FernUniversität in Hagen
Fakultät für Mathematik und Informatik
Lehrgebiet Programmiersysteme
Prof. Dr. Friedrich Steimann

Mutantengenerierung durch Type Constraints

Abschlussarbeit im Studiengang
Bachelor of Science im Fach Informatik

vorgelegt von

Robert Bär
Heusenstammer Str. 31
63179 Obertshausen
Matrikelnummer: 7120702
robsmail@gmx.de

betreut durch

Prof. Dr. Friedrich Steimann
Dipl.-Inform. Andreas Thies

August 2010

Danksagung

Herzlich danken möchte ich meinem Betreuer Andreas Thies für seine ausführlichen Hilfestellungen und seine wertvollen Tipps. Mein Dank gilt weiter meinem Großvater Dieter Bär für die anregenden Diskussionen zum Thema und seine fundierten Korrekturvorschläge. Nicht zuletzt danke ich meiner Verlobten Anke Spahn für ihre unendliche Geduld und Unterstützung während der zurückliegenden Zeit.

Inhaltsverzeichnis

1 Einleitung ... 1

1.1 Motivation ... 1
1.2 Verwandte Arbeiten ... 2
1.3 Aufbau der Arbeit ... 2

2 Grundlagen ... 3

2.1 Automatisierte Tests ... 3
2.2 Mutation Testing ... 3
2.3 Mutationsoperatoren ... 5
2.4 Type Constraints ... 5
2.4.1 Einführung ... 5
2.4.2 Aufbau der Constraints ... 7
2.4.3 Lösung eines Constraintsystems ... 9
2.4.4 Syntaktische und semantische Constraint-Regeln ... 10

3 Mutantengenerierung durch Type Constraints ... 11

3.1 Prinzip ... 11
3.2 Invertierung von Type Constraints ... 13
3.2.1 Semantische Constraint-Regeln ... 13
3.2.2 Syntaktische Constraint-Regeln ... 18
3.2.3 Zusammenfassung ... 22
3.3 Prüfung der Mutanten ... 23

4 Implementierung ... 25

4.1 Übersicht ... 25
4.2 Generierung der Type Constraints ... 25
4.3 Lösung des Constraintsystems ... 26
4.4 Prüfung der Mutanten ... 26
4.5 Prüfung der Tests ... 26
4.6 Beschreibung der Oberfläche ... 27
4.7 Übersicht der Pakete und Klassen ... 28

5 Auswertungen ... 29

5.1 Generierte Mutanten ... 30
5.2 Ungültige Mutanten (Compilerfehler) ... 32
5.3 Gültige Mutanten ... 35
5.4 Abgelehnte Mutanten ... 36

6 Diskussion ... 40

6.1 Compilerfehler ... 40
6.1.1 Fehlerhafte Erzeugung der Constraint-Variablen ... 40
6.1.2 Sichtbarkeitsproblem ... 40
6.2 Abgelehnte Mutanten ... 41
6.3 Bewertung des Ansatzes ... 42

7 Erweiterung 43
7.1 Prinzip 43
7.2 Auswertungen 44
7.3 Diskussion 46

8 Schlussbetrachtungen 47
8.1 Zusammenfassung 47
8.2 Fazit 47

A. Inhalt der beiliegenden CD 49

B. Installation des Plugin 50

Abbildungsverzeichnis 51

Literaturverzeichnis 53

Erklärung 55

1 Einleitung

1.1 Motivation

Der Programmquelltext von Softwaresystemen im aktiven Einsatz unterliegt ständigen Veränderungen. Neue Anforderungen an das Produkt müssen integriert und identifizierte Fehler korrigiert werden. Doch wie lässt sich gewährleisten, dass durch die Veränderungen nicht neue Fehlzustände entstehen und die Korrektur eines Fehlers nicht zur Entstehung von neuen Problemen führt?

Neben begleitenden Maßnahmen der Qualitätssicherung (wie beispielsweise "*Reviews*", siehe [Henrich 2001] Abs. 8.4.1) gehört das Testen zu den wichtigsten Mitteln, um Programmierfehler zu finden. Hierbei bieten automatisierte Tests den Vorteil, dass sie ohne ständiges Eingreifen einer Person arbeiten und am Ende ihres Durchlaufs über gefundene Fehlzustände berichten können.

Automatisierte Tests können allerdings nur die Fehler finden, die in ihren Routinen berücksichtigt und explizit geprüft werden. Somit stellt sich folgende Frage: "Wie lässt sich herausfinden, ob die automatisierten Tests **neu auftretende** Fehlzustände wirklich erkennen würden?". Oder allgemein formuliert: "Wie kann man ihre Qualität, bezogen auf den Abdeckungsgrad, beurteilen?".

Ein Ansatz zur Beantwortung dieser Fragen liegt im *Mutation Testing*. Er basiert auf der Idee, absichtlich Änderungen an einem Programm vorzunehmen und anschließend zu prüfen, ob der entstandene sogenannte Quelltext-Mutant von den Tests als fehlerhaft erkannt wird. So naheliegend diese Idee scheint, sie scheitert meist an folgenden Problemen: Zum einen soll durch die Veränderung nur kompilierbarer Quellcode erzeugt werden. Dies kann zwar durch einen Compiler geprüft werden, hat dann allerdings sehr hohe Laufzeitkosten zur Folge. Zum anderen gilt es herauszufinden, ob sich die Änderung wirklich als solche im Programmverhalten niederschlägt. Wird beispielsweise die Reihenfolge zweier unabhängiger Anweisungen getauscht (wie etwa `this.a = a; this.b = b;` in einem Konstruktor), so hat dies keine Auswirkungen auf das Programmverhalten und kann daher auch nicht von den Tests erkannt werden. Doch wie lässt sich eine solche notwendige Verhaltensänderung sicherstellen?

Um dieses Ziel zu erreichen, sollen in der vorliegenden Arbeit Erkenntnisse aus dem Bereich der *Refactoring-Tools* herangezogen werden. Diese nehmen ebenfalls Anpassungen am Quellcode vor, allerdings ohne dabei das Verhalten des Programms zu verändern. Bestimmte *Refactoring-Tools* greifen dazu auf sogenannte *Type Constraints* zurück, welche für die Verwendung von Klassen und Schnittstellen (*Interfaces*) eine Beibehaltung des Programmverhaltens garantieren. Diese Technik soll dazu genutzt werden, um den gewünschten gegenteiligen Effekt zu erzielen: Es werden Quelltext-Mutanten erzeugt, die sich anders als das Ausgangsprogramm verhalten und daher von den automatisierten Tests als fehlerhaft erkannt werden sollten.

1.2 Verwandte Arbeiten

In [Steimann & Thies 2010] wurde das beschriebene Vorgehen erfolgreich zur Erzeugung von Mutanten genutzt. Im Unterschied zur vorliegenden Arbeit greifen Steimann & Thies [2010] hingegen auf die *Accessibility Constraints* zurück. Es handelt sich dabei um *Constraints* (siehe Abschnitt 2.4), die an den Zugreifbarkeiten (bspw. `public` oder `private`) von Programmkonstrukten (wie Klassen, Methoden oder Attributen) ansetzen. Durch eine Veränderung dieser Zugreifbarkeiten lassen sich Mutanten generieren, die von den automatisierten Tests geprüft werden.

Die vorliegende Arbeit knüpft an die Erkenntnisse aus [Steimann & Thies 2010] an und überprüft, ob der Ansatz auf die *Type Constraints* übertragen werden kann.

1.3 Aufbau der Arbeit

Nach der informellen Einführung in Kapitel 1 werden im folgenden Abschnitt 2 die Grundlagen erläutert, auf denen diese Arbeit aufbaut. Hierzu zählen die automatisierten Tests, der Ansatz des *Mutation Testing* sowie die Arbeitsweise der *Type Constraints*.

In Kapitel 3 wird anschließend das theoretische Grundprinzip zur Erzeugung von Quelltext-Mutanten durch Invertierung von *Type Constraints* hergeleitet und erklärt. Abschnitt 4 beschreibt weiterführend, wie dieser Ansatz durch die Implementierung als *Plugin* für die Entwicklungsumgebung "Eclipse" praktisch umgesetzt werden kann. Im 5. Kapitel wird ausgewertet, welche Erkenntnisse sich bei der Anwendung dieses *Plugin* auf eine Reihe von *Open Source* Projekten ergeben haben. Eine Diskussion der Ergebnisse erfolgt im Kapitel 6. Das Kapitel 7 stellt eine Erweiterung des Ansatzes vor, diskutiert und vergleicht die dabei entstandenen Ergebnisse mit denen des ursprünglichen Ansatzes. In Kapitel 8 werden die Erkenntnisse zusammengefasst und ein Fazit zur Anwendbarkeit des ursprünglichen Ansatzes gezogen.

In diesem Text werden eine Reihe englischsprachiger Begriffe verwendet, die sich im jeweiligen Themenbereich etabliert haben. Um eine missverständliche oder unübliche Übersetzung zu vermeiden, werden diese unverändert übernommen und durch eine *kursive* Schreibweise kenntlich gemacht.

2 Grundlagen

2.1 Automatisierte Tests

Softwaretests werden verwendet, um die Qualität eines entsprechenden Produktes zu beurteilen. Es wird geprüft, ob die Software Fehler enthält, die die vorgegebenen Anforderungen (Spezifikationen) verletzt.

Das erste Vorgehensmodell, welches das Testen in den Erstellungsprozess einer Software integriert, ist das Wasserfallmodell (siehe [Six & Winter 2002] Abs. I.4.1). Die Testaktivitäten werden darin in einer eigenständigen Phase angesiedelt, die nach der Konzeption und Programmierung folgt. Neuere, agilere Methoden integrieren das Testen in den Gesamtablauf. So zielt beispielsweise der Ansatz des *Test-driven development* (kurz TDD [Beck 2003]) darauf ab, dass Tests vor der eigentlichen Implementierung eines Programmabschnitts konzipiert und erstellt werden sollen.

Die TDD-Methode fokussiert auf eine Automatisierung von Tests. Anstelle von Testpersonen, die manuell bestimmte Eingaben vornehmen und die Ausgaben des Programms überprüfen, übernehmen hier Softwaremodule diese Aufgabe. Im Mittelpunkt steht die Prüfung der Funktionalität einzelner Module (Klassen) oder Gruppierungen von Modulen, sodass diese Tests oft unter dem Begriff "Komponententests" (*Unit Tests*) zusammengefasst werden. Testwerkzeuge (wie *JUnit*, [JUnit 2010]) unterstützen den Entwickler bei der Generierung dieser Tests, ihrer automatisierten Ausführung und der Protokollierung der Ergebnisse. Eine Menge an Komponententests kann zu sogenannten *Testsuites* zusammengefasst werden und ermöglicht damit das automatisierte Testen aller Softwarekomponenten.

Diese Komponententests finden sicherlich nicht jeden Fehler. Sie stoßen insbesondere beim Test von komplexen Softwaresystemen an ihre Grenzen, da sich die gesamte Funktionalität nur mit erheblichem Aufwand durch die Testroutinen abbilden lässt. Als ergänzende Maßnahme der Qualitätssicherung sind Komponententests dennoch sehr nützlich: Sie können dem Entwickler wichtige Hinweise geben, ob seine Implementierung Fehler enthält, welche die Spezifikation verletzen. Durch die Ausführung der kompletten *Testsuite* kann geprüft werden, ob eine Änderung negative Auswirkungen auf das Verhalten der Software hat. Der Ansatz ermöglicht damit eine Verbesserung der Produktqualität, da Fehler direkt bei der Implementierung erkannt werden können und nicht erst in nachfolgenden Testphasen.

2.2 Mutation Testing

Der Ansatz des *Mutation Testing* (auch bekannt als *Mutation Analysis*) zielt darauf ab, in mehreren Durchläufen unterschiedliche Änderungen an einem Programm durchzuführen, um so die Qualität der automatisierten Tests (bezogen auf ihren Abdeckungsgrad) zu überprüfen [Steimann & Thies 2010]. Die Änderung wird dabei als Mutation bezeichnet, das resultierende Programm wird Mutant genannt. Eine Mutation soll das ursprüngliche Programm fehlerhaft machen, sodass es seiner Spezifikation nicht mehr in vollem Umfang gerecht wird. Falls dann bei erneuter Testausführung Fehler protokolliert werden, zeigt dies, dass die vorhandenen Tests ausreichen, um den durch die Mutation eingebauten Fehler zu finden. Falls bei den neuen Tests hingegen kein Fehlverhalten erkannt wird, kann dies auf eine mangelnde Testabdeckung hinweisen.

Damit derart nützliche Erkenntnisse gewonnen werden können, müssen Mutanten folgende Anforderungen erfüllen:

- **Gültigkeit**
  Jeder Mutant muss der Syntax der verwendeten Programmiersprache genügen und damit kompilierbaren Quelltext erzeugen. Bei der Mutationsgenerierung sollen daher möglichst nur "gültige Mutanten" erzeugt werden.
- **Äquivalenz**
  Die Mutation muss zusätzlich die Logik des Programms berücksichtigen. Ein Mutant soll (semantisch) **nicht** äquivalent zum Ausgangsprogramm sein. Das Vertauschen zweier unabhängiger Anweisungen kann beispielsweise zu einem identischen Programmverhalten führen, wie das folgende Beispiel in Abbildung 1 demonstriert. Bei der Mutationsgenerierung ist daher darauf zu achten, dass semantisch "äquivalente Mutanten" möglichst erkannt und vermieden werden.

