vertrag. Dieser Fall sollte im Rahmen einer anschließenden Arbeit erneut betrachtet werden.

```
Returns the index of the first occurrence of the specified element
in this list, or -1 if this list does not contain the element.
More formally, returns the lowest index <tt>i</tt> such that
<tt>(o==null ? get(i)==null : o.equals(get(i)))
</tt>, or -1 if there is no such index.

@param o element to search for
@return the index of the first occurrence of the specified element in
        this list, or -1 if this list does not contain the element
@throws ClassCastException if the type of the specified element
        is incompatible with this list (optional)
@throws NullPointerException if the specified element is null and
        this list does not permit null elements (optional)
```

Quelltext A.9: Informelle Spezifikation von List.indexOf(Object)

```
/*@ public normal_behavior
  @ requires o == null;
  @ ensures (result != -1) <==> (\exists int i; 0 <= i
  @       && i < elements.length; elements[i] == null);
  @ ensures \result > -1 ==> elements[\result] == null;
  @
  @ also
  @ public normal_behavior
  @ requires o != null;
  @ ensures (result != -1) <==> (\exists int i; 0 <= i
  @       && i < elements.length; o.equals(elements[i]));
  @ ensures \result > -1 ==> o.equals(elements[\result]);
  @
  @ also
  @ public behavior
  @ ensures \result >= -1;
  @ signals_only ClassCastException, NullPointerException;
  @ signals (NullPointerException e) (!supportsNull && o == null);
  @*/
int /*@pure@*/ indexOf(/*@nullable@*/Object o);
```

Quelltext A.10: Formale Spezifikation von List.indexOf(Object)

A.2.6 Spezifikation des Konstruktors der ArrayList()

```
Constructs an empty list with an initial capacity of ten.
```

Quelltext A.11: Informelle Spezifikation von ArrayList()

https://dx.doi.org/10.24355/dbbs.084-201801311013

```
/*@ public normal_behavior
  @ ensures elementData.length == 10;
  @*/
public ArrayList();
```

Quelltext A.12: Formale Spezifikation von ArrayList()

A.2.7 Spezifikation von ArrayList.fastRemove(int)

```
Private remove method that skips bounds checking and does not
return the value removed.
```

Quelltext A.13: Informelle Spezifikation von ArrayList.fastRemove(int)

```
/*@ normal_behavior
  @ requires index < size;
  @ requires index >= 0;
  @ requires size > 0;
  @ ensures elementData[size] == null;
  @ ensures elementData == \old(elementData);
  @ ensures (\forall int i; 0 <= i && i < index;
       elementData[i] == \old(elementData[i]));
  @ ensures (\forall int i; index < i && i < size;
       elementData[i] == \old(elementData[i+1]));
  @ assignable changeable;
  @*/
private void fastRemove(int index)
```

Quelltext A.14: Formale Spezifikation von ArrayList.fastRemove(int)

A.2.8 Spezifikation von ArrayList.outOfBoundsMsg()

```
Constructs an IndexOutOfBoundsException detail message.
Of the many possible refactorings of the error handling code,
this "outlining" performs best with both server and client VMs.
```

Quelltext A.15: Informelle Spezifikation von ArrayList.outOfBoundsMsg()

```
/*@ public normal_behavior
  @ requires true;
  @*/
private /*@pure@*/ String outOfBoundsMsg(int index)
```

Quelltext A.16: Formale Spezifikation von ArrayList.outOfBoundsMsg()

A.2.9 Spezifikation von ArrayList.rangeCheck()

```
Checks if the given index is in range.  If not, throws an appropriate
runtime exception.  This method does *not* check if the index is
negative: It is always used immediately prior to an array access,
which throws an ArrayIndexOutOfBoundsException if index is negative.
```

Quelltext A.17: Informelle Spezifikation von ArrayList.rangeCheck()

https://dx.doi.org/10.24355/dbbs.084-201801311013

```
1  /*@ public behavior
2    @ signals_only IndexOutOfBoundsException;
3    @ signals (IndexOutOfBoundsException e) index >= size;
4    @*/
5  private /*@pure@*/ void rangeCheck(int index)
```

Quelltext A.18: Formale Spezifikation von ArrayList.rangeCheck()

A.2.10 Spezifikation von ArrayList.rangeCheckForAdd()

```
1  A version of rangeCheck used by add and addAll.
```

Quelltext A.19: Informelle Spezifikation von ArrayList.rangeCheckForAdd()

```
1  /*@ public behavior
2    @ signals_only IndexOutOfBoundsException;
3    @ signals (IndexOutOfBoundsException e)
4         !(0 <= index && index <= size);
5    @*/
6  private /*@pure@*/ void rangeCheckForAdd(int index)
```

Quelltext A.20: Formale Spezifikation von ArrayList.rangeCheckForAdd()

A.2.11 Spezifikation von ArrayList.trimToSize()

```
1  Trims the capacity of this <tt>ArrayList</tt> instance to be the
2  list's current size.  An application can use this operation to
3  minimize the storage of an <tt>ArrayList</tt> instance.
```

Quelltext A.21: Informelle Spezifikation von ArrayList.trimToSize()

```
1  /*@ public normal_behavior
2    @ ensures modCount != \old(modCount);
3    @ ensures elementData.length == size;
4    @ ensures (\forall int i; 0 <= i && i < size;
5         \old(elementData[i]) == elementData[i]);
6    @ assignable elementData;
7    @ assignable modCount;
8    @*/
9  public void trimToSize()
```

Quelltext A.22: Formale Spezifikation von ArrayList.trimToSize()

A.2.12 Spezifikation von Math.abs(int)

```
1  Returns the absolute value of an {@code int} value.
2  If the argument is not negative, the argument is returned.
3  If the argument is negative, the negation of the argument is
4  returned.
5
6  <p>Note that if the argument is equal to the value of
7  {@link Integer#MIN_VALUE}, the most negative representable
8  {@code int} value, the result is that same value, which is
9  negative.
10
```

https://dx.doi.org/10.24355/dbbs.084-201801311013

```
11 @param   a   the argument whose absolute value is to be determined
12 @return  the absolute value of the argument.
```

Quelltext A.23: Informelle Spezifikation von Math.abs(int)

```
1 /*@ public normal_behavior
2   @ ensures (a <  0) ==> (\result == -a);
3   @ ensures (a >= 0) ==> (\result == a);
4   @ ensures (a == Integer.MIN_VALUE) ==>
5   @       (\result == Integer.MIN_VALUE);
6   @*/
7 public static /*@pure@*/ int abs(int a){
8      return (a < 0) ? -a : a;
9 }
```

Quelltext A.24: Formale Spezifikation und Implementierung von Math.abs(int)

Bei der Verifikation dieser Methode muss beachtet werden, dass bei `Integer.MIN_VALUE` das Resultat `Integer.MIN_VALUE` erwartet wird. Das Resultat ist in diesem Fall -a, da `Integer.MIN_VALUE < 0` ist. Man erwartet zwar im ersten Moment, dass dies ein positiver Wert ist, jedoch kommt es zu einem Überlauf, wodurch das Ergebnis wieder `Integer.MIN_VALUE` ist. Damit dies verifiziert werden kann, muss KeY als *IntRules JavaSemantics* verwenden, damit der Überlauf abgebildet wird.