Während die (syntaktische) Gültigkeit eines Mutanten unabhängig vom konkreten Programm verstanden werden kann, muss die (semantische) Äquivalenz genau genommen relativ zur Spezifikation des Programms bewertet werden: Änderungen des Programmverhaltens sind nur dann wirklich hilfreich, wenn sie das **spezifizierte** Verhalten des Programms verletzen.

Original Quellcode

```
1 public class A {
2    int a;
3    int b;
4    public A(int a, int b) {
5            this.a = a;
6            this.b = b;
7    }
8 }
```

Äquivalenter Mutant

```
  1 public class A {
  2    int a;
  3    int b;
  4    public A(int a, int b) {
! 5            this.b = b;
! 6            this.a = a;
  7    }
  8 }
```

Abbildung 1: Beispiel eines äquivalenten Mutanten

Abbildung 1 zeigt auf der rechten Seite einen Mutanten, bei dem die beiden Anweisungen innerhalb des Konstruktors vertauscht wurden (Zeile 5 und 6). Dies führt allerdings zu keiner Verhaltensänderung gegenüber dem Ausgangsprogramm, sodass die automatisierten Tests diesen Mutanten unmöglich aufspüren können (und auch gar nicht sollen).

Dieser Umstand muss bei der Auswertung der Testergebnisse berücksichtigt werden. Allgemein sind bei der Ausführung der automatisierten Tests nach der Generierung eines Mutanten zwei Situationen möglich:

Resultat 1: Die Tests melden Fehler (Verletzung der Spezifikation)

> In diesem Fall hat der Mutant eine Veränderung des Programmverhaltens bewirkt, die erfolgreich durch die automatisierten Tests festgestellt wurde. Im Jargon des *Mutation Testing* wird dies auch als "der Mutant wird getötet" (*mutant killed*) bezeichnet und ist ein Zeichen für einen entsprechenden Abdeckungsgrad.

Resultat 2: Die Tests melden keinen Fehler

> Im zweiten Fall wurde die Mutation nicht durch die Tests festgestellt. Dies impliziert allerdings noch nicht, dass die Testabdeckung unvollständig ist. Der Entwickler muss daraufhin den Mutanten analysieren und die Auswirkungen der Mutation abschätzen. Er muss herausfinden, ob es sich bei der Veränderung um einen potenziellen Fehler handelt und ob daher zusätzliche Testfälle zu dessen Erkennung notwendig sind.

Der Erfolg dieses Vorgehens hängt von den konkret durchgeführten Mutationen ab und im Speziellen davon, ob sie potenzielle Programmierfehler in ausreichendem Maße abbilden. Eine Klassifizierung der Mutationen wird durch die sogenannten Mutationsoperatoren ermöglicht.

2.3 Mutationsoperatoren

Ein Mutationsoperator ist eine Vorschrift, nach der ein Mutant erzeugt werden kann. Er legt fest, welche Änderungen am Quelltext des Ausgangsprogramms durchgeführt werden sollen. Ein einfacher Mutationsoperator ist beispielsweise einer, der das Auftreten eines "+" durch ein "-" ersetzt.

Laut [Jia & Harman 2010] wurden die ersten Mutationsoperatoren für die Programmiersprache Fortran konzipiert. Sie waren zunächst auf die Gegebenheiten der klassischen Programmiersprachen (wie z.B. auch Pascal oder C) ausgelegt und beispielsweise in der Lage, Ausdrücke durch das Ersetzen von arithmetischen Operatoren zu verändern. In [Ma et al. 2002] und [Offut et al. 2006] werden weitere Operatoren vorgestellt, die zusätzlich auch objektorientierte Sprachmittel berücksichtigen. Hierbei werden insbesondere die Paradigmen Datenkapselung, Vererbung und Polymorphismus genannt. Die vorgestellten Mutationsoperatoren sollen häufige Fehlerursachen abbilden, die bei der Entwicklung in objektorientierten Umgebungen, speziell in der Programmiersprache Java, relevant sind.

Der Mutationsoperator *Overriding method deletion* (IOD) entfernt beispielsweise eine überschreibende Methode aus einer Klasse. Seine Implementierung sucht entsprechend nach allen überschreibenden Methoden des aktuellen Projekts und löscht jeweils eine. Dies hat zur Folge, dass an ihren aufrufenden Stellen die ursprünglich überschriebene Methode aus der Elternklasse gebunden wird. Ein Test, der diese Mutation erkennen soll, muss somit sicherstellen, dass der Methodenaufruf an die korrekte Methode gebunden wird.

Es gibt weitere Mutationsoperatoren zur Veränderung der Sichtbarkeiten von Methoden und Attributen (*Access modifier change*) oder auch zur Erzeugung von Fehlern im Zusammenhang mit überladenen Methoden, Konstruktoren oder Parametern.

Implementierungen existieren für Java und andere Programmiersprachen und ermöglichen die Durchführung dieser Mutationsoperatoren an konkretem Quellcode. In [Ma et al. 2005] wird "MuJava" vorgestellt, ein Mutationssystem für Java, das die Operatoren aus [Ma et al. 2002] beherrscht und entsprechende Mutanten erzeugen kann.

Forschungen haben allerdings gezeigt, dass diese Art der Verwendung von Mutationsoperatoren zu einer hohen Anzahl an semantisch äquivalenten Mutanten (siehe Abschnitt 2.2) führt. In [Ma et al. 2009] wurde anhand einiger Testprojekte ausgewertet, dass mehr als 86% der mit "MuJava" generierten Mutanten ein identisches Programmverhalten gegenüber dem jeweiligen Ausgangsprogramm besitzen.

2.4 Type Constraints

2.4.1 Einführung

Das System der *Type Constraints* wurde von Palsberg & Schwartzbach [1993] eingeführt. Es stellt ein Mittel dar, um die Verwendung von statischen Typen innerhalb eines Programms zu beschreiben. Mit ihm lassen sich Bedingungen (*Constraints*) über die verwendeten Typen formulieren, die im Rahmen einer Typänderung eingehalten werden müssen, um die

Sprachspezifikation (bei Java die *Java Language Specification* [Gosling et al. 2005]) zu erfüllen. Ein *Type Constraint* besteht aus zwei *Constraint*-Variablen, die zueinander in Verhältnis gesetzt werden. Die *Constraint*-Variablen können unter anderem mit den Typen von Attributen oder Methodenparametern belegt werden, um Bedingungen über ihre jeweiligen Typen auszudrücken [Tip et al. 2003]. Dies soll anhand eines Beispiels gezeigt werden, bevor im Abschnitt 2.4.2 auf den genauen Aufbau der *Type Constraints* eingegangen wird.

```
 1 public class A {}
 2
 3 public class B extends A {}
 4
 5 public class C extends B {}
 6
 7 public class Start {
 8    A val1;
 9    B val2;
10    Start() {
11            val1 = val2;
12    }
13 }
```

Abbildung 2: Quellcode-Beispiel

In Zeile 11 des obigen Quellcode-Beispiels wird der Wert des Attributs `val2` dem Attribut `val1` zugewiesen. In Java wird gemäß der Sprachspezifikation [Gosling et al. 2005] gefordert, dass der statische Typ der rechten Seite einer Zuweisung Subtyp oder vom identischen Typ wie der der linken Seite der Zuweisung zu sein hat. Dies ist im konkreten Beispiel erfüllt: `val2` ist vom statischen Typ `B`, der ein Subtyp des statischen Typs von `val1` ist.

Diese Zusicherung kann als *Type Constraint* in folgender Form formuliert werden:

$$[\text{val2}] \leq [\text{val1}]$$

Die eckigen Klammernpaare symbolisieren dabei die Typen der eingeschlossenen Variablen. Der Operator ≤ schreibt vor, dass beide Typen entweder überein zu stimmen haben oder der linke ein Subtyp des rechten sein muss.

Sollen Änderungen an deklarierten Typen vorgenommen werden, so lässt sich deren Gültigkeit mithilfe der *Type Constraints* prüfen. Angenommen, der deklarierte Typ des Attributs `val1` soll durch den Typ `B` ersetzt werden. Das oben genannte *Constraint* ist konform mit dieser Änderung, da die deklarierten Typen von `val1` und `val2` anschließend identisch sind. Eine Ersetzung der Deklaration von `val1` durch den Typ `C` wäre hingegen nicht konform, da sie die geforderte Bedingung verletzen würde.

Das Ersetzen von deklarierten Typen kann in der Entwicklungsumgebung "Eclipse" (www.eclipse.org) mithilfe des *Refactoring-Tools* "Generalize Declared Type" durchgeführt werden. Dieses verwendet *Type Constraints*, um mögliche Typersetzungen einer Deklaration zu berechnen und dem Entwickler vorzuschlagen. In [Tip et al. 2003] wird gezeigt, wie *Type Constraints* im *Refactoring-Tool* "Extract Interface" zum Einsatz kommen, welches den Zugriff auf eine Klasse über ein neu eingeführtes *Interface* ermöglicht.

Durch die Verwendung von *Type Constraints* stellen beide *Refactoring-Tools* sicher, dass die vorgenommenen Typersetzungen konform mit der Java-Sprachspezifikation [Gosling et al. 2005] sind und somit bei einer Typänderung keine Compilerfehler im Programm entstehen.

Um dies zu erreichen, wurden für Java eine Reihe von *Type Constraint*-Regeln definiert, die dessen Sprachspezifikation abbilden (siehe Abschnitt 2.4.2). Gemäß diesen Regeln werden für alle Sprachkonstrukte eines Quelltextes (wie beispielsweise Zuweisungen und Methodenaufrufe) die entsprechenden *Constraints* generiert. Eine Typänderung muss alle generierten *Type Constraints* erfüllen, damit sie erfolgreich durchgeführt werden kann.

2.4.2 Aufbau der Constraints

Ein *Type Constraint* besteht aus zwei oder mehr Variablen, die die Typen von Deklarationselementen oder Ausdrücken abbilden. In Abbildung 3 werden alle Elemente aufgeführt, die als Variable in einem *Type Constraint* dargestellt werden können. Die Notation von *Type Constraints* wird in Abbildung 4 gezeigt. In Abbildung 5 wird eine Funktion dargestellt, die von den in Abbildung 6 aufgeführten Regeln zur Generierung von *Type Constraints* referenziert wird. Die Abbildungen sind aus [Tip et al. 2003], [Tip et al. 2007] und [Kegel 2007] übernommen.

Variable	**Bedeutung**
[*E*]	Typ eines Ausdrucks oder Deklarationselementes *E*
[*M*]	Rückgabetyp der Methode *M*
[*F*]	Deklarierter Typ des Attributes *F*
Decl(M)	Typ, in dem die Methode *M* deklariert ist
Decl(F)	Typ, in dem das Attribut *F* deklariert ist
Param(M, i)	Typ des *i*-ten formalen Parameters der Methode *M*

Abbildung 3: Variablen der *Type Constraints*

Die Elemente der linken Spalte aus Abbildung 3 werden als *Constraint*-Variablen bezeichnet.

Notation	**Bedeutung**
$T_1 \leq T_2$	T_1 ist Subtyp von T_2 oder identisch mit T_2
$T_1 < T_2$	T_1 ist Subtyp von T_2
$T_1 = T_2$	T_1 ist identisch mit T_2

Abbildung 4: Notation der *Type Constraints*

Funktion	**Bedeutung**
RootDefs(M)	Enthält alle Methoden, die *M* überschreiben bzw. implementieren

Abbildung 5: Funktion zur Methodenbestimmung

Programmkonstrukt	***Constraint***	
Zuweisung $E_1 = E_2$	$[E_2] \leq [E_1]$	(1)
Methodenaufruf $E.m(E_1, \ldots, E_n)$ der Methode M	$[E.m(E_1,\ldots, E_n)] = [M]$	(2)
	$[E_i] \leq [Param(M,i)]$	(3)
	$[E] \leq Decl(M_1)$ oder … oder $[E] \leq Decl(M_k)$ mit $RootDefs(M) = \{M_1, \ldots, M_k\}$	(4)
Attributzugriff $E.f$ auf Attribut F	$[E.f] = [F]$	(5)
	$[E] \leq Decl(F)$	(6)
`return` E in Methode M	$[E] \leq [M]$	(7)
M_2 überschreibt M_1, M_2 != M_1	$[Param(M_2, \text{i})] = [Param(M_1, \text{i})]$	(8)
	$[M_2] \leq [M_1]$	(9)
	$Decl(M_2) < Decl(M_1)$	(10)
F_2 versteckt F_1	$Decl(F_2) < Decl(F_1)$	(11)
Konstruktoraufruf `new` $C(E_1, \ldots, E_n)$ an Konstruktor M	$[E_i] \leq [Param(M,i)]$	(12)
Cast $(C)E$	$[E] \leq [(C)E]$ oder $[(C)E] \leq [E]$ falls $[E]$ eine Klasse ist	(13)
Für jeden Typ T	$T \leq$ `java.lang.Object`	(14)
	`[null]` $\leq T$	(15)
Implizite Verwendung von `this` in Methode M	`[this]` $= Decl(M)$	(16)
Ausdruck `new` $C(E_1, \ldots, E_n)$	`[new` $C(E_1, \ldots, E_n)] = C$	(17)
Deklaration einer Methode M (innerhalb des Typs T)	$Decl(M) = T$	(18)
Deklaration eines Attributs F (innerhalb des Typs T)	$Decl(F) = T$	(19)

Abbildung 6: Regeln zur Generierung von *Type Constraints* in Java

Anhand eines kurzen Beispiels soll die Generierung der *Type Constraints* gezeigt werden. Im nachfolgenden Quellcode ist die Klasse `B` ein Subtyp der Klasse `A`.

```
 1 public class Start {
 2    public Start() {
 3            B b = new B();
 4            A a = b;
 5            m(a);
 6    }
 7    public void m(A obj) {
 8            // ...
 9    }
10 }
```

Abbildung 7: Quelltext als Basis zum Generieren von *Type Constraints*

Die gesamte Struktur des Quelltextes (genauer: der konkreten Kompilierungseinheit) kann in einen abstrakten Syntaxbaum (kurz AST für "*Abstract syntax tree*" [Kuhn & Thomann 2006]) überführt werden. Dies ist eine Baumstruktur, die alle syntaktischen Elemente der Kompilierungseinheit entsprechend ihrem hierarchischen Aufbau enthält. So existiert

beispielsweise ein Knoten für jede Klasse unter dem sich wiederum Knoten für die deklarierten Attribute und Methoden befinden. Zum Aufbau der *Constraints* werden diese Knoten durchsucht und die passenden *Constraint*-Regeln angewendet (weitere Informationen zum Erstellungsprozess können in [Kegel 2007], Absatz 5.1.2 nachgelesen werden).

Mithilfe der Regeln aus Abbildung 6 werden auf diese Weise die folgenden *Type Constraints* für die Klasse aus Abbildung 7 erzeugt:

Zeile	***Constraint***	
2	*Decl*(Start.Start()) = Start	(18)
3	[new B()] ≤ [b]	(1)
3	[new B()] = B	(17)
4	[b] ≤ [a]	(1)
5	[a] ≤ [*Param*(Start:m(),0)]	(3)
7	*Decl*(Start.m()) = Start	(18)

Abbildung 8: Generierte *Type Constraints*

Diese *Constraints* stellen die geforderten Bedingungen dar, die die Klasse Start erfüllen muss, um bei einer Änderung der deklarierten Typen syntaktisch und semantisch korrekt zu bleiben. Bei der Betrachtung dieser Gesamtmenge an Bedingungen, die *Constraintsystem* genannt wird, fällt folgendes auf: Für die Variable `a` wird aufgrund der Zuweisung in Zeile 4 gefordert, dass sie identisch oder ein Elterntyp von `b` (also vom Typ `B`) sein muss. Der Aufruf in Zeile 5 führt weiter dazu, dass `a` identisch oder ein Subtyp des ersten Parameters der Methode `m()` zu sein hat (also vom Typ `A`). Als Konsequenz ergibt sich damit, dass der deklarierte Typ von `a` in `B` geändert werden darf, ohne dass dabei die *Constraints* verletzt werden. Bei einem Blick auf den Quellcode-Abschnitt in Abbildung 7 lässt sich dies nachvollziehen: Wird die Variable `a` in Zeile 4 mit dem Typ `B` deklariert, so ergibt sich ein kompilierbarer Programmcode, der einen identischen Programmablauf gewährleistet.

2.4.3 Lösung eines Constraintsystems

Nachdem im vorherigen Abschnitt beschrieben wurde, wie ein *Constraintsystem* aufgebaut wird, soll hier darauf eingegangen werden, wie es gelöst werden kann. Ein *Constraintsystem* zu lösen bedeutet, eine Belegung der *Constraint*-Variablen (Abbildung 3) mit Typen zu finden, mit der alle *Constraints* konform sind.

Eine gültige Lösung stellt insbesondere die Ausgangsbelegung dar, also die Typen der *Constraint*-Variablen zum Zeitpunkt der *Constraint*-Erstellung. Die generierten *Type Constraints* aus Abbildung 8 sind demnach für die deklarierten Typen aus dem Quelltext in Abbildung 7 erfüllt. Bei der Änderung des deklarierten Typs eines Programmelementes lässt sich mithilfe des *Constraintsystems* prüfen, ob diese neue Variablenbelegung gültig ist. Zu diesem Zweck müssen alle *Constraints* geprüft werden, die die geänderte *Constraint*-Variable referenzieren. Falls diese *Constraints* erfüllt sind, stellt die Typänderung eine Lösung des *Constraintsystems* dar und überführt den Programmcode in einen gültigen Mutanten. Dieser ist somit konform mit der Sprachspezifikation und besitzt ein identisches Programmverhalten gegenüber dem Ursprungsprogramm.

Neben der Prüfung von angestrebten Typänderungen kann das *Constraintsystem* auch genutzt werden, um mögliche Typänderungen vorzuschlagen. In [Tip et al. 2003] wird beispielsweise

gezeigt, wie ein System aus *Type Constraints* genutzt werden kann, um für das *Refactoring-Tool* "*Extract Interface*" ein möglichst allgemeines *Interface* zu berechnen.

2.4.4 Syntaktische und semantische Constraint-Regeln

An dieser Stelle soll eine Klassifizierung der *Type Constraint*-Regeln vorgenommen werden, die in späteren Erläuterungen von Bedeutung ist und auf Überlegungen aus [Tip et al. 2003] zurückgeht. Dort wird beschrieben, dass die vorgestellten *Type Constraint*-Regeln konzipiert wurden, um die Semantik der Sprache Java abzubilden. Sie sollen damit eine Typprüfung ermöglichen, die sonst von Java Compilern durchgeführt wird. Ein paar der Regeln gehen allerdings einen Schritt weiter und werden dazu benötigt, die Beibehaltung des Programmverhaltens sicherzustellen.

Um diesen Umstand berücksichtigen zu können, sollen die *Type Constraint*-Regeln entsprechend ihrer Absicht klassifiziert werden:

Definition 1: **Semantische *Constraint*-Regeln**
Als semantisch wird jede *Constraint-Regel* bezeichnet, die eine Veränderung des Programmverhaltens verhindern soll. Ist nach einer Typersetzung ein auf Basis dieser Regel generiertes *Constraint* nicht erfüllt, äußert sich dies im veränderten Quelltext nicht durch einen Compilerfehler.

Zu dieser Klasse zählen die *Constraint*-Regeln (8), (10) und (11).

Definition 2: **Syntaktische *Constraint*-Regeln**
Als syntaktisch wird jede *Constraint*-Regel bezeichnet, das das Auftreten eines Compilerfehlers verhindern soll. Eine Mutation, die eines auf Basis dieser Regel generierten *Constraint* nicht erfüllt, kann entsprechend zu Compilerfehlern führen.

Zu dieser Klasse zählen die übrigen *Constraint*-Regeln (1)-(7), (9) und (12)-(19).

Die Klassifikation soll beispielhaft für jeweils eine *Constraint*-Regel gezeigt werden: Die Verletzung eines *Constraint*, das auf Basis von Regel (1) generiert wurde, könnte dadurch erreicht werden, dass ein Typ durch einen echten Obertyp ersetzt wird. Für den Quellcode aus Abbildung 7 entspräche dies einer Zuweisung `B b = new A()`. Dies ist gemäß der Java Sprachspezifikation [Gosling et al. 2005] nicht möglich und führt zu einem Compilerfehler. Dagegen stellt ein auf Basis von Regel (8) generiertes *Constraint* nur sicher, dass die Parametertypen einer überschreibenden und der überschriebenen Methode identisch sind. Eine Verletzung dieses *Constraint* (beispielsweise durch die Veränderung des Parametertyps) führt dazu, dass die Methode die andere nicht mehr überschreibt und nicht zwangsweise zu einem Compilerfehler.

3 Mutantengenerierung durch Type Constraints

3.1 Prinzip

Im Folgenden wird ein Ansatz vorgestellt, der zur Überprüfung der Abdeckung von automatisierten Tests genutzt werden kann. Er basiert auf der Idee des *Mutation Testing* (siehe Abschnitt 2.2) und dient zur Erzeugung von Quelltext-Mutanten, mit denen die Abdeckung der automatisierten Tests überprüft werden kann. Der Fokus liegt dabei auf einer hohen Qualität der erzeugten Mutanten.

In Abschnitt 2.2 wurde diskutiert, welche Kriterien zur Beurteilung der Qualität von Mutanten herangezogen werden können: Zum einen kann geprüft werden, ob der erzeugte Mutant gemäß der Sprachspezifikation gültig ist und damit erfolgreich kompiliert werden kann. Zum anderen sollte beachtet werden, ob durch die Mutation ein verändertes Programmverhalten erreicht wird. Das Ziel des im Folgenden vorgestellten Ansatzes ist es, ausschließlich Mutanten zu generieren die beide Kriterien erfüllen.

Zu diesem Zweck wird auf die *Type Constraints* zurückgegriffen, die normalerweise zur Überprüfung von Typersetzungen verwendet werden (siehe Abschnitt 2.4.1). Mit ihrer Hilfe soll zum einen sichergestellt werden, dass nur gültige (kompilierbare) Mutanten erzeugt werden. Dies wird dadurch erreicht, dass die *Type Constraint*-Regeln die Sprachspezifikation von Java abbilden. Zum anderen bieten sie Potenzial, um eine Veränderung des Programmverhaltens zu erreichen. Dies lässt sich insbesondere anhand der in Abschnitt 2.4.4 vorgestellten semantischen *Constraint*-Regeln verdeutlichen. Eine solche Regel zielt darauf ab, das Programmverhalten bei einer Typänderung zu erhalten. Kehrt man eine semantische Regel hingegen um, so lässt sich damit genau der gegenteilige Effekt erzielen: Es wird sichergestellt, dass sich das Programmverhalten ändert.

Dieser Grundgedanke soll an folgendem Beispielprogramm gezeigt werden:

```
 1 public class A {
 2    public void m(int i) {
 3            // ...
 4    }
 5 }
 6
 7 public class B extends A {
 8    public void m(int i) {
 9            // ...
10    }
11 }
12
13 public class Start {
14    public static void main(String[] args) {
15            B b = new B();
16            b.m(10);
17    }
18 }
```

Abbildung 9: Quellcode – Beispiel einer überschriebenen Methode

Das Beispiel zeigt die Klasse `A` und die von ihr abgeleitete Klasse `B`. Die Methode `B.m()` überschreibt die gleichnamige Methode aus Klasse `A`, da sie eine identische Signatur besitzen. Für diesen Quellcode würde unter anderem das *Type Constraint*

[Param(B.m(), 0)] = [Param(A.m(), 0)]

generiert, das eine Typgleichheit der Parameter beider Methoden fordert (siehe Regel (8) aus Abschnitt 2.4.2). Dieses *Constraint* würde beispielsweise eine Typänderung des Parameters in `B.m()` von `int` auf `Object` verhindern, da es sonst zu einem Umbinden kommen würde: Der Aufruf in Zeile 16 würde nach dieser Änderung an die Methode aus Klasse `A` gebunden.

Soll eine solche Umbindung – die einen gesuchten Mutanten darstellt – absichtlich herbeigeführt werden, so lässt sich dies ebenso durch das Invertieren des *Constraint* erreichen.

Definition 3: **Invertiertes (*Type*) *Constraint***
Ein *Constraint* c' heißt Inversion des *Constraint* c, wenn c' für jede Variablenbelegung genau dann erfüllt ist, wenn c nicht erfüllt ist. Die Inversion eines *Constraint* wird invertiertes *Constraint* genannt.

Gemäß dieser Definition lautet das oben genannte *Constraint* in invertierter Form

[Param(B.m(), 0)] ≠ [Param(A.m(), 0)]

Es wird somit gefordert, dass die Parametertypen beider Methoden `m()` unterschiedlich zu sein haben und anschließend nach einer Lösung des gesamten *Constraintsystems* gesucht (zu dem auch die weiteren (unverändert bleibenden) *Constraints* gehören, die hier nicht noch einmal aufgeführt werden sollen; siehe Abschnitt 2.4.2). Eine mögliche Lösung ist im konkreten Fall das beschriebene Ersetzen des Parametertypen von `int` durch `Object`. Dabei kommt es zu keinen Compilerfehlern, da die weiteren generierten *Constraints* für diese Typersetzung erfüllt sind.

Zusammengefasst basiert die Idee dieses Ansatzes darauf, eine Lösung für das *Constraintsystem* zu finden, in der ein bestimmtes *Constraint* absichtlich verletzt ist.

Jeder Lösung ist folgendes gemein:

1. Sie erfüllt alle nicht invertierten *Constraints* und stellt damit sicher, dass aus einem typkorrekten Programm ein typkorrektes Programm wird.
2. Sie erfüllt zusätzlich das invertierte *Constraint* und stellt damit sicher, dass eine Änderung des Bindungsverhaltens stattfindet.

In der bisherigen Ausführung wurde die zu erreichende Änderung des Programmverhaltens mit einer Änderung des Bindungsverhaltens in Verbindung gebracht. Dieser Zusammenhang wird in [Steimann & Thies 2010] eingeführt und basiert auf der Idee, dass durch eine Änderung der Bindung ein abweichender Programmablauf erreicht wird. Wird wie im obigen Beispiel ein Aufruf an eine andere Methode gebunden, so kommen dadurch Anweisungen aus anderen Stellen des Programmquelltextes zur Ausführung. Ob die Anweisungen in der neu gebundenen Methode allerdings wirklich schädliche Wirkungen auf das Verhalten des Gesamtprogramms haben, muss vom Entwickler geprüft werden. Auch in dieser Arbeit wird das Ziel verfolgt, relevante Mutanten gemäß der Definition aus [Steimann & Thies 2010] zu erzeugen.

Definition 4: **Relevanter Mutant**
Ein Mutant wird als relevant bezeichnet, wenn er kompilierbar ist und zusätzlich ein abweichendes Bindungsverhalten gegenüber dem Ausgangsprogramm besitzt.

Der Entwickler muss entscheiden, ob ein relevanter Mutant ein identisches (oder wenigstens mit der Spezifikation vereinbares) Programmverhalten gegenüber dem Ausgangsprogramm besitzt oder ob er zur Weiterentwicklung der automatisierten Tests anregt.