A.2.13 Spezifikation von Math.min(int, int)

```
1 Returns the smaller of two {@code int} values. That is,
2 the result the argument closer to the value of
3 {@link Integer#MIN_VALUE}.  If the arguments have the same
4 value, the result is that same value.
5
6 @param   a   an argument.
7 @param   b   another argument.
8 @return  the smaller of {@code a} and {@code b}.
```

Quelltext A.25: Informelle Spezifikation von Math.min(int, int)

```
1 /*@ public normal_behavior
2   @ ensures a <= b ==> \result == a;
3   @ ensures b <= a ==> \result == b;
4   @*/
5 public static int min(int a, int b)
```

Quelltext A.26: Formale Spezifikation von Math.min(int, int)

A.2.14 Spezifikation von Math.max(int, int)

```
1 Returns the greater of two {@code int} values. That is, the
2 result is the argument closer to the value of
3 {@link Integer#MAX_VALUE}. If the arguments have the same value,
4 the result is that same value.
5
6 @param   a   an argument.
7 @param   b   another argument.
```

https://dx.doi.org/10.24355/dbbs.084-201801311013

```
8 @return  the larger of {@code a} and {@code b}.
```

Quelltext A.27: Informelle Spezifikation von Math.max(int, int)

```
1 /*@ public normal_behavior
2   @ ensures a >= b ==> \result == a;
3   @ ensures b >= a ==> \result == b;
4   @*/
5 public static int max(int a, int b)
```

Quelltext A.28: Formale Spezifikation von Math.max(int, int)

https://dx.doi.org/10.24355/dbbs.084-201801311013

A.3 Weitere JavaDoc-Einträge

```
An ordered collection (also known as a <i>sequence</i>).  The user of
this interface has precise control over where in the list each
element is inserted.  The user can access elements by their
integer index (position in the list), and search for elements in the
list.<p>

Unlike sets, lists typically allow duplicate elements.  More
formally, lists typically allow pairs of elements <tt>e1</tt> and
<tt>e2</tt> such that <tt>e1.equals(e2)</tt>, and they typically
allow multiple null elements if they allow null elements at all.  It
is not inconceivable that someone might wish to implement a list
that prohibits duplicates, by throwing runtime exceptions when
the user attempts to insert them, but we expect this usage to be
rare. <p>

The <tt>List</tt> interface places additional stipulations, beyond
those specified in the <tt>Collection</tt> interface, on the
contracts of the <tt>iterator</tt>, <tt>add</tt>, <tt>remove</tt>,
<tt>equals</tt>, and <tt>hashCode</tt> methods.  Declarations
for other inherited methods are also included here for convenience.
<p>

The <tt>List</tt> interface provides four methods for positional
(indexed) access to list elements.  Lists (like Java arrays) are zero
based.  Note that these operations may execute in time proportional
to the index value for some implementations (the <tt>LinkedList</tt>
class, for example). Thus, iterating over the elements in a list is
typically preferable to indexing through it if the caller does not
know the implementation.<p>

The <tt>List</tt> interface provides a special iterator, called a
<tt>ListIterator</tt>, that allows element insertion and replacement,
and bidirectional access in addition to the normal operations that
the <tt>Iterator</tt> interface provides.  A method is provided to
obtain a list iterator that starts at a specified position in the
list.<p>

The <tt>List</tt> interface provides two methods to search for a
specified object.  From a performance standpoint, these methods
should be used with caution.  In many implementations they will
perform costly linear searches.<p>

The <tt>List</tt> interface provides two methods to efficiently
insert and remove multiple elements at an arbitrary point in the
list.<p>

Note: While it is permissible for lists to contain themselves as
elements, extreme caution is advised: the <tt>equals</tt> and
<tt>hashCode</tt> methods are no longer well defined on such a list.

<p>Some list implementations have restrictions on the elements that
they may contain.  For example, some implementations prohibit null
elements, and some have restrictions on the types of their elements.
Attempting to add an ineligible element throws an unchecked
exception, typically <tt>NullPointerException</tt> or
```

https://dx.doi.org/10.24355/dbbs.084-201801311013

```
56 <tt>ClassCastException</tt>.  Attempting to query the presence
57 of an ineligible element may throw an exception, or it may simply
58 return false; some implementations will exhibit the former
59 behavior and some will exhibit the latter.  More generally,
60 attempting an operation on an ineligible element whose completion
61 would not result in the insertion of an ineligible element into the
62 list may throw an exception or it may succeed, at the option of the
63 implementation. Such exceptions are marked as "optional" in the
64 specification for this interface.
65
66 <p>This interface is a member of the
67 <a href="{@docRoot}/../technotes/guides/collections/index.html">
68 Java Collections Framework</a>.
69
70 @author  Josh Bloch
71 @author  Neal Gafter
72 @see Collection
73 @see Set
74 @see ArrayList
75 @see LinkedList
76 @see Vector
77 @see Arrays#asList(Object[])
78 @see Collections#nCopies(int, Object)
79 @see Collections#EMPTY_LIST
80 @see AbstractList
81 @see AbstractSequentialList
82 @since 1.2
```

Quelltext A.29: JavaDoc zur List-Klasse

```
1  Resizable-array implementation of the <tt>List</tt> interface.
2  Implements all optional list operations, and permits all elements,
3  including <tt>null</tt>.  In addition to implementing the
4  <tt>List</tt> interface, this class provides methods to manipulate
5  the size of the array that is used internally to store the list.
6  (This class is roughly equivalent to <tt>Vector</tt>, except
7  that it is unsynchronized.)
8
9  <p>The <tt>size</tt>, <tt>isEmpty</tt>, <tt>get</tt>, <tt>set</tt>,
10 <tt>iterator</tt>, and <tt>listIterator</tt> operations run in
11 constant time.  The <tt>add</tt> operation runs in <i>amortized
12 constant time</i>, that is, adding n elements requires O(n) time.
13 All of the other operations run in linear time (roughly speaking).
14 The constant factor is low compared to that for the
15 <tt>LinkedList</tt> implementation.
16
17 <p>Each <tt>ArrayList</tt> instance has a <i>capacity</i>.  The
18 capacity is the size of the array used to store the elements in the
19 list. It is always at least as large as the list size.  As elements
20 are added to an ArrayList, its capacity grows automatically.  The
21 details of the growth policy are not specified beyond the fact that
22 adding an element has constant amortized time cost.
23
24 <p>An application can increase the capacity of an <tt>ArrayList</tt>
25 instance before adding a large number of elements using the
26 <tt>ensureCapacity</tt> operation.  This may reduce the amount of
27 incremental reallocation.
28
29 <p><strong>Note that this implementation is not synchronized.
30 </strong> If multiple threads access an <tt>ArrayList</tt> instance
31 concurrently, and at least one of the threads modifies the list
32 structurally, it <i>must</i> be synchronized externally.
33 (A structural modification is any operation that adds or deletes
34 one or more elements, or explicitly resizes the backing array; merely
35 setting the value of an element is not a structural modification.)
36 This is typically accomplished by synchronizing on some object that
37 naturally encapsulates the list.
38
39 If no such object exists, the list should be "wrapped" using the
40 {@link Collections#synchronizedList Collections.synchronizedList}
41 method.  This is best done at creation time, to prevent accidental
42 unsynchronized access to the list:<pre> List list =
43 Collections.synchronizedList(new ArrayList(...));</pre>
44
45 <p><a name="fail-fast"/> The iterators returned by this class's
46 {@link #iterator() iterator} and {@link #listIterator(int)
47 listIterator} methods are <em>fail-fast</em>: if the list is
48 structurally modified at any time after the iterator is created, in
49 any way except through the iterator's own
50 {@link ListIterator#remove() remove} or
51 {@link ListIterator#add(Object) add} methods, the iterator will
52 throw a {@link ConcurrentModificationException}.  Thus, in the face
53 of concurrent modification, the iterator fails quickly and cleanly,
54 rather than risking arbitrary, non-deterministic behavior at an
55 undetermined time in the future.
56
```

https://dx.doi.org/10.24355/dbbs.084-201801311013

```
57 <p>Note that the fail-fast behavior of an iterator cannot be
58 guaranteed as it is, generally speaking, impossible to make any
59 hard guarantees in the presence of unsynchronized concurrent
60 modification.  Fail-fast iterators throw {@code
61 ConcurrentModificationException} on a best-effort basis. Therefore,
62 it would be wrong to write a program that depended on this exception
63 for its correctness:  <i>the fail-fast behavior of iterators should
64 be used only to detect bugs.</i>
65
66 <p>This class is a member of the <a href="{@docRoot}/../technotes/
67 guides/collections/index.html"> Java Collections Framework</a>.
68
69 @author  Josh Bloch
70 @author  Neal Gafter
71 @see     Collection
72 @see     List
73 @see     LinkedList
74 @see     Vector
75 @since   1.2
```

Quelltext A.30: JavaDoc zur ArrayList-Klasse

https://dx.doi.org/10.24355/dbbs.084-201801311013

```
1  The number of times this list has been <i>structurally modified</i>.
2  Structural modifications are those that change the size of the
3  list, or otherwise perturb it in such a fashion that iterations in
4  progress may yield incorrect results.
5
6  <p>This field is used by the iterator and list iterator
7  implementation returned by the {@code iterator} and {@code
8  listIterator} methods. If the value of this field changes
9  unexpectedly, the iterator (or list iterator) will throw a
10 {@code ConcurrentModificationException} in
11 response to the {@code next}, {@code remove}, {@code previous},
12 {@code set} or {@code add} operations.  This provides
13 <i>fail-fast</i> behavior, rather than non-deterministic behavior in
14 the face of concurrent modification during iteration.
15
16 <p><b>Use of this field by subclasses is optional.</b> If a subclass
17 wishes to provide fail-fast iterators (and list iterators), then it
18 merely has to increment this field in its {@code add(int, E)} and
19 {@code remove(int)} methods (and any other methods that it overrides
20 that result in structural modifications to the list).  A single call
21 to {@code add(int, E)} or {@code remove(int)} must add no more than
22 one to this field, or the iterators (and list iterators) will throw
23 bogus {@code ConcurrentModificationExceptions}.  If an implementation
24 does not wish to provide fail-fast iterators, this field may be
25 ignored.
```

Quelltext A.31: JavaDoc des modCount-Felds der AbstractList-Klasse

https://dx.doi.org/10.24355/dbbs.084-201801311013

A.4 Feature-Modelle aus FeatureIDE

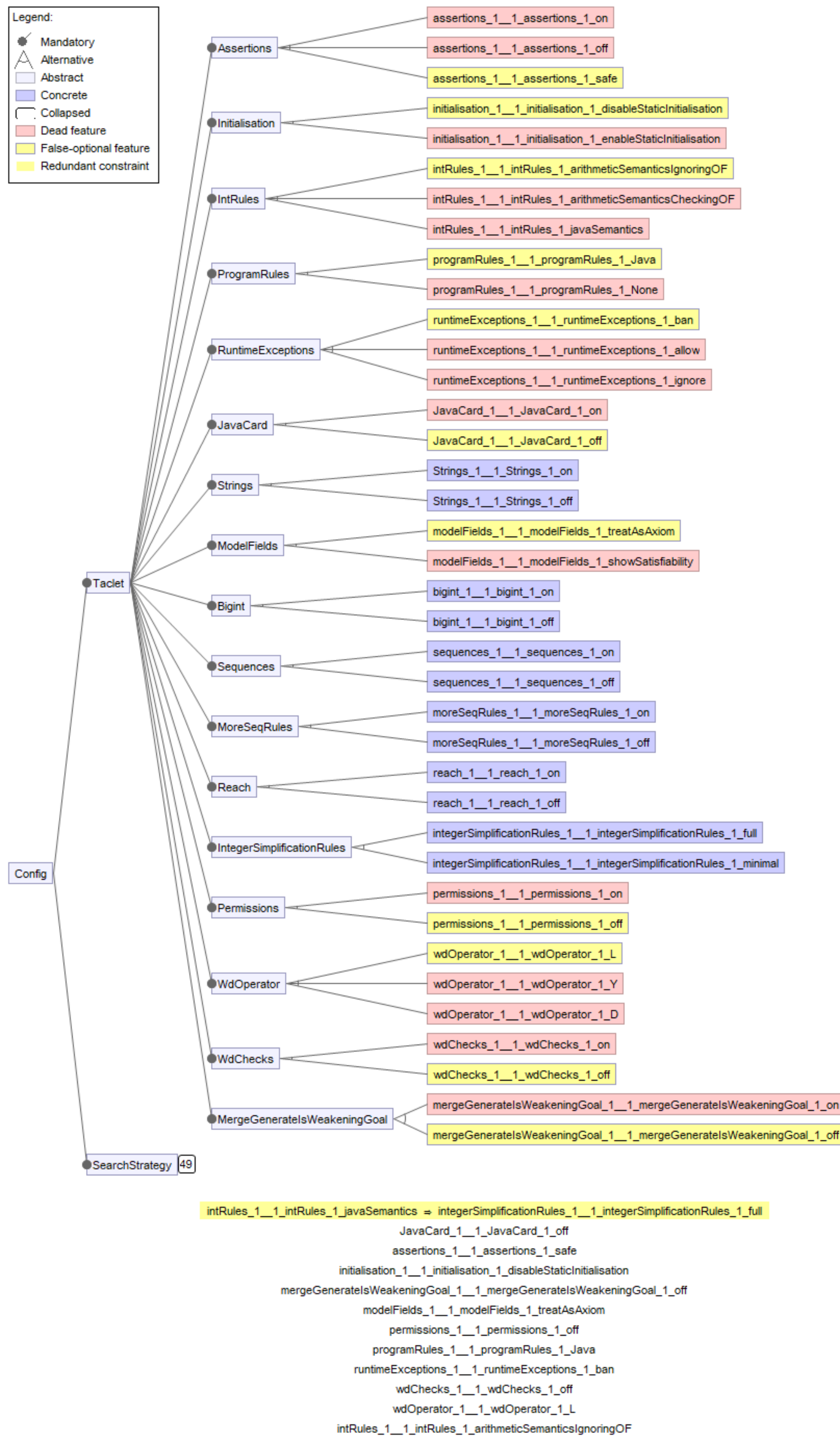

Abbildung A.1: Feature-Modell mit arithmeticSematicsIgnoringOF (Taclet Optionen)