Im nächsten Schritt soll untersucht werden, welche der in Abschnitt 2.4 vorgestellten Regeln *Type Constraints* generieren, die durch eine Invertierung zu einem relevanten Mutanten führen können. Es wird zu diesem Zweck geprüft, wie das jeweils entstandene *Constraint-system* gelöst werden kann.

3.2 Invertierung von Type Constraints

Die vorgestellten *Type Constraint*-Regeln aus Abschnitt 2.4.2 sind entweder syntaktischer oder semantischer Natur (siehe Klassifizierung in Abschnitt 2.4.4). Während erstere die Kompilierbarkeit des Quelltextes sicherstellen, sorgen letztere für eine Erhaltung des Programmverhaltens.

Wie im vorhergehenden Abschnitt beschrieben, können einzelne – auf Basis von semantischen Regeln generierte – *Constraints* invertiert werden, um ein abweichendes Programmverhalten zu erreichen. Dieses Vorgehen ist allerdings nicht auf die semantisch-orientierten *Constraints* beschränkt. Die Verletzung eines syntaktisch-orientierten – also auf Basis einer syntaktischen Regel generierten – *Constraint* muss nicht zwangsweise zu einem Compilerfehler führen. Unter bestimmten Voraussetzungen können durch sie ebenfalls relevante Mutanten generiert werden.

Im folgenden Abschnitt werden die semantischen *Type Constraint*-Regeln behandelt. Es wird gezeigt, bei welchen *Type Constraints* eine Invertierung zu einem relevanten Mutanten führen kann. Der darauf folgende Abschnitt erläutert, welche syntaktischen *Constraint*-Regeln in welchen Fällen ebenfalls zur Generierung von relevanten Mutanten genutzt werden können.

3.2.1 Semantische Constraint-Regeln

Die semantischen *Type Constraint*-Regeln aus Abschnitt 2.4.2 sollen für eine Erhaltung des Programmverhaltens sorgen und lassen sich nach folgenden Schwerpunkten gruppieren:

- Überschreiben von Methoden (*Method overriding*)
- Verstecken von Attributen (*Variable hiding*)

3.2.1.1 Überschreiben von Methoden

Eine Klasse erbt automatisch alle zugreifbaren Methoden seiner Elternklasse. Das Überschreiben von Methoden ermöglicht es der Subklasse, eine abweichende Implementierung einer geerbten Methode zu realisieren. Dazu muss die Subklasse eine Methode M' mit demselben Namen und derselben Signatur wie die (überschriebene) Methode M der Oberklasse enthalten. Der Rückgabetyp von M' muss entweder identisch oder ein Subtyp des Rückgabetyps von M sein.

Die folgenden *Constraint*-Regeln beschreiben den generellen Sachverhalt des Überschreibens von Methoden:

$$\frac{M_2 \text{ überschreibt } M_1,\ M_2 \mathrel{!}= M_1}{[Param(M_2, i)] = [Param(M_1, i)] \quad (8)}$$

$$[M_2] \leq [M_1] \quad (9)$$

$$Decl(M_2) < Decl(M_1) \quad (10)$$

Abbildung 10: *Type Constraint*-Regeln – Überschreiben von Methoden

Regel (8) fordert, dass die Typen korrespondierender Parameter beider Methoden M_1 und M_2 identisch sein müssen. Das *Constraint* (9) drückt aus, dass der Rückgabetyp der überschreibenden Methode M_2 der gleiche oder ein Subtyp desjenigen der Methode M_1 sein muss. Das *Constraint* (10) definiert die Hierarchie zwischen den Klassen, in denen die Methoden implementiert sind: Die Methode M_1 muss einer (echten) Oberklasse der Klasse von M_2 angehören.

Die Regeln (8) und (10) stellen die semantische Korrektheit sicher. Regel (9) ist hingegen notwendig, um das Programm kompilierbar zu halten. Dies wird im weiteren Verlauf für die jeweils generierten *Constraints* gezeigt.

Die Erstellung der *Constraints* soll an folgendem Quellcode verdeutlicht werden:

```
 1 public class A {
 2    int val;
 3    A(int v) {
 4            val = v;
 5    }
 6    void setVal(A a) {
 7            val = a.val + 1;
 8    }
 9 }
10
11 public class B extends A {
12    B(int v) {
13            super(v);
14    }
15    void setVal(A a) {
16            val = a.val;
17    }
18 }
```

Abbildung 11: Quellcode – Überschreiben von Methoden

Die Methode `setVal()` in der Klasse `B` überschreibt die gleichnamige Methode der Elternklasse `A`. Für dieses Beispiel konkretisieren sich die obigen Regeln (8), (9) und (10) zu folgenden *Constraints*:

	Zeile	***Constraint***	**Verw. Regel**
1	15	[Param(B.setVal(), 0)] = [Param(A.setVal(), 0)]	(8)
2	15	[B.setVal()] ≤ [A.setVal()]	(9)
3	15	Decl(B.setVal()) < Decl(A.setVal())	(10)

Abbildung 12: *Constraints* zum Überschreiben von Methoden im betrachteten Beispiel

Das *Constraint* 1 lässt sich zu folgender Ungleichung invertieren:

[Param(B.setVal(), 0)] ≠ [Param(A.setVal(), 0)]

Dies bedeutet, dass der Typ des Parameters `a` in der Methode `B.setVal()` von dem Parameter in der Methode `A.setVal()` abweichen muss. Erreichen lässt sich dies entweder durch eine Veränderung des Parametertyps der überschriebenen oder der überschreibenden Methode. Offen ist hierbei, welchen neuen Typ der Parameter haben soll. Losgelöst vom konkreten Quelltextbeispiel ist theoretisch eine Ersetzung durch

- eine seiner Oberklasse (möglicherweise bis hin zu `java.lang.Object`),
- eine seiner Kindklassen,
- eine seiner implementierten Schnittstellen (*Interfaces*) oder
- jeden beliebigen anderen Typ

denkbar.

Welche Ersetzung in einem konkreten Fall möglich ist, lässt sich durch die weiteren *Constraints* prüfen. Nur dann, wenn alle weiteren *Constraints* erfüllt sind, kann die Ersetzung durchgeführt werden. Das Ergebnis ist ein Mutant, der kompilierbar ist aber auch ein abweichendes Bindungsverhalten besitzt.

Eine mögliche Lösung des *Constraintsystems* stellt die Typersetzung des Parameters aus `B.setVal()` von `A` zu `B` dar:

Original Quellcode

```
11 public class B extends A {
12    B(int v) {
13            super(v);
14    }
15    void setVal(A a) {
16            val = a.val + 1;
17    }
18 }
```

Relevanter Mutant

```
  11 public class B extends A {
  12    B(int v) {
  13            super(v);
  14    }
! 15    void setVal(B a) {
  16            val = a.val + 1;
  17    }
  18 }
```

Abbildung 13: Relevanter Mutant zum Quellcode aus Abbildung 11

Durch diese Änderung wird ein relevanter Mutant erzeugt. Der Quellcodeabschnitt in Abbildung 14 zeigt das durch die Typersetzung erreichte abweichende Programmverhalten. Die Anweisung in Zeile 6 prüft den Wert des Attributes von `b.val` und erwartet den Wert 1, da dieser durch den Aufruf von `setVal()` in Zeile 5 gesetzt wurde. Bei der Ausführung des JUnit-Tests mit dem ursprünglichen Code wird kein Fehler festgestellt, `b.val` hat den Wert 1. Die Ausführung auf dem Mutanten führt hingegen zu einem Fehler, da `b.val` mit dem Wert 2 belegt ist. Der Grund dafür ist, dass in Zeile 5 nicht (wie ursprünglich) die Methode in Klasse `B` aufgerufen wird, sondern die Methode in Klasse `A`.

```
1 public class GeneralTest extends junit.framework.TestCase {
2   public static void test() {
3           A a = new A(1);
4           B b = new B(0);
5           b.setVal(a);
6           assertEquals(1, b.val);
7   }
8 }
```

Abbildung 14: JUnit-Test zum Zeigen des abweichenden Programmverhaltens

Neben der vorgestellten Mutation sind auch die weiteren angesprochenen Ersetzungen (insbesondere durch eine Kindklasse oder ein implementiertes *Interface*) als gültige Lösung des *Constraintsystems* denkbar.

Diese Lösungen besitzen Ähnlichkeiten zu einem Mutationsoperator, der in [Ma et al. 2002] vorgestellt wird: Der Operator PPD (*Parameter variable declaration with child class type*) führt Änderungen an den Typen von Parametern durch, indem er sie durch den Typ ihrer Elternklasse ersetzt (siehe Beispiel in Abbildung 15).

Original Quellcode	**PPD Mutant**
`boolean equals (B o) {…}`	`boolean equals (A o) {…}`

Abbildung 15: Beispiel eines PPD Mutanten

Durch das Invertieren eines *Constraint* sind folglich Mutanten entstanden, die sich einem bereits aus der Literatur bekannten Mutationsoperatoren zuordnen lassen. Der Vorteil des Vorgehens per *Constraint*-Invertierung gegenüber einer naiven Implementierung des PPD liegt allerdings darin, dass nur die Typersetzungen berechnet werden, die zu relevanten Mutanten führen. Die Anwendung von PPD auf jedem Methodenparameter eines Programms führt zu einer hohen Anzahl an Mutanten, die nicht kompiliert werden können (vgl. Ausführung in Abschnitt 2.3). Weiterhin ließe sich bei den übrigen – erfolgreich kompilierbaren – PPD-Mutanten nicht feststellen, ob sie eine Änderung am Bindungs- oder gar am Programmverhalten bewirken würden.

Das zweite *Constraint* aus Abbildung 12 ließe sich beispielsweise invertieren, indem der Rückgabetyp der Methode `setVal()` in der Klasse `B` so geändert würde, dass er nicht mehr identisch oder Subtyp desjenigen der Methode aus Klasse `A` wäre. Eine solche Änderung ist allerdings nicht zulässig und führt zu einem Compilerfehler gemäß §8.4.5 der *Java Language Specification* [Gosling et al. 2005]. Das *Constraint* stellt damit die syntaktische Korrektheit sicher und kann daher nicht zur Generierung eines relevanten Mutanten genutzt werden.

Constraint 3 aus Abbildung 12 besagt, dass die überschreibende Methode (`setVal()` aus Klasse `B`) in einer Subklasse der überschriebenen Methode (`setVal()` aus Klasse `A`) enthalten sein muss. Dieses *Constraint* ließe sich beispielsweise invertieren, indem die überschreibende Methode `setVal()` aus der Klasse `B` gelöscht würde. Dadurch wäre die geforderte Bedingung (überschreibende Methode in Subklasse von überschriebener Methode) nicht mehr erfüllt und es würde zu einer Bindungsänderung aller Aufrufe der Methode `B.setVal()` kommen. Die *Type Constraints* sind allerdings nicht in der Lage, um derartige

Veränderungen an Deklarationen abzubilden. Sie beschränken sich auf Typersetzungen, da *Constraint*-Variablen nur mit Typen von Deklarationselementen oder Ausdrücken belegt werden können (siehe Abschnitt 2.4.2). Es lässt sich somit nicht entscheiden, ob durch das Entfernen einer überschreibenden Methode ein kompilierbarer Mutant erzeugbar ist.

3.2.1.2 Verstecken von Attributen

Ähnlich wie das Überschreiben von Methoden kann das Verstecken von Attributen genutzt werden, um eine abweichende Implementierung in einer Subklasse zu realisieren. Zu diesem Zweck wird in einer Subklasse ein gleichnamiges Attribut aus der Elternklasse deklariert, wodurch das ursprüngliche Attribut "versteckt" wird. Die folgende *Constraint*-Regel bildet diesen Sachverhalt ab:

$$\frac{F_2 \text{ versteckt } F_1}{Decl(F_2) < Decl(F_1)} \qquad (11)$$

Abbildung 16: *Type Constraint*-Regel – Verstecken von Attributen

Diese Regel drückt aus, dass die Klasse mit dem Attribut F' von der Klasse mit dem Attribut F abgeleitet sein muss, damit F von F' versteckt wird. Anhand eines Quellcodebeispiels wird eine mögliche Verletzung des *Constraint* gezeigt:

```
 1 public class A {
 2    int val = 10;
 3    A(int v) {
 4            val = v;
 5    }
 6    int getVal() {
 7            return val;
 8    }
 9 }
10
11 public class B extends A {
12    int val = 20;
13    B(int v) {
14            super(v);
15    }
16    int getVal() {
17            return val;
18    }
19 }
```

Abbildung 17: Quellcode – Verstecken von Attributen

Das Attribut `val` ist in der Elternklasse `A` deklariert (siehe Zeile 2) und wird in der abgeleiteten Klasse `B` versteckt (siehe Zeile 12). Auf Basis der oben genannten Regel wird folgendes *Constraint* erzeugt:

	Zeile	*Constraint*	Verw. Regel
1	12	Decl(B.val) < Decl(A.val)	(11)

Abbildung 18: *Constraint* zum Verstecken von Attributen im betrachteten Beispiel

Das *Constraint* ließe sich invertieren, indem das Attribut aus der Klasse `B` gelöscht würde. Damit läge keine Subtyp-Beziehung mehr zwischen beiden Attributen vor, das Attribut `A.val` wäre für die abgeleitete Klasse `B` sichtbar. Wie bereits zum *Constraint* 3 in Abschnitt 3.2.1.1 diskutiert, kann auch in diesem Fall das Entfernen eines Deklarationselementes nicht durch die *Type Constraints* abgebildet werden.