https://dx.doi.org/10.24355/dbbs.084-201801311013

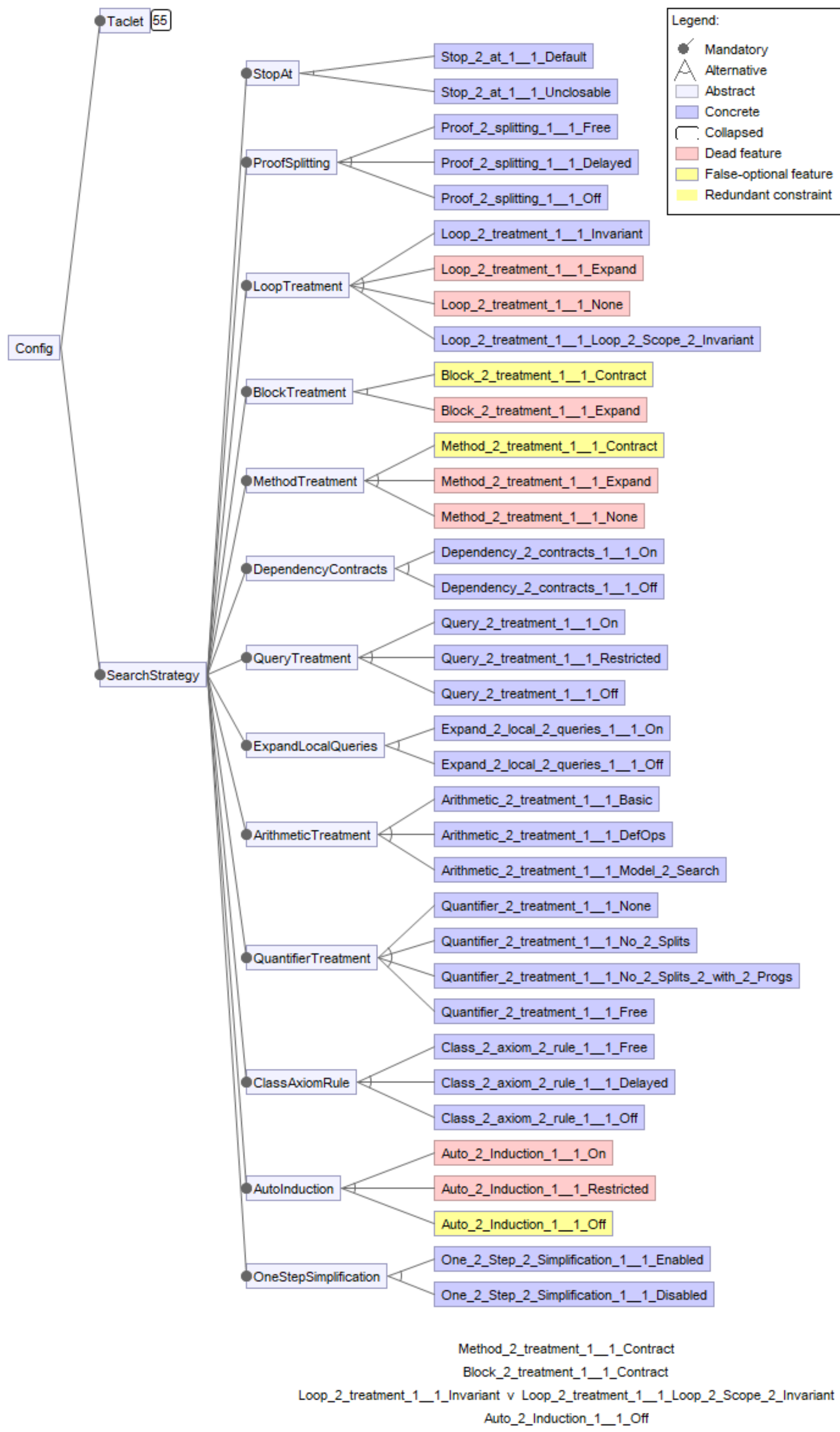

Abbildung A.2: Feature-Modell mit arithmeticSematicsIgnoringOF (Search Strategy Optionen)

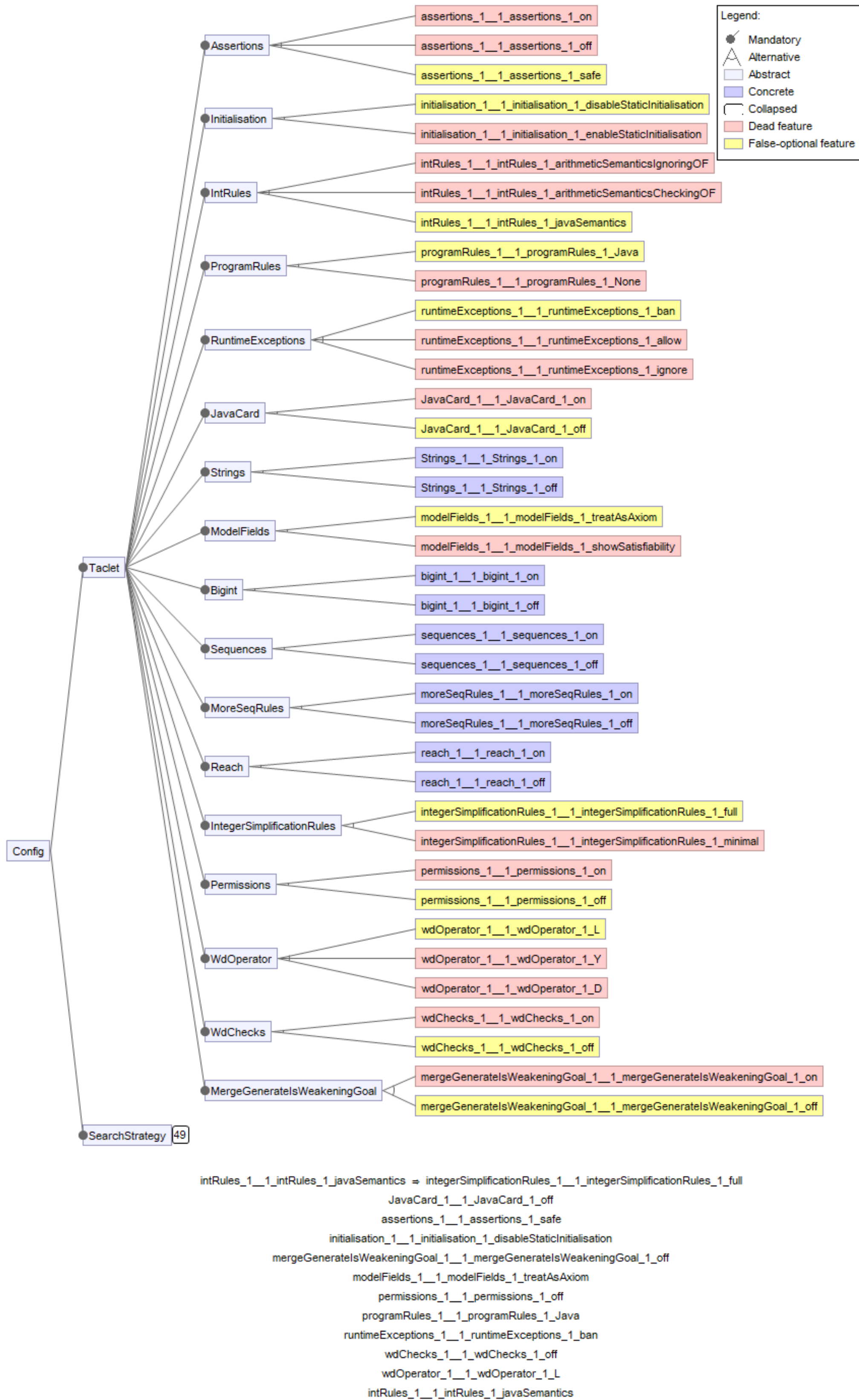

Abbildung A.3: Feature-Modell mit javaSemantics (Taclet Optionen)

https://dx.doi.org/10.24355/dbbs.084-201801311013

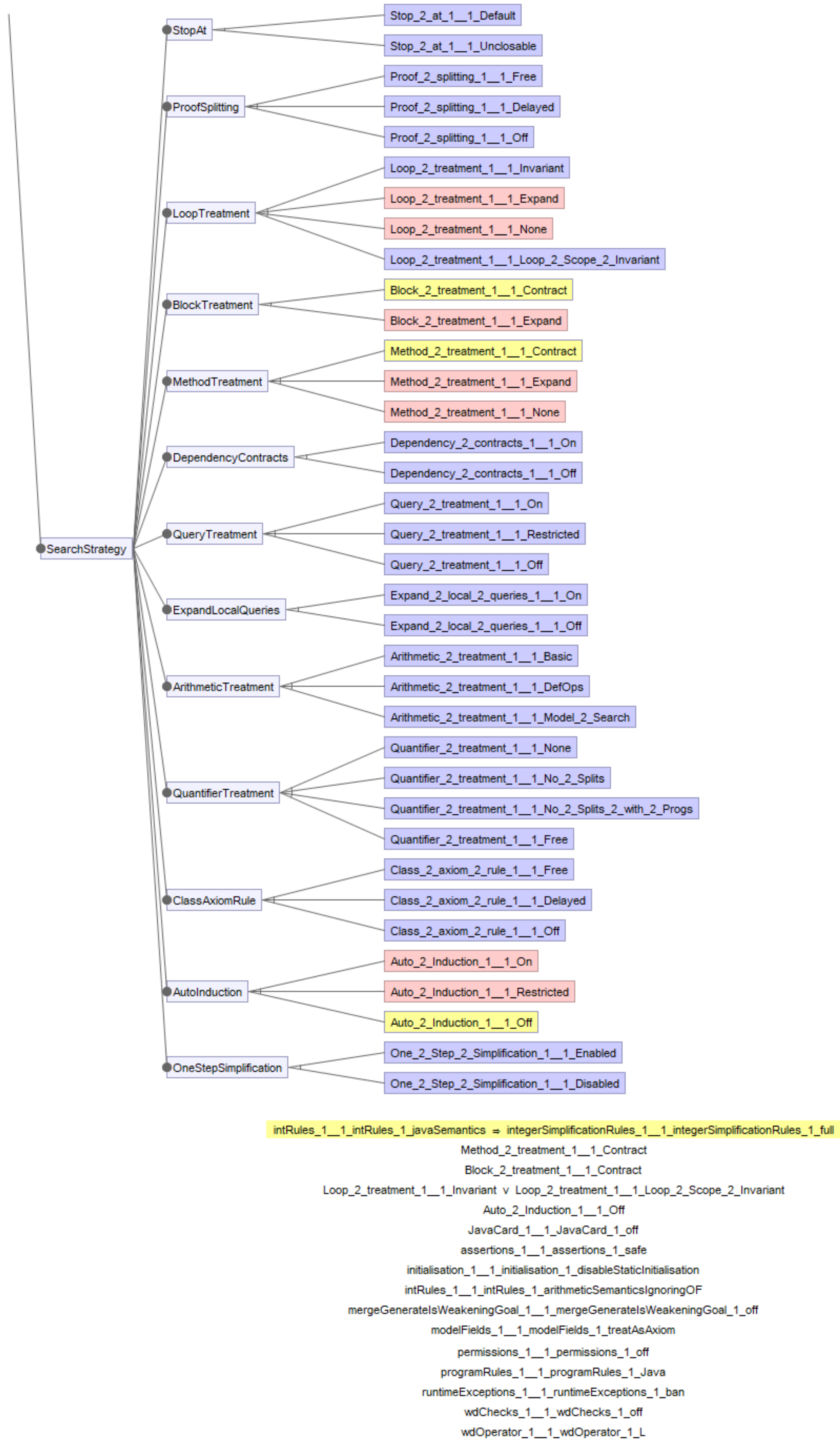

Abbildung A.4: Feature-Modell mit javaSemantics (Search Strategy Optionen)

https://dx.doi.org/10.24355/dbbs.084-201801311013

A.5 Speicherverbrauch von KeY

Während der Arbeit mit KeY ist aufgefallen, dass die Ausführung von Beweisen nach längerer Laufzeit der virtuellen Maschine deutlich länger gedauert hat, auch wenn der Aufwand des Beweises selbst nicht gestiegen ist. Um dieses Phänomen zu untersuchen, wurde in einer virtuellen Maschine derselbe Beweis in einer Dauerschleife wiederholt. Dies wurde mit Java Mission Control und dem Java Flight Recorder eine Stunde lang ausgewertet [Oak14, S. 59f]. Die Speicherauswertung wurde in Abbildung A.5 dargestellt. Bis etwa Minute 26 erkennt man ein ansteigendes Sägezahnmuster. Das Muster ist dadurch zu erklären, dass der Java-Garbagecollector einen Teil des Speichers wieder freigeben kann. Allerdings erreicht der Speicher nach der Ausführung des Garbagecollectors nicht mehr die Auslastung, die nach der letzten Ausführung erreicht wurde. Dadurch wird der Speicher immer stärker ausgelastet, was zu immer mehr Garbagecollection-Pausen führt (siehe Abbildung A.6). Dieses Verhalten lässt ein Speicherleck in KeY vermuten. Eine solche Untersuchung würde jedoch den Umfang dieser Arbeit übersteigen.