3.2.2 Syntaktische Constraint-Regeln

Die im vorherigen Abschnitt vorgestellten semantischen *Constraint*-Regeln zielen darauf ab, das Programmverhalten bei Typersetzungen beizubehalten. Es liegt daher nahe, dass eine Umkehrung dieser Regeln genau den gegenteiligen Effekt hat: Das Programmverhalten wird verändert. Die syntaktischen *Constraint*-Regeln zielen hingegen darauf ab, den Quelltext kompilierbar zu halten. In der Regel wird die Umkehrung eines per syntaktischer Regel generierten *Constraint* demnach zu Quellcode führen, der sich nicht übersetzen lässt und damit einen ungültigen Mutanten (*invalid mutant*) darstellen würde. Im Folgenden wird allerdings gezeigt, dass unter bestimmten Umständen auch syntaktische Regeln zur Generierung relevanter Mutanten invertiert werden können. Die betrachteten syntaktischen *Constraint*-Regeln lassen sich folgendermaßen gliedern:

- Methodenaufrufe (*Method calls*)
- Konstruktoraufrufe (*Constructor calls*)
- Attributzugriffe (*Field access*)

3.2.2.1 Methodenaufrufe

In [Kegel 2007] werden die folgenden *Type Constraints* für Methodenaufrufe genannt:

Methodenaufruf $E.m(E_1, \ldots, E_n)$ der Methode M	
$[E.m(E_1,\ldots, E_n)] = [M]$	(2)
$[E_i] \leq [Param(M,i)]$	(3)
$[E] \leq Decl(M_1)$ oder … oder $[E] \leq Decl(M_k)$ mit $RootDefs(M) = \{M_1, \ldots, M_k\}$[1]	(4)

Abbildung 19: *Type Constraint*-Regeln – Methodenaufruf

Regel (2) definiert, dass der Rückgabetyp von M identisch mit dem zurückgegebenen Typ des Methodenaufrufs ist. Des Weiteren fordert Regel (3), dass die übergebenen Parametertypen identisch oder Subtyp der deklarierten Parametertypen der Methode M zu sein haben. Abschließend wird durch die *Constraint*-Regel (4) ausgedrückt, dass der Ausdruck E

[1] In der Menge RootDefs(M) befinden sich alle Methoden, die M überschreiben

identisch oder ein Subtyp der Klasse zu sein hat, in der die Methode M (oder eine sie überschreibende Methode) deklariert ist.

Im Kontext von überladenen Methoden lassen sich diese *Constraint*-Regeln nutzen, um Mutanten zu erzeugen, die ein abweichendes Programmverhalten bewirken. Dies lässt sich an folgendem Quellcode zeigen:

```
 1 public class A {
 2    int val = 1;
 3 }
 4
 5 public class B extends A {
 6    int val = 2;
 7 }
 8
 9 public class Setter {
10    int val;
11    Setter() {
12            B x = new B();
13            setVal(x);
14    }
15    void setVal(A a) {
16            this.val = a.val;
17    }
18    void setVal(B b) {
19            this.val = b.val;
20    }
21 }
```

Abbildung 20: Quellcode – Überladene Methoden

Im Konstruktor der Klasse Setter wird in Zeile 13 die Methode `setVal()` aufgerufen. Dieser Aufruf mündet in den folgenden *Constraints*, die auf Basis der Regeln aus Abbildung 19 erstellt wurden:

	Zeile	***Constraint***	**Verw. Regel**
1	13	[this.setVal(x)] = [Setter.setVal(B b)]	(2)
2	13	[x] ≤ [B b]	(3)
3	13	[Setter] ≤ Decl(Setter.setVal(B b))	(4)

Abbildung 21: *Constraints* zum Methodenaufruf im betrachteten Beispiel

Das erste *Constraint* lässt sich nicht zur Generierung eines relevanten Mutanten nutzen. Der Rückgabetyp des Aufrufs muss gemäß §8.4.5 [Gosling et al. 2005] mit der aufgerufenen Methode übereinstimmen. Eine Verletzung dieses *Constraint* führt somit in jedem Fall zu einem Compilerfehler. Ebenso kann das *Constraint* 3 nicht genutzt werden, um durch Invertierung einen Mutanten zu erzeugen. Gemäß §15.12.2.1 der *Java Language Specification* [Gosling et al. 2005] muss die Methode an der Aufrufstelle zugreifbar sein.

Das *Constraint* 2 lässt sich in diesem Beispiel hingegen zu folgendem Ausdruck invertieren:

[x] > [B b]

Das übergebene Parameterobjekt `x` müsste in dieser Umkehrung ein Supertyp des formalen Parameters `b` sein. Dies würde dazu führen, dass der Aufruf nicht mehr an die Methode `setVal(B b)` gebunden ist. Es wäre allerdings möglich, dass dieser Aufruf an eine überladene Methode gebunden wird.

Voraussetzung für diese Umbindung ist, dass die überladene Methode mindestens einen Parameter besitzt, dessen Typ echt allgemeiner ist als derjenige der ursprünglichen Methode. Nur in diesem Fall kann durch die Invertierung des *Constraint* ein kompilierbarer Mutant erzeugt werden.

Im Beispiel aus Abbildung 20 ist diese Voraussetzung erfüllt: Neben der in Zeile 13 aufgerufenen Methode `setVal(B b)` gibt es eine überladene Methode `setVal(A a)`. Eine Lösung des *Constraintsystems* stellt entsprechend die Ersetzung des deklarierten Typs von `x` dar, und zwar seines ursprünglichen Typs `B` durch `A`. Auf diese Weise wird der Methodenaufruf an die Methode in Zeile 15 gebunden. Der resultierende Mutant wird in der folgenden Abbildung dargestellt.

Relevanter Mutant

```
   9 public class Setter {
  10    int val;
  11    Setter() {
! 12            A x = new B();
  13            setVal(x);
  14    }
  15    void setVal(A a) {
  16            this.val = a.val;
  17    }
  18    void setVal(B b){
  19            this.val = b.val;
  20    }
  21 }
```

Abbildung 22: Relevanter Mutant des Beispiel-Quellcodes

Diese Mutation stellt eine Instanz des Mutationsoperators PMD (*Member variable declaration with parent class type* [Ma et al. 2002]) dar. Dieser deklariert eine Variable mit dem Typ ihrer Oberklasse.

3.2.2.2 Konstruktoraufrufe

Für einen Konstruktoraufruf ist folgende *Constraint*-Regel definiert:

$$\frac{\begin{array}{c}\text{Konstruktoraufruf} \\ \texttt{new } C(E_1, \ldots, E_n) \\ \text{an Konstruktor } M\end{array}}{[E_i] \leq [Param(M,i)]} \qquad (12)$$

Abbildung 23: *Type Constraint*-Regel – Konstruktoraufruf

Diese Regel entspricht der Regel (3) aus dem vorherigen Abschnitt über Methodenaufrufe. Die gewonnenen Erkenntnisse über Methoden lassen sich direkt auf Konstruktoren übertragen.

Vorausgesetzt wird hier, dass der aufgerufene Konstruktor Parameter besitzt und demnach nicht der Standardkonstruktor ist. Zusätzlich muss ein weiterer Konstruktor vorhanden sein, dessen Signatur sich in einem Punkt vom aufgerufenen Konstruktor unterscheidet: Mindestens ein Parametertyp ist echt allgemeiner als der entsprechende Parametertyp des ursprünglichen Konstruktors.

Folgendes Beispiel zeigt diesen Sachverhalt (Klasse `B` ist erneut abgeleitet von `A`):

```
 1 public class Constructor {
 2    public Constructor(A a) {
 3           System.out.println("Konstruktor A");
 4    }
 5    public Constructor(B b) {
 6           System.out.println("Konstruktor B");
 7    }
 8 }
 9
10 public class Test {
11    public static void main(String[] args) {
12           B b = new B();
13           new Constructor(b);
14    }
15 }
```

Abbildung 24: Quellcode – Konstruktoraufruf

In Zeile 13 findet die Instanziierung der Klasse `Constructor` statt, die zu einem Aufruf des zugehörigen Konstruktors in Zeile 5 führt. Durch ein Verallgemeinern des übergebenen Parametertypen `b` kann der Aufruf auf den anderen Konstruktor umgebunden werden (siehe Ausführungen im vorherigen Abschnitt 3.2.2.1).

3.2.2.3 Attributzugriffe

Um den korrekten Zugriff auf Attribute zu gewährleisten, werden in [Tip et al. 2003] die folgenden *Constraint*-Regeln angegeben:

$$\frac{\text{Attributzugriff } E.f \text{ auf Attribut } F}{[E.f] = [F] \qquad (5)}$$
$$[E] \leq Decl(F) \qquad (6)$$

Abbildung 25: *Type Constraint*-Regeln – Attributzugriff

Die Regel 5 definiert, dass der Typ des Attributes identisch sein muss mit dem zurückgegebenen Typen der Attributabfrage. Die *Constraint*-Regel 6 gibt an, dass die abgefragte Instanz E identisch oder Subtyp der Klasse sein muss, in der F deklariert ist.

Die *Constraint*-Regel 5 lässt sich nicht zur Generierung eines Mutanten nutzen. Sie beschreibt einen Zustand, der für jeden Attributzugriff gilt und nicht durch eine Mutation verändert werden kann. Die Regel 6 lässt sich allerdings im Rahmen von versteckten Attributen verletzen und führt zu einem kompilierbaren Quelltext, der ein abweichendes Bindungsverhalten besitzt. Dies lässt sich durch folgendes Beispielprogramm zeigen:

```
 1 public class A {
 2    int val;
 3    A() {}
 4 }
 5
 6 public class B extends A {
 7    int val = 2;
 8    B() {}
 9 }
10
11 public class Test {
12    public static void main(String[] args) {
13            B b = new B();
14            System.out.println(b.val);
15    }
16 }
```

Abbildung 26: Quellcode – Attributzugriff

Auf Basis der Regeln in Abbildung 25 lassen sich für den Attributzugriff in Zeile 14 die folgenden *Constraints* formulieren:

	Zeile	***Constraint***	**Verw. Regel**
1	14	[b.val] = [B.val]	(5)
2	14	[b] ≤ Decl(B.val)	(6)

Abbildung 27: *Constraints* zum Attributzugriff im betrachteten Beispiel

Das *Constraint* 2 lässt sich zum Ausdruck

[b] > Decl(B.val)

invertieren. Eine mögliche Lösung des entsprechenden *Constraintsystems* stellt eine Variablenbelegung dar, in der `b` ein Supertyp von `B` ist. In diesem Fall würde das versteckte Attribut aus der Elternklasse anstelle des ursprünglichen Attributes abgefragt. Im zugehörigen Mutant wäre der deklarierte Typ in der Zeile 13 durch `A` ersetzt.

Diese Mutation lässt sich erneut einem etablierten Mutationsoperator zuordnen. Der Operator PMD (*Member variable declaration with parent class type*) deklariert eine Instanz mit dem Typ ihrer Elternklasse und wurde bereits im Abschnitt 3.2.2.1 eingeführt.

3.2.3 Zusammenfassung

In den vorherigen Abschnitten wurde untersucht, welche Mutationen möglich sind, um einzelne konkrete *Constraints* zu verletzen. Die Mutationen stellen dabei mögliche Lösungen für das jeweilige *Constraintsystem* dar. Sie erfüllen in den Beispielprogrammen alle *Constraints* (also das invertierte und auch alle weiteren) und erzeugen somit einen relevanten Mutanten.

Auf diese Weise konnten mehrere *Constraint*-Regeln identifiziert werden, die in der nachfolgenden Tabelle zusammengefasst sind. Bei der Anwendung auf einem beliebigen Programmcode können die entsprechenden konkreten *Constraints* invertiert und die jeweils möglichen Mutationen (Typersetzungen) geprüft werden. Falls eine Mutation mit dem vorliegenden *Constraintsystem* konform ist (und demnach eine Lösung des *Constraintsystems* darstellt), führt sie zu einem relevanten Mutanten.

In diesem Sinne wurden in den vorherigen Abschnitten Einzelfälle herausgearbeitet, die jetzt in Form eines allgemeinen Schemas auf andere Fälle übertragen werden können. Zu diesem Schema gehört neben der identifizierten *Constraint*-Regel auch die Mutation, die zur Verletzung der jeweiligen *Constraints* führt. Die folgende Abbildung fasst diese Erkenntnisse zusammen.

Thema	Regel	Abschnitt	Mögliche Mutation
Überschriebene Methoden	(8)	3.2.1.1	Ersetzen des Parametertyps der überschreibenden Methode (Mutationsoperator PPD)
Überladene Methoden / Konstruktoren	(3)	3.2.2.1	Deklaration des der Methode übergebenen Objekts als sein Elterntyp (Mutationsoperator PMD)
	(12)	3.2.2.2	Deklaration der dem Konstruktor übergebenen Instanz als sein Elterntyp (Mutationsoperator PMD)
Versteckte Attribute	(6)	3.2.2.3	Deklaration der abgefragten Instanz als sein Elterntyp (Mutationsoperator PMD)

Abbildung 28: Übersicht der identifizierten *Constraint*-Regeln

3.3 Prüfung der Mutanten

In den vorhergehenden Abschnitten wurde der Fokus auf das Invertieren von *Constraints* gelegt und damit auf die Frage, wie bei der Generierung von Mutanten eine Verhaltens-änderung erreicht werden kann. In diesem Abschnitt steht die Frage im Mittelpunkt, ob die erzeugten Mutanten auch gültigen Quellcode besitzen und somit kompilierbar sind (siehe Motivation der "gültigen Mutanten" in Abschnitt 2.2).

Die Kompilierbarkeit der Mutanten soll dadurch sichergestellt werden, dass jede Lösung des *Constraintsystems* alle *Constraints* erfüllt, die auf Basis von syntaktischen Regeln erzeugt wurden. Aus einem typkorrekten Programm entsteht somit ein mutiertes typkorrektes Programm. Das folgende Quellcode-Beispiel zeigt jedoch, dass dies alleine nicht ausreicht, um die Kompilierbarkeit des Quellcodes sicherzustellen.

```
 1 package a;
 2 public class A {
 3    void m(int i) {
 4            // ...
 5    }
 6 }
 7
 8 package a;
 9 public class B extends A {
10    public void m(int i) {
11            // ...
12    }
13 }
14
15 package b;
16 import a.B;
17 public class Start {
18    public static void main(String[] args) {
19            B b = new B();
20            b.m(10);
21    }
22 }
```

Abbildung 29: Quellcode – Beispiel einer überschriebenen Methode

Dieser Quellcode ähnelt dem aus Abbildung 9, besitzt aber zwei wesentliche Unterschiede: Zum einen findet der Aufruf der Methode `B.m()` in Zeile 20 aus einem anderen Paket heraus statt und zum anderen ist die Methode `A.m()` nicht als *public* deklariert und daher nur innerhalb des Paketes `a` sichtbar.

Wie im Beispiel aus Abschnitt 3.1 ließe sich auch hier ein Mutant erzeugen, indem das *Constraint* invertiert wird, welches die Typgleichheit der Parameter beider Methoden `A.m()` und `B.m()` fordert.[2] Der Parameter der Methode in `B` könnte von `int` auf `Object` geändert werden um damit eine Umbindung des Aufrufs in Zeile 20 zu bewirken. Dieser Mutant ist konform mit allen *Type Constraints* und führt dennoch zu einem Compilerfehler: Der Aufruf in Zeile 20 kann nicht mehr an eine Methode gebunden werden, da `A.m()` im Paket `b` nicht sichtbar ist.

Um auch diese Fälle bei der Mutationserzeugung berücksichtigen zu können, lassen sich weitere *Constraints* zurate ziehen. Die bereits im Abschnitt 1.2 erwähnten *Accessibility Constraints* zielen darauf ab, die Zugreifbarkeit von Programmkonstrukten abzubilden. Durch sie kann sichergestellt werden, dass jeder Aufruf nach der Mutation an eine zugreifbare Methode gebunden wird (siehe [Steimann & Thies 2009]). Auf eine Kombination beider *Constraint*-Typen wird in dieser Arbeit verzichtet, da sie deren Rahmen sprengen würde. Im folgenden Kapitel 4 wird beschrieben, wie dieses Problem bei der Implementierung behandelt wird.

[2] Dieses *Constraint* wurde auf Basis von Regel (9) generiert.

4 Implementierung

4.1 Übersicht

Um den in Kapitel 3 beschriebenen Ansatz in der Praxis erproben zu können, wurde das *Plugin* "Type Constraints Tester" (kurz: TCT) für die Entwicklungsumgebung "Eclipse" entwickelt. Dieses erstellt für beliebige Java-Projekte eine Reihe von Mutanten auf Basis der *Type Constraints* und lässt sie von den JUnit-Tests prüfen. Die Implementierung wird anhand der folgenden Schwerpunkte beschrieben:

- Generierung der *Type Constraints*
- Lösung des *Constraintsystems*
- Prüfung der Mutanten
- Prüfung der Tests
- Beschreibung der Oberfläche
- Übersicht der Pakete und Klassen

4.2 Generierung der Type Constraints

In Abschnitt 2.4.2 wurde beschrieben, wie die Generierung von *Constraints* über den abstrakten Syntaxbaum einer Kompilierungseinheit (*Compilation Unit*) möglich ist: Alle Knoten des Syntaxbaumes werden verarbeitet und gemäß der *Constraint*-Regeln die passenden *Constraints* erzeugt.

Für die Verarbeitung des *Plugin* ist es notwendig, alle *Constraints* zu identifizieren, die im Weiteren invertiert werden sollen. Es handelt sich dabei um diejenigen, die auf Basis einer der in Abbildung 28 aufgeführten Regeln erstellt wurden. Ein denkbarer Anknüpfpunkt zum Identifizieren der zu invertierenden *Constraints* ließe sich direkt bei der Generierung der *Constraints* sehen. An den Stellen, an denen ein konkretes *Constraint* erzeugt wird, müsste geprüft werden, ob eine der aufgeführten *Constraint*-Regeln zum Einsatz kommt. Solche *Constraints* würden dann in eine separate Liste eingefügt, sodass sie später zur Generierung von Mutanten genutzt werden könnten. Dieser Ansatz hat aber einen entscheidenden Nachteil: Er ist sehr eng verdrahtet mit der *Constraint*-Generierung. Würden in der Zukunft weitere *Constraint*-Regeln erforscht, die eine umfassendere Prüfung des Quelltextes ermöglichten, so müsste die gesamte Identifizierung der zu invertierenden *Constraints* neu implementiert werden.

Aus diesem Grund setzt das *Plugin* einen Schritt später an: Die Generierung der *Constraints* wird von einer unabhängigen Bibliothek durchgeführt. Erst das Ergebnis, also die Auflistung der *Constraints*, wird untersucht um daraus die zu invertierenden *Constraints* herauszufiltern. In der aktuellen Version wird die Erzeugung durch eine Standardimplementierung durchgeführt, die Bestandteil von "Eclipse" ist: Die Klasse `FullConstraintCreator` aus dem Paket `org.eclipse.jdt.internal.corext.refactoring.typeconstraints` erzeugt alle vorgestellten *Type Constraints* aus Abschnitt 2.4.2.

4.3 Lösung des Constraintsystems

Das *Plugin* verarbeitet anschließend in mehreren Durchläufen jeweils ein identifiziertes *Constraint*. Dieses wird invertiert und anschließend nach einer Lösung für das *Constraintsystem* gesucht. Auf einen allgemeinen *Constraint*-Löser, der alle möglichen Typersetzungen berechnet um das *Constraintsystem* zu erfüllen, wird in dieser Implementierung verzichtet. Es sollen dagegen die Erkenntnisse aus den Untersuchungen im Abschnitt 3.2 genutzt und die dort ermittelten Typersetzungen evaluiert werden (siehe Zusammenfassung in Abschnitt 3.2.3).

Für ein konkretes invertiertes *Constraint* bedeutet dies, dass jede Typersetzung gegen das *Constraintsystem* geprüft wird. Soll beispielsweise der deklarierte Typ eines Parameters geändert werden, muss für alle *Constraints* (inklusive des invertierten) geprüft werden, ob diese nach der Typersetzung weiterhin erfüllt sind. Eine gültige Typersetzung stellt damit eine Lösung des *Constraintsystems* dar und führt zu einem Mutanten, der die gewünschte Bindungsänderung bewirkt.

Der Vorteil dieses Vorgehens liegt darin, dass jeder erzeugte Mutant durch eine einzelne Typersetzung entsteht. Ein allgemeiner *Constraint*-Löser kann hingegen auch Lösungen präsentieren, die mehrere Typersetzungen durchführen. Durch die Beschränkung auf **eine** Typersetzung ist es einfacher, die Ergebnisse mit denen von anderen Arbeiten zu vergleichen. Insbesondere durch die Zuordnung der Mutationen zu bekannten Mutationsoperatoren kann eine Einordnung der Ergebnisse dieses Ansatzes geschehen.

4.4 Prüfung der Mutanten

In Abschnitt 3.3 wurde gezeigt, dass die *Type Constraints* nicht ausreichen, um die Kompilierbarkeit eines Mutanten in jedem Fall sicherzustellen. Um dies jedoch zu erreichen, können die *Accessibility Constraints* berücksichtigt werden. Auf eine Kombination beider *Constraint*-Arten wurde in dieser Implementierung aus Komplexitätsgründen verzichtet. Dieses Thema bietet allerdings Spielraum für zukünftige Forschungen. Ein einfacherer, wenn auch laufzeitintensiverer, Weg ist die Verwendung des Java Compilers, der in dieser Implementierung Anwendung findet.

4.5 Prüfung der Tests

Jeder gültige Mutant wird im darauffolgenden Schritt durch die automatisierten Tests geprüft. Zu diesem Zweck werden die vorhandenen *Testsuites* ausgeführt und das Ergebnis protokolliert. Das Resultat gibt Aufschluss darüber, ob der Mutant erkannt wird oder nicht. Falls der Test einen Fehler ausgibt, wurde die Veränderung erfolgreich identifiziert und der Mutant "getötet". Wird hingegen kein Fehler ausgegeben, kann dies ein Hinweis auf einen fehlenden Testfall sein. Der konkrete Mutant kann vom Programmierer analysiert werden um weitergehende Erkenntnisse über die Verbesserung der Tests zu erhalten. Die Implementierung unterstützt diesen Schritt indem eine schnelle Visualisierung des Ursprungs- und Mutantenquelltextes ermöglicht wird. Diese wird im nachfolgenden Abschnitt vorgestellt.

4.6 Beschreibung der Oberfläche

Die folgende Abbildung zeigt die Oberfläche des "Type Constraints Tester", die sich als *View* in die Entwicklungsumgebung "Eclipse" einfügt.

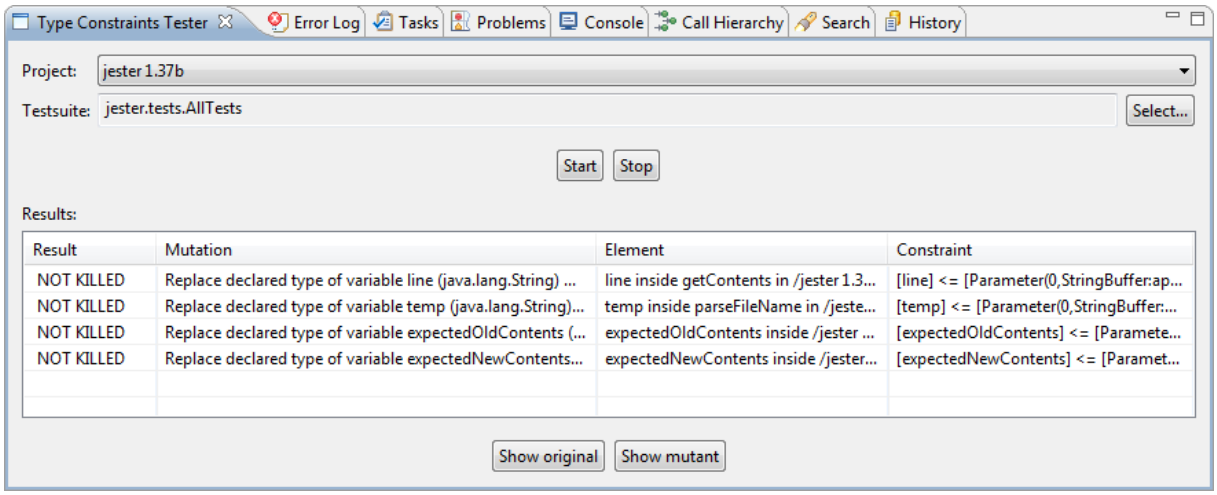

Abbildung 30: Oberfläche des "Type Constraints Tester"

Über die Dropdown-Liste "Project" kann ein Java-Projekt ausgewählt werden, das sich im aktuellen Arbeitsbereich befindet. Anschließend kann über die Schaltfläche "Select…" die *Testsuite* bestimmt werden, die für jeden Mutanten ausgeführt werden soll. Durch die Schaltflächen "Start" und "Stop" kann die Mutationsgenerierung anschließend gestartet und wieder unterbrochen werden.

Während der Verarbeitung werden die Ergebnisse der bereits geprüften Mutanten in der Liste im unteren Bereich angezeigt:

- *Result*: Hier wird der Erfolg des Mutationstests dargestellt. Folgende Inhalte sind möglich: "NOT KILLED" (Mutant wurde nicht "getötet"), "KILLED" (Mutant wurde "getötet") oder "COMPILE-ERROR" (Mutant ist nicht gültig).
- *Mutation*: Diese Spalte beschreibt die durchgeführte Mutation.
- *Element*: Dies zeigt den Ort (Paketpfad und Klassenname) des mutierten Programmelements an.
- *Constraint*: In dieser Spalte wird das *Constraint* dargestellt, das zur Erzeugung des Mutanten invertiert wurde.

Über die Schaltflächen "Show original" und "Show mutant" lassen sich die Auswirkungen der in der Ergebnisliste markierten Mutation anzeigen. Erstere öffnet die mutierte Klasse im Editor von "Eclipse" und springt direkt auf das betroffene Programmelement. Durch die Anwahl von "Show mutant" wird das Ergebnis der Mutation im Editor dargestellt.

Auf diese Weise lässt sich sehr schnell beurteilen, wie hochwertig ein Mutant ist. Der Entwickler kann somit insbesondere bei nicht erkannten Mutanten prüfen, ob ein zusätzlicher Test für den jeweiligen Fall Sinn macht.

4.7 Übersicht der Pakete und Klassen

Abschließend sollen die zentralen Klassen und Pakete aufgeführt und kurz beschrieben werden. Eine ausführliche Beschreibung der einzelnen Klassen existiert als Dokumentation in Form von *JavaDoc*. Diese ist auf der beiliegenden CD enthalten (siehe Anhang A).