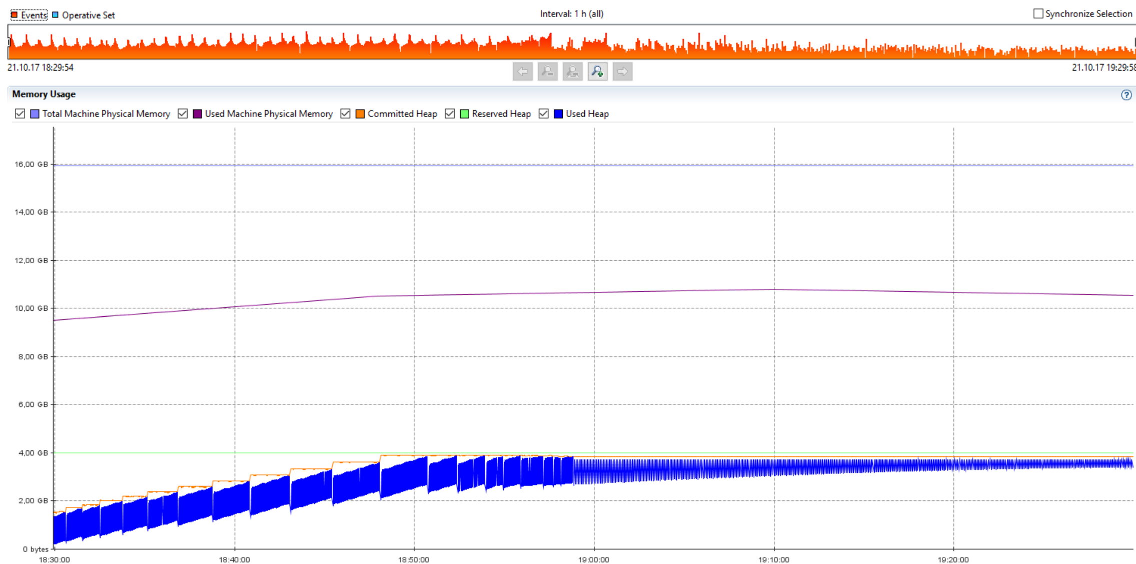

Abbildung A.5: Darstellung des Speicherverbrauchs von KeY

Abbildung A.6: Darstellung des Garbagecollectors bei der Ausführung von KeY

https://dx.doi.org/10.24355/dbbs.084-201801311013

A.6 Modelle zum Beweisaufwand

Im Rahmen der Modellerstellung mit SPLConqueror wurden zehn Modelle erzeugt. Diese Modelle werden in Tabelle A.1 und Tabelle A.2 dargestellt. In der ersten Spalte werden jeweils die Optionen gelistet, gefolgt von den Faktoren des jeweiligen Modells. Dies gilt jedoch nicht für die erste und die letzte Zeile, in der die Nummer des Modells, beziehungsweise der jeweilige Testfehler des Modells dargestellt wird.

Modell	1	2	3	4	5
root	643	308	789	678	239
Arithmetic treatment::Basic	32	-119	-142	-79	-22
Class axiom rule::Delayed	1253	1078	1100	1166	1056
Expand local queries::Off	0	0	0	83	0
Loop treatment::Invariant	0	142	0	0	0
Loop treatment::Loop Scope Invariant	0	166	0	13	0
One Step Simplification::Disabled	0	159	61	0	0
One Step Simplification::Enabled	0	0	0	-2	0
Proof splitting::Delayed	0	0	0	-162	236
Proof splitting::Free	0	-141	0	0	298
Proof splitting::Off	-647	-509	-570	-664	-295
Quantifier treatment::Free	0	-32	-54	0	0
Query treatment::Restricted	0	-146	-63	0	0
Stop at::Default	0	0	0	0	123
Stop at::Unclosable	0	0	0	0	116
Strings::Strings:off	0	0	-7	0	0
bigint::bigint:off	0	106	-242	0	0
bigint::bigint:on	0	202	0	0	0
moreSeqRules::moreSeqRules:off	0	0	0	0	0
moreSeqRules::moreSeqRules:on	0	0	32	0	0
reach::reach:on	0	0	-233	0	0
TestError	629	525	521	549	573

Tabelle A.1: Darstellung der erzeugten Modelle 1 bis 5

https://dx.doi.org/10.24355/dbbs.084-201801311013

Modell	6	7	8	9	10
root	557	243	235	542	650
Arithmetic treatment::Basic	-60	-126	-75	-75	-73
Class axiom rule::Delayed	1182	1176	1167	1164	1173
Expand local queries::Off	0	0	0	0	40
Loop treatment::Invariant	0	0	0	0	0
Loop treatment::Loop Scope Invariant	0	0	0	-8	0
One Step Simplification::Disabled	0	172	0	57	56
One Step Simplification::Enabled	0	71	0	0	0
Proof splitting::Delayed	0	192	218	0	-113
Proof splitting::Free	105	308	326	101	0
Proof splitting::Off	-526	-257	-309	-522	-637
Quantifier treatment::Free	0	0	0	0	0
Query treatment::Restricted	0	0	0	0	-107
Stop at::Default	0	0	0	0	0
Stop at::Unclosable	0	0	0	0	0
Strings::Strings:off	0	0	0	0	0
bigint::bigint:off	0	0	0	0	0
bigint::bigint:on	0	0	0	0	0
moreSeqRules::moreSeqRules:off	0	0	109	0	0
moreSeqRules::moreSeqRules:on	0	0	126	0	0
reach::reach:on	0	0	0	0	0
TestError	567	626	553	525	521

Tabelle A.2: Darstellung der erzeugten Modelle 6 bis 10

https://dx.doi.org/10.24355/dbbs.084-201801311013

A.7 Betrachtung der Qualifikation des Autors als Softwareingenieur

Diese Arbeit soll aus der Sicht eines Softwareingenieurs angefertigt werden. Als Begründung für meine Qualifikation soll hierbei mein beruflicher Werdegang genannt werden.

Neben dem Studium als Informatiker kann zusätzlich eine parallele Ausbildung als Fachinformatiker für Anwendungsentwicklung und damit einhergehend eine dreijährige Berufserfahrung als Softwareentwickler nachgewiesen werden. In diesem Zusammenhang war ich zwei Jahre als Softwareentwickler im *JavaEE*- und *COBOL*-Umfeld als externer Mitarbeiter für die Volkswagen AG tätig. Anschließend war ich zwei Jahre Werksstudent für die Volkswagen Financial Services AG, wo ich im Rahmen eines Projekts die Weiterentwicklung eines bestehenden Legacy-Systems mit übernommen habe. Anschließend bin ich dort seit 11 Monaten als Systemanalytiker beschäftigt, übernehme selbstständig die Architektur und die Weiterentwicklung eines bestehenden Softwaresystems und bin stellvertretender Teamleiter.

Auf Grund dieser 6 Jahre Berufserfahrung als Softwareentwickler bin ich hinreichend qualifiziert, um die Betrachtungen aus Sicht eines Softwareingenieurs durchzuführen.

https://dx.doi.org/10.24355/dbbs.084-201801311013

https://dx.doi.org/10.24355/dbbs.084-201801311013

Literaturverzeichnis

[ABB+14] Wolfgang Ahrendt, Bernhard Beckert, Daniel Bruns, Richard Bubel, Christoph Gladisch, Sarah Grebing, Reiner Hähnle, Martin Hentschel, Mihai Herda, Vladimir Klebanov, Wojciech Mostowski, Christoph Scheben, Peter H. Schmitt, and Mattias Ulbrich. *The KeY Platform for Verification and Analysis of Java Programs*, pages 55–71. Springer International Publishing, Cham, 2014. (zitiert auf Seite 95)

[ABB+16] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich (Eds.). *Deductive Software Verification – The KeY Book*. Springer, 2016. (zitiert auf Seite vii, 1, 2, 8, 14, 15, 16, 32, 46, 47, 48, 49, 50, 53, 54, 59, 62, 63, 64, 66 und 95)

[BBBB12] Christoph Baumann, Bernhard Beckert, Holger Blasum, and Thorsten Bormer. Lessons learned from microkernel verification–specification is the new bottleneck. *arXiv preprint arXiv:1211.6186*, 2012. (zitiert auf Seite 96)