Paket	Wichtige Klassen	Beschreibung
`tct.gui`	`MainView`	Anzeige und Steuerung der Oberfläche als *View*
`tct.core`	`Core`	Zentraler *Controller* für die Ablaufsteuerung
`tct.core.constraints`	`ConstraintsChecker` `InvertConstraintCreator`	Prüfung und Invertierung der *Type Constraints*

`tct.core.mutation`	`Mutation`
	`IMutantOperator`
	`PPD`
	`PMD`
		Durchführung der Mutationen und Implementierung der verwendeten Mutationsoperatoren

`tct.core.compile`	`CompileTest`	Prüfung der Mutanten per Compiler
`tct.core.junit`	`JUnitTest` `JUnitRunner`	Prüfung der Mutanten per JUnit-Test
`tct.core.result`	`ResultList`	Speicherung der Ergebnisse

Abbildung 31: Übersicht der zentralen Pakete und Klassen des *Plugin*

5 Auswertungen

In diesem Kapitel soll untersucht werden, welche Mutanten durch den Einsatz des im vorherigen Abschnitt beschriebenen *Plugin* generiert werden können. Zu diesem Zweck wurden die nachfolgenden Java-Projekte ausgewählt, deren Quelltexte frei verfügbar sind und die über eine Reihe von automatisierten JUnit-Tests verfügen. Die in diesem Kapitel vorgestellten Ergebnisse werden im darauffolgenden Kapitel 6 diskutiert und bewertet.

Projekt	Anz. Klassen	Anz. Tests	Quelle
JUnit 3.7.2	110	102	junit.org
Jester 1.37b	78	66	jester.sourceforge.net
Commons Codec 1.3	43	191	commons.apache.org/codec
Commons IO 1.4	152	396	commons.apache.org/io
Draw2d 3.4.2	358	173	eclipse.org/gef
HTML Parser 1.6	249	669	htmlparser.sourceforge.net
Jaxen 1.1.1	306	2030	jaxen.org
JHotDraw 6.01b	543	1160	jhotdraw.org
jpaul 2.5.1	153	19	jpaul.sourceforge.net

Abbildung 32: Übersicht der verwendeten *Open Source* Projekte

5.1 Generierte Mutanten

Im ersten Schritt soll gezeigt werden, wie viele Mutanten das *Plugin* für die *Open Source* Projekte erzeugen kann.

Projekt	*Type Constraints*		Mutanten		
	gesamt	**invertiert**	**erzeugt**	**gültig**	**getötet**
JUnit 3.7.2	8.696	35	0	0	0
Jester 1.37b	4.844	5	4	4	0
Commons Codec 1.3	12.649	1	0	0	0
Commons IO 1.4	23.958	50	3	2	0
Draw2d 3.4.2	47.456	206	8	6	0
HTML Parser 1.6	39.766	107	28	26	4
Jaxen 1.1.1	28.767	28	4	4	0
JHotDraw 6.01b	48.483	193	28	27	0
jpaul 2.5.1	13.338	24	0	0	0
gesamt	**227.957**	**649**	**75**	**69**	**4**
		0,3%	**11,6%**	**92,0%**	**5,8%**

Abbildung 33: Übersicht der generierten Mutanten

Zu jedem Projekt wird aufgeführt, wie viele *Type Constraints* insgesamt erzeugt werden, um die Verwendung von Typen innerhalb des entsprechenden Quellcodes zu beschreiben (siehe Abschnitt 4.2). Es wird außerdem dargestellt, wie viele *Constraints* aus dieser Gesamtmenge invertiert werden sollen. Hierzu zählen alle *Constraints*, die auf Basis einer der in Abschnitt 3.2 identifizierten Regeln erzeugt wurden. Das Verletzen dieser *Constraints* führt entsprechend zu Mutanten, die ein abweichendes Bindungsverhalten besitzen.

Für jedes invertierte *Constraint* werden alle ermittelten Typersetzungen mit Hilfe des *Constraintsystems* evaluiert (siehe Abschnitt 4.3). Daraus ergibt sich die Anzahl der erzeugten Mutanten, die in allen Fällen geringer ist als die Anzahl der invertierten *Type Constraints*. Die Differenz zwischen beiden Werten gibt entsprechend an, für wie viele invertierte *Constraints* keine gültige Typersetzung möglich ist. Für jeden erzeugten Mutanten findet eine Gültigkeitsprüfung mit dem Compiler statt (siehe Abschnitt 4.4). In der Spalte "gültig" wird entsprechend die Anzahl der Mutanten angezeigt, die diese Prüfung bestehen und somit kompilierbar sind. Abweichende Werte zwischen erzeugten und gültigen Mutanten weisen darauf hin, dass es Fälle gibt, in denen eine Mutation zu Compilerfehlern geführt hat. Konkret ist dies bei den Projekten "Commons IO", "Draw2d", "HTML Parser" und "JHotDraw" zu beobachten. Auf die Ursachen für diese Compilerfehler wird in Abschnitt 5.2 eingegangen. Die letzte Spalte gibt an, wie viele der gültigen Mutanten durch die automatisierten Tests erkannt und damit "getötet" werden.

Zu jeder Spalte wird die Summe der jeweiligen Werte angezeigt. Für die invertierten *Constraints* wird ihr Anteil an der Gesamtmenge der *Constraints* angegeben. Die Angabe für die erzeugten Mutanten bezieht sich auf ihren Anteil an den invertierten *Constraints* und damit der potenziell generierbaren Mutanten. Die Prozentzahl der gültigen Mutanten gilt im Verhältnis zu den erzeugten und diejenige der "getöteten" zu der Anzahl der gültigen Mutanten.

Es soll betrachtet werden, welche der identifizierten *Constraint*-Regeln (siehe Abbildung 28 in Abschnitt 3.2.3) zur Erzeugung von Mutanten geführt haben. Die nachfolgende Tabelle gibt Aufschluss darüber, wie sich die invertierten *Constraints* entsprechend ihrer zugrundeliegenden Regeln verteilen.

		Type Constraints	**Mutanten**		
Thema	***Constraint*-Regel**	**invertiert**	**erzeugt**	**gültig**	**getötet**
Überschriebene Methoden	(8)	547	43	38	4
Überladene Methoden / Konstruktoren	(3) / (12)	100	32	31	0
Versteckte Attribute	(6)	2	0	0	0
gesamt		**649**	**75**	**69**	**4**

Abbildung 34: Übersicht der generierten Mutanten gemäß der *Constraint*-Regeln

Abschließend soll untersucht werden, welche Laufzeitkosten bei der Generierung der Mutanten entstanden sind. Es wird dabei unterschieden, wie viel Zeit für das Kompilieren der Quelltexte, das Durchführen der automatisierten Tests und die Überprüfung der Mutationen durch das *Constraint*-System benötigt wird. Die dargestellten Werte ergaben sich bei der Ausführung auf einem Intel Core i5-750-System (2,66 GHz Taktfrequenz, 6 GB RAM) mit der Eclipse Version 3.4.2 und Windows 7.

Projekt	***Type Constraints*** **invertiert**	**Mutanten erzeugt**	**Laufzeit (in Sekunden)**		
			Kompilieren	**Test-ausführung**	***Constraint*-Prüfung**
JUnit 3.7.2	35	0	0	0	0
Jester 1.37b	5	4	1	2	0
Commons Codec 1.3	1	0	0	0	0
Commons IO 1.4	50	3	1	12	0
Draw2d 3.4.2	206	8	4	24	2
HTML Parser 1.6	107	28	18	405	3
Jaxen 1.1.1	28	4	2	12	0
JHotDraw 6.01b	193	28	43	134	6
jpaul 2.5.1	24	0	0	0	0
gesamt	**649**	**75**	**69**	**589**	**11**
			10,3%	**88,0%**	**1,7%**

Abbildung 35: Laufzeit bei Generierung der Mutanten

5.2 Ungültige Mutanten (Compilerfehler)

Wird die Anzahl der gültigen Mutanten mit der Menge an erzeugten Mutanten verglichen, fällt eine Abweichung auf: Bei sechs Mutanten – einem Anteil von 8% – wird Quelltext erzeugt, der sich nicht kompilieren lässt. Da der vorliegende Ansatz durch die Nutzung der *Type Constraints* ausschließlich Mutanten erzeugen möchte, die kompilierbar sind, wurden diese sechs ungültigen Mutanten genauer untersucht.

Im Folgenden wird ein Mutant betrachtet, der im Projekt "Draw2d" zu einem Compilerfehler führt: In der Klasse `PrecisionPoint` existiert die nachfolgend dargestellte Methode `setLocation()`, die die gleichnamige Methode aus der Elternklasse `Point` überschreibt.

```
1 /**
2  * @see org.eclipse.draw2d.geometry.Point#setLocation(Point)
3  */
4 public Point setLocation(Point pt) {
5    preciseX = pt.preciseX();
6    preciseY = pt.preciseY();
7    updateInts();
8    return this;
9 }
```

Abbildung 36: Methode setLocation() der Klasse PrecisionPoint (Projekt Draw2d)

Für den Parameter dieser Methode wird das folgende *Type Constraint* erzeugt.

	Zeile	***Constraint***	**Verw. Regel**
1	4	[Param(Point:setLocation(), 0)] = [Param(PrecisionPoint:setLocation(), 0)]	(8)

Abbildung 37: *Type Constraint* zur Methode setLocation()

Gemäß Abschnitt 3.2.1.1 stellt die Ersetzung des Parametertypen durch einen beliebigen anderen Typen eine mögliche Verletzung dieses *Constraint* dar. Im konkreten Fall wird geprüft, ob ein gültiger Mutant erzeugt werden kann, indem der Parametertyp `Point` durch den Typen `java.lang.Object` ersetzt wird. Dies sollte nicht der Fall sein, da in Zeile 5 und 6 auf je eine Methode zugegriffen wird, die nicht in der Klasse `Object` deklariert ist. Eine entsprechende Ersetzung würde also zu Compilerfehlern führen.

Die folgenden *Type Constraints* bilden diesen Sachverhalt ab:

	Zeile	***Constraint***	**Verw. Regel**
2	4	[Param(PrecisionPoint:setLocation(), 0)] = [pt]	-
3	5	[pt] ≤ Decl(Point:preciseX())	(4)
4	6	[pt] ≤ Decl(Point:preciseY())	(4)

Abbildung 38: Weitere *Type Constraints* zur Methode setLocation()

Constraint 2 gibt an, dass der Parameter als Variable `pt` deklariert wird. Die beiden weiteren *Type Constraints* stellen daraufhin sicher, dass der Parametertyp ein Subtyp oder identisch mit der Klasse `Point` zu sein hat, damit auf die Methoden `preciseX()` und `preciseY()` zugegriffen werden kann.

Obwohl diese *Constraints* augenscheinlich ausreichen, um die Ersetzung von `Point` in `Object` zu verhindern, scheitern sie an folgendem Umstand: Die *Constraint*-Variable [Param(PrecisionPoint:setLocation(), 0)], die den Parameter repräsentiert, existiert zweimal. In *Constraint* 1 und *Constraint* 2 wird dabei jeweils eine andere Variable referenziert, obwohl dieselbe Variable gemeint ist. Die Deklaration von *Constraint* 2 wird damit nicht mit dem zu ersetzenden Parametertypen aus *Constraint* 1 in Verbindung gebracht. Für den *Constraint*-Löser werden damit die *Constraints* 3 und 4 bei einer Ersetzung gar nicht in Betracht gezogen, da die Variable `pt` nicht mit dem Parameter assoziierbar ist.

Dieselbe Problemursache lässt sich bei vier weiteren ungültigen Mutanten erkennen. Der zweite nicht-kompilierbare Mutant aus dem Projekt "Draw2d" sowie jeweils einer aus "Commons IO", "HTML Parser" und "JHotDraw" lassen sich auf die fehlerhafte Erstellung der *Constraint*-Variablen zurückführen. Da diese Erstellung durch eine Standardbibliothek von "Eclipse" vorgenommen wird, scheint der Fehler in dessen Implementierung zu liegen.

Der sechste ungültige Mutant hat eine andere Ursache. Er wird innerhalb der folgenden Methode der Klasse `ScriptTag` des Projekts "HTML Parser" erzeugt.

```
 1     /**
 2      * Places the script contents into the provided buffer.
 3      * @param verbatim If <code>true</code> return as close to the
 4      * original page text as possible.
 5      * @param sb The buffer to add the script to.
 6      */
 7     protected void putChildrenInto (StringBuffer sb, boolean verbatim)
 8     {
 9         Node node;
10
11         if (null != getScriptCode())
12             sb.append(getScriptCode());
13         else
14             for (SimpleNodeIterator e=children(); e.hasMoreNodes();)
15             {
16                 node = e.nextNode ();
17                 // eliminate virtual tags
18                 if (!verbatim || !(node.getStartPosition() ==
19                         node.getEndPosition()))
20                     sb.append(node.toHtml(verbatim));
21             }
22     }
23 }
```

Abbildung 39: Methode putChildrenInto() der Klasse ScriptTag (Projekt HTML Parser)

Ähnlich wie im zuvor betrachteten Beispielcode aus Abbildung 36 überschreibt auch diese Methode eine andere aus der Elternklasse. Der ungültige Mutant wird erzeugt, indem der Parameter `StringBuffer sb` in Zeile 7 durch seine Elternklasse `AbstractStringBuilder` (aus dem Paket `java.lang`) ersetzt wird. Diese Veränderung ist konform mit allen *Type Constraints*. Das Problem entsteht diesmal durch die Eigenschaften des Typs `AbstractStringBuilder`. Dieser ist nur paketlokal sichtbar – also ausschließlich innerhalb des Pakets `java.lang` – und daher in der Klasse `ScriptTag` nicht zugreifbar. Es handelt sich bei diesem Fall demnach um eine Form des Sichtbarkeitsproblems, das in Abschnitt 3.3 beschrieben wurde.

Die nachfolgende Tabelle fasst die identifizierten Ursachen der ungültigen Mutanten zusammen.