[BBG16] Bernhard Beckert, Thorsten Bormer, and Daniel Grahl. *Deductive Verification of Legacy Code*, pages 749–765. Springer International Publishing, Cham, 2016. (zitiert auf Seite 96)

[BBM98] Patrick Behm, Lilian Burdy, and Jean Marc Meynadier. *Well defined B*, pages 29–45. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998. (zitiert auf Seite 64)

[BCD+05] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K Rustan M Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO*, volume 5, pages 364–387. Springer, 2005. (zitiert auf Seite 9, 95 und 103)

[BHS13] Thomas Baar, Reiner Hähnle, and Steffen Schlager. Key quicktour for jml. http://i12www.ira.uka.de/key/download/quicktour/quicktour-2.0.zip, 2013. [Online; zugegriffen am 19.06.2017]. (zitiert auf Seite 15, 44 und 53)

[BLS05] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. *The Spec# Programming System: An Overview*, pages 49–69. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005. (zitiert auf Seite 9, 95 und 103)

https://dx.doi.org/10.24355/dbbs.084-201801311013

[BRR08] Richard Bubel, Andreas Roth, and Philipp Rümmer. Ensuring the correctness of lightweight tactics for javacard dynamic logic. *Electronic Notes in Theoretical Computer Science*, 199:107 – 128, 2008. Proceedings of the Fourth International Workshop on Logical Frameworks and Meta-Languages (LFM 2004). (zitiert auf Seite 15)

[CDH+09] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. *VCC: A Practical System for Verifying Concurrent C*, pages 23–42. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. (zitiert auf Seite 95)

[Che00] Zhiqun Chen. *Java card technology for smart cards: architecture and programmer's guide*. Addison-Wesley Professional, 2000. (zitiert auf Seite 6)

[CKLP06] Patrice Chalin, Joseph R. Kiniry, Gary T. Leavens, and Erik Poll. *Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2*, pages 342–363. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006. (zitiert auf Seite 1)

[Cok11] David R. Cok. *OpenJML: JML for Java 7 by Extending OpenJDK*, pages 472–479. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. (zitiert auf Seite 2 und 95)

[Cok14] David R Cok. Openjml: software verification for java 7 using jml, openjdk, and eclipse. *arXiv preprint arXiv:1404.6608*, 2014. (zitiert auf Seite 95)

[dGdBB+17] Stijn de Gouw, Frank S. de Boer, Richard Bubel, Reiner Hähnle, Jurriaan Rot, and Dominic Steinhöfel. Verifying openjdk's sort method for generic collections. *Journal of Automated Reasoning*, Aug 2017. (zitiert auf Seite 96)

[dGRdB+15] Stijn de Gouw, Jurriaan Rot, Frank S. de Boer, Richard Bubel, and Reiner Hähnle. Openjdk's java.utils.collection.sort() is broken: The good, the bad and the worst case. In *CAV*, pages 273 – 289, 2015. (zitiert auf Seite 2 und 96)

[dMB08] Leonardo de Moura and Nikolaj Bjørner. *Z3: An Efficient SMT Solver*, pages 337–340. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. (zitiert auf Seite 66)

[DMR08] Ádám Darvas, Farhad Mehta, and Arsenii Rudich. *Efficient Well-Definedness Checking*, pages 100–115. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. (zitiert auf Seite 64)

[DS13] Linda DeMichiel and Bill Shannon. *Java^{TM} Platform, Enterprise Edition (Java EE) Specification, v7.* 2013. (zitiert auf Seite 6)

https://dx.doi.org/10.24355/dbbs.084-201801311013

[Fil11] Jean-Christophe Filliâtre. Deductive software verification. *International Journal on Software Tools for Technology Transfer*, 13(5):397, Aug 2011. (zitiert auf Seite 8)

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java^{TM}Language Specification.* Addison-Wesley, third edition, 2005. (zitiert auf Seite 1, 2, 6, 7, 8, 9, 14, 39, 40, 41, 59 und 60)

[Gla12] Christoph D. Gladisch. Model generation for quantified formulas with application to test data generation. *International Journal on Software Tools for Technology Transfer*, 14(4):439–459, 2012. (zitiert auf Seite 53)

[HLL+12] John Hatcliff, Gary T. Leavens, K. Rustan M. Leino, Peter Müller, and Matthew Parkinson. Behavioral interface specification languages. *ACM Comput. Surv.*, 44(3):16:1–16:58, June 2012. (zitiert auf Seite 9)

[HM15] M. Huisman and W. Mostowski. A symbolic approach to permission accounting for concurrent reasoning. In *2015 14th International Symposium on Parallel and Distributed Computing*, pages 165–174, June 2015. (zitiert auf Seite 64)

[HP04] Engelbert Hubbers and Erik Poll. *Reasoning about Card Tears and Transactions in Java Card*, pages 114–128. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. (zitiert auf Seite 56 und 64)

[ISO10] ISO/IEC. ISO/IEC 25010 - Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models. Technical report, 2010. (zitiert auf Seite 1)

[j14] Jsr 59: J2se^{TM} merlin release contents. https://www.jcp.org/en/jsr/detail?id=59. [Online; zugegriffen am 14.09.2017]. (zitiert auf Seite 6)

[j15] Jsr 176: J2se^{TM} 5.0 (tiger) release contents. http://www.oracle.com/technetwork/java/javase/relnotes-139183.html. [Online; zugegriffen am 14.09.2017]. (zitiert auf Seite 6)

[j16] Jsr 270: Java^{TM} se 6 release contents. https://www.jcp.org/en/jsr/detail?id=270. [Online; zugegriffen am 14.09.2017]. (zitiert auf Seite 6)

[j17] Jsr 336: Java^{TM} se 7 release contents. https://www.jcp.org/en/jsr/detail?id=336. [Online; zugegriffen am 14.09.2017]. (zitiert auf Seite 6)

[j18] Jsr 337: Java^{TM} se 8 release contents. https://www.jcp.org/en/jsr/detail?id=337. [Online; zugegriffen am 14.09.2017]. (zitiert auf Seite 6)

https://dx.doi.org/10.24355/dbbs.084-201801311013

[java] How to write doc comments for the javadoc tool. http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html. [Online; zugegriffen am 12.08.2017]. (zitiert auf Seite 7)

[javb] Javadoc tool. http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html. [Online; zugegriffen am 17.09.2017]. (zitiert auf Seite 7)

[jcp] Jsr 41: A simple assertion facility. https://www.jcp.org/en/jsr/detail?id=337. [Online; zugegriffen am 14.09.2017]. (zitiert auf Seite 59)

[jPr] Jsr 334: Small enhancements to the javatm programming language. https://jcp.org/en/jsr/detail?id=334. [Online; zugegriffen am 14.09.2017]. (zitiert auf Seite 6)

[K+95] Ron Kohavi et al. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Ijcai*, volume 14, pages 1137–1145. Stanford, CA, 1995. (zitiert auf Seite 85)

[Key] Key. http://i12www.ira.uka.de/~key/download/nightly/. [Online; zugegriffen am 31.08.2017]. (zitiert auf Seite 44 und 56)

[KMS+11] Vladimir Klebanov, Peter Müller, Natarajan Shankar, Gary T. Leavens, Valentin Wüstholz, Eyad Alkassar, Rob Arthan, Derek Bronish, Rod Chapman, Ernie Cohen, Mark Hillebrand, Bart Jacobs, K. Rustan M. Leino, Rosemary Monahan, Frank Piessens, Nadia Polikarpova, Tom Ridge, Jan Smans, Stephan Tobies, Thomas Tuerk, Mattias Ulbrich, and Benjamin Weiß. *The 1st Verified Software Competition: Experience Report*, pages 154–168. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. (zitiert auf Seite 2)

[LBR06] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of jml: A behavioral interface specification language for java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38, May 2006. (zitiert auf Seite 1, 9 und 12)

[LPC+13] Gary T Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, Joseph Kiniry, Patrice Chalin, Daniel M Zimmerman, et al. Jml reference manual. *Draft revision*, 1:200, 2013. (zitiert auf Seite 1, 9, 10, 11, 12, 13, 14, 50 und 59)

[LW94] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, November 1994. (zitiert auf Seite 7)

[LYBB15] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java^{TM} Virtual Machine Specification*. Addison-Wesley, java se8 edition, 2015. (zitiert auf Seite 2)

[Mey88] Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall International, 1988. (zitiert auf Seite 2)

https://dx.doi.org/10.24355/dbbs.084-201801311013

[MKR+16] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. A comparison of 10 sampling algorithms for configurable systems. *CoRR*, abs/1602.02052, 2016. (zitiert auf Seite 73)

[Mos07] Wojciech Mostowski. Fully verified java card api reference implementation. *Verify*, 7, 2007. (zitiert auf Seite 96)

[MTS+17] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. *Mastering software variability with FeatureIDE*. Springer, Cham, 2017. Access date: 2017-10-29. (zitiert auf Seite 44, 73 und 74)

[Oak14] Scott Oaks. *Java performance : the definitive guide ; getting the most of your code*. O'Reilly, Beijing u.a., 1. aufl edition, 2014. Access date: 2017-10-31. (zitiert auf Seite 128)

[obl] Oracle binary code license agreement for the java se platform products and javafx. http://www.oracle.com/technetwork/java/javase/terms/license/index.html. [Online; zugegriffen am 30.07.2017]. (zitiert auf Seite 18)

[Onw00] Anthony J Onwuegbuzie. Expanding the framework of internal and external validity in quantitative research. 2000. (zitiert auf Seite 85, 87 und 88)

[opea] Openjdk homepage. [Online; zugegriffen am 29.07.2017]. (zitiert auf Seite 18)

[opeb] Openjdk hotspot repository. http://hg.openjdk.java.net/jdk7/modules/hotspot/file/9646293b9637. [Online; zugegriffen am 16.09.2017]. (zitiert auf Seite 7)

[Oraa] Oracle. Gnu general public license, version 2, with the classpath exception. http://openjdk.java.net/legal/gplv2+ce.html. [Online; zugegriffen am 30.07.2017]. (zitiert auf Seite 105)

[Orab] Oracle. The java hotspot performance engine architecture. http://www.oracle.com/technetwork/java/whitepaper-135217.html. [Online; zugegriffen am 11.06.2017]. (zitiert auf Seite 7)

[Orac] Oracle. Java platform, micro edition (java me). http://www.oracle.com/technetwork/java/embedded/javame/index.html. [Online; zugegriffen am 14.09.2017]. (zitiert auf Seite 6)

[Orad] Oracle. Java se. http://www.oracle.com/technetwork/java/javase/overview/index.html. [Online; zugegriffen am 14.09.2017]. (zitiert auf Seite 6)

[Orae] Oracle. Oracle java embedded - documentation. http://www.oracle.com/technetwork/java/embedded/documentation/javaembedded-documentation-1981555.html. [Online; zugegriffen am 14.09.2017]. (zitiert auf Seite 6)

https://dx.doi.org/10.24355/dbbs.084-201801311013

[Oraf] Oracle. Oracle technology network for java developers. http://www.oracle.com/technetwork/java/index.html. [Online; zugegriffen am 14.09.2017]. (zitiert auf Seite 6)

[Ora15] Oracle. Java™ platform, standard edition 6 api specification. https://docs.oracle.com/javase/6/docs/api/, 2015. [Online; zugegriffen am 01.06.2017]. (zitiert auf Seite 1)

[Ora17] Oracle. Collection (java se9 & jdk 9). https://docs.oracle.com/javase/9/docs/api/java/util/Collection.html, 2017. [Online; zugegriffen am 26.11.2017]. (zitiert auf Seite 36)

[Row16] Philip Rowe. *Essential Statistics for the Pharmaceutical Sciences*. John Wiley & Sons, second edition, 2016. (zitiert auf Seite 74 und 75)

[rPr] R: The r project for statistical computing. https://www.r-project.org/. [Online; zugegriffen am 23.11.2017]. (zitiert auf Seite 75)

[Rü10] Bernhard Rüger. *Test- und Schätztheorie : Band II: Statistische Tests*. VWL 6-2010. Oldenbourg Wissenschaftsverlag, München, 2010. Access date: 2017-11-08. (zitiert auf Seite 75)

[Sch15] Dominic Scheurer. From trees to dags: A general lattice model for symbolic execution. Master's thesis, 2015. (zitiert auf Seite 64)

[SHB16] Dominic Scheurer, Reiner Hähnle, and Richard Bubel. *A General Lattice Model for Merging Symbolic Execution Branches*, pages 57–73. Springer International Publishing, Cham, 2016. (zitiert auf Seite 49, 50 und 102)

[SRK98] Manfred Sommer, Werner Remmele, and Konrad Klockner. *Interaktion im Web - Innovative Kommunikationsformen*. Vieweg+Teubner Verlag, 1998. (zitiert auf Seite 6)

[SRK+12] Norbert Siegmund, Marko Rosenmüller, Martin Kuhlemann, Christian Kästner, Sven Apel, and Gunter Saake. Spl conqueror: Toward optimization of non-functional properties in software product lines. *Software Quality Journal*, 20(3):487–517, Sep 2012. (zitiert auf Seite 84)

[SSZ+15] Wenhui Sun, Yuting Sun, Zhifei Zhang, Jingpeng Tang, Kendall E. Nygard, and Damian Lampl. Specification and verification of garbage collector by java modeling language. In *FUTURE COMPUTING 2015 : The Seventh International Conference on Future Computational Technologies and Applications*, pages 18–23, 2015. (zitiert auf Seite 2 und 96)

[TSAH12] Thomas Thüm, Ina Schaefer, Sven Apel, and Martin Hentschel. Family-based deductive verification of software product lines. In *ACM SIGPLAN Notices*, volume 48, pages 11–20. ACM, 2012. (zitiert auf Seite 79)

https://dx.doi.org/10.24355/dbbs.084-201801311013

[Wei11] Benjamin Weiß. *Deductive Verification of Object-Oriented Software.* KIT Scientific Publishing, 2011. (zitiert auf Seite 8 und 10)

[WRH+00] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Software Engineering: An Introduction.* Kluwer Academic Publishers, Norwell, MA, USA, 2000. (zitiert auf Seite 73, 74 und 75)

https://dx.doi.org/10.24355/dbbs.084-201801311013

https://dx.doi.org/10.24355/dbbs.084-201801311013

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Braunschweig, den 01. Dezember 2017

https://dx.doi.org/10.24355/dbbs.084-201801311013