Projekt	Mutanten		
	ungültig	Fehlerhafte *Constraint*-Variablen	Sichtbarkeitsproblem
JUnit 3.7.2	0	0	0
Jester 1.37b	0	0	0
Commons Codec 1.3	0	0	0
Commons IO 1.4	1	1	0
Draw2d 3.4.2	2	2	0
HTML Parser 1.6	2	1	1
Jaxen 1.1.1	0	0	0
JHotDraw 6.01b	1	1	0
jpaul 2.5.1	0	0	0
gesamt	**6**	**5** **83,3%**	**1** **16,7%**

Abbildung 40: Übersicht der ungültigen Mutanten

5.3 Gültige Mutanten

Nachdem im letzten Abschnitt die ungültigen – also nicht kompilierbaren – Mutanten im Vordergrund standen, soll an dieser Stelle ein genauer Blick auf die gültigen Mutanten geworfen werden. Es ist von Interesse, wie hochwertig die erzeugten Mutanten sind und ob diese tatsächlich eine Veränderung des Programmverhaltens bewirken, die von den automatisierten Tests erkennbar ist.

Nach der Auswertung der gültigen Mutanten lassen sich die folgenden beiden Fälle identifizieren, in denen keine wirkliche Veränderung des Programmverhaltens stattgefunden hat. Es handelt sich um äquivalente Mutanten, die nicht durch automatisierte Tests bemängelt werden können (siehe Kriterium der "Äquivalenz" in Abschnitt 2.2).

- **Fall 1: Überladene Methode mit identischem Ergebnis**

  Durch die Mutation wird eine andere überladene Methode aufgerufen. In dieser werden allerdings Anweisungen aufgerufen, die dasselbe Programmverhalten wie die ursprüngliche Methode zur Folge haben. Häufig zu beobachten ist dieser Effekt beim Aufruf der Methoden `StringBuffer.append()` sowie `Assert.assertEquals()`. Erstere fügt den Inhalt von Objekten an einen Zeichenpuffer an, letztere wird zur Zusicherung von Objektgleichheit verwendet.

- **Fall 2: Überschreibende leere Methode**

  Durch die Mutation wird eine überschriebene Methode aufgerufen. Sowohl die überschreibende als auch die überschriebene Methode besitzen keine Anweisungen und führen damit zum selben Programmverhalten.

Die nachfolgende Tabelle gibt Auskunft über die Häufigkeit und die Verteilung von entsprechenden Mutanten. Zusätzlich wird die Anzahl der relevanten Mutanten vermerkt, also derjenigen, die eine Änderung der Bindungsänderung bewirken.

Projekt	Mutanten				
	gültig	relevant	mit Verhaltens-änderung	ohne Verhaltens-änderung (Fall 1)	ohne Verhaltens-änderung (Fall 2)
JUnit 3.7.2	0	0	0	0	0
Jester 1.37b	4	4	0	4	0
Commons Codec 1.3	0	0	0	0	0
Commons IO 1.4	2	2	0	2	0
Draw2d 3.4.2	6	6	5	0	1
HTML Parser 1.6	26	26	4	22	0
Jaxen 1.1.1	4	4	1	3	0
JHotDraw 6.01b	27	27	24	0	3
jpaul 2.5.1	0	0	0	0	0
gesamt	**69**	**69** **100%**	**34** **49,2%**	**31** **45,0%**	**4** **5,8%**

Abbildung 41: Übersicht der gültigen Mutanten

5.4 Abgelehnte Mutanten

In den vorhergehenden Abschnitten wurden die erzeugten Mutanten genauer untersucht. Es wurde geprüft, in welchen Fällen die *Type Constraints* die Generierung von nicht-kompilierbaren Mutanten zulassen. Außerdem wurde betrachtet, ob die gültigen Mutanten zu einer Verhaltensänderung des Programms führen. An dieser Stelle soll untersucht werden, ob durch die Verwendung der *Type Constraints* auch alle möglichen Mutanten generiert werden. Es wird geprüft, ob der *Constraint*-Löser zu restriktiv arbeitet und damit Typersetzungen ablehnt, die zu kompilierbaren Mutanten führen würden.

Zu diesem Zweck soll das Ergebnis des *Constraint*-Lösers durch den Java Compiler überprüft werden. Anstatt wie ursprünglich nur die als gültig klassifizierten Mutanten auf diese Weise testen zu lassen, werden nun alle abgelehnten Typersetzungen als Mutant generiert und kompiliert. Das Auftreten von Compilerfehlern bedeutet dann, dass der *Constraint*-Löser die Typersetzung korrekterweise als ungültig klassifiziert hat. Im anderen Fall – falls also keine Compilerfehler auftreten – wurde ein gültiger Mutant fälschlicherweise als ungültig klassifiziert.

Die nachfolgende Tabelle fasst die Ergebnisse zusammen, die durch eine entsprechende Modifikation des *Plugin* entstanden sind.

Projekt	Mutanten		
	abgelehnt	ungültig	gültig
JUnit 3.7.2	24	24	0
Jester 1.37b	0	0	0
Commons Codec 1.3	1	1	0
Commons IO 1.4	6	6	0
Draw2d 3.4.2	111	109	2
HTML Parser 1.6	98	98	0
Jaxen 1.1.1	24	24	0
JHotDraw 6.01b	184	180	4
jpaul 2.5.1	0	0	0
gesamt	**448**	**442** **98,7%**	**6** **1,3%**

Abbildung 42: Übersicht der abgelehnten Mutanten

Beim Projekt "JHotDraw" gibt es vier Mutanten, die sich kompilieren lassen, obwohl sie scheinbar vom *Constraintsystem* nicht erfüllt sind. Diese Mutanten gehören alle zu ein und demselben invertierten *Constraint* (siehe Abbildung 44), das der folgenden überschreibenden Methode zugeordnet wird. Die Methode `getContent()` entstammt der Klasse `ColorContentProducer`, die von `FigureDataContentProducer` abgeleitet ist.

```
 1    /**
 2     * Produces the contents for the color
 3     *
 4     * @param context       the calling client context
 5     * @param ctxAttrName   the color attribute name
 6     *                      (FrameColor, TextColor, etc)
 7     * @param ctxAttrValue  the color
 8     * @return              The string RBG value for the color
 9     */
10    public Object getContent(ContentProducerContext context,
11                    String ctxAttrName, Object ctxAttrValue) {
12            // if we have our own color then use it
13            // otherwise use the one supplied
14            Color color = (getColor() != null) ? getColor() :
15                    (Color)ctxAttrValue;
16            String colorCode = Integer.toHexString(color.getRGB());
17            return "0x" + colorCode.substring(colorCode.length() - 6);
18    }
```

Abbildung 43: Methode getContent() aus ColorContentProducer (Projekt JHotDraw)

	Zeile	*Constraint*	Verw. Regel
1	10	[Param(FigureDataContentProducer:getContent(), 1)] = [Param(ColorContentProducer:getContent(), 1)]	(8)

Abbildung 44: *Type Constraint* zur Methode getContent()

Dieses *Constraint* wird invertiert und es ergeben sich die folgenden Typersetzungen als potenzielle Lösungen für das resultierende *Constraintsystem*: Ersetzung des Typs von Parameter `String ctxAttrName` durch

- `java.lang.Object`,
- `java.io.Serializable`,
- `java.lang.Comparable` oder
- `java.lang.CharSequence`.

Da auf den Parameter innerhalb der Methode nicht zugegriffen wird, sollte jede Typersetzung zu kompilierbarem Quelltext führen. Im konkreten Fall wird dies aber durch das folgende *Constraint* verhindert.

	Zeile	*Constraint*	Verw. Regel
2	10	[Param(ColorContentProducer:getContent(), 1)] = [Param(HTMLColorContentProducer:getContent(), 1)]	(8)

Abbildung 45: Weiteres *Type Constraint* zur Methode getContent()

Die Klasse `ColorContentProducer` wird von der Klasse `HTMLColorContentProducer` abgeleitet. Das *Constraint* 2 fordert entsprechend, dass die jeweiligen Parametertypen beider Methoden übereinstimmen müssen. Diese Bedingung wird allerdings verletzt, wenn eine der oben genannten Typersetzungen vorgenommen werden soll.

Beim Projekt "Draw2d" fallen zwei weitere Mutanten auf, die nicht zu Compilerfehlern führen. Diese beziehen sich auf die folgende Methode `applyGPrime()` der Klasse `CompoundHorizontalPlacement`.

```
 1 /**
 2  * @see org.eclipse.graph.HorizontalPlacement#applyGPrime()
 3  */
 4 void applyGPrime() {
 5    super.applyGPrime();
 6    NodeList subgraphs = ((CompoundDirectedGraph)graph).subgraphs;
 7    for (int i = 0; i < subgraphs.size(); i++) {
 8            Subgraph s = (Subgraph)subgraphs.get(i);
 9            s.x = s.left.x;
10            s.width = s.right.x + s.right.width - s.x;
11    }
12 }
```

Abbildung 46: Methode CompoundHorizontalPlacement.applyGPrime() (Projekt Draw2d)

Die Mutanten werden durch die Invertierung der folgenden *Constraints* hervorgebracht.

	Zeile	***Constraint***	**Verw. Regel**
1	9	[s] ≤ Decl(Subgraph:left)	(6)
2	10	[s] ≤ Decl(Subgraph:right)	(6)

Abbildung 47: *Type Constraints* zur Methode applyGPrime()

Um das *Constraint* 1 zu verletzen, soll die Deklaration der Variable `s` verändert werden. Der deklarierte Typ soll durch `org.eclipse.draw2d.graph.Node` ersetzt werden, dem Elterntyp des aktuell verwendeten Typs `Subgraph`. Diese Mutation führt dazu, dass in Zeile 9 das Attribut `left` aus der Klasse `Node` abgefragt wird. Verhindert wird diese Typersetzung allerdings durch das *Constraint* 2. Umgekehrt verhält es sich genauso: Der Mutant zum zweiten *Constraint* wird vom *Constraintsystem* abgewiesen, da es das erste *Constraint* nicht erfüllt.

6 Diskussion

Die im vorherigen Kapitel vorgestellten Ergebnisse werden im Folgenden analysiert und bewertet.

6.1 Compilerfehler

Durch die Verwendung der *Type Constraints* soll sichergestellt werden, dass ausschließlich kompilierbare Mutanten erzeugt werden. In Abschnitt 5.2 wurden allerdings zwei unterschiedliche Ursachen für die Entstehung von ungültigen Mutanten identifiziert, die im Folgenden diskutiert werden.

6.1.1 Fehlerhafte Erzeugung der Constraint-Variablen

In fünf von sechs Fällen scheint die Fehlerursache innerhalb des erstellten *Constraintsystems* zu liegen. In unterschiedlichen *Constraints* werden Zusicherungen über dasselbe Programmelement gemacht, wie beispielsweise über den Parameter der Methode `setLocation()` in Abbildung 37 und Abbildung 38. In bestimmten Fällen werden für dieses Programmelement allerdings unterschiedliche *Constraint*-Variablen erzeugt, sodass kein Zusammenhang mehr zwischen den unterschiedlichen *Constraints* herstellbar ist. Eine korrekte Auswertung der möglichen Typersetzungen ist dann nicht mehr möglich.

Der Grund für das mehrfache Erstellen derselben *Constraint*-Variable ist in der Implementierung der Eclipse-Bibliothek `FullConstraintCreator` zu suchen, die zum Aufbau der *Constraints* verwendet wird. Es scheint sich dabei allerdings um ein spezielles Problem zu handeln, welches nicht ohne weiteres isolierbar ist. Löst man beispielsweise die betroffenen Klassen `Point` und `PrecisionPoint` aus dem Projekt "Draw2d" heraus, so werden die *Constraint*-Variablen korrekt erstellt und der ungültige Mutant durch den *Constraint*-Löser abgewiesen. Die Ursache dieses Problems liegt demnach in den Routinen der verwendeten Eclipse-Bibliothek und nicht in der Implementierung des *Plugin*.

6.1.2 Sichtbarkeitsproblem

Eine zweite Ursache für das Entstehen von Compilerfehlern liegt in einer Einschränkung von *Type Constraints*: Sie können keine Aussagen über die Sichtbarkeit von Programmelementen treffen. Die Compilerfehler zu Abbildung 39 entstehen, da der neu eingesetzte Parametertyp (`AbstractStringBuilder`) an dieser Stelle nicht sichtbar ist, da auf ihn nur paketlokal zugegriffen werden darf. In Abschnitt 3.3 wurde ein weiterer Fall vorgestellt, der zu einem Compilerfehler führen kann: Nach dem angestrebten Umbinden eines Methodenaufrufs ist die neugebundene Methode nicht sichtbar und der Aufruf damit nicht auflösbar. Im Zusammenhang mit dem Zugriff auf Attribute sind ähnliche Fehlerquellen denkbar.

Eine Lösung dieses Sichtbarkeitsproblems kann, wie in Abschnitt 3.3 bereits erwähnt, die Verwendung der *Accessibility Constraints* aus [Steimann & Thies 2009] sein. Für jeden Zugriff auf ein deklariertes Programmelement (wie Methoden, Attribute oder Typen) werden entsprechende *Constraints* erstellt. Diese können sicherstellen, dass das referenzierte Element an der betroffenen Stelle auch sichtbar ist. Auf diese Weise können die oben genannten Compilerfehler verhindert werden.

6.2 Abgelehnte Mutanten

Die Analyse der abgelehnten Mutanten in Abschnitt 5.4 zeigt Fälle, in denen gültige – und damit kompilierbare – Mutanten nicht erzeugt werden, da die Typersetzung gegen das *Constraintsystem* verstößt. Das *Constraint* in Abbildung 45 verhindert beispielsweise im Projekt "JHotDraw", dass der Typ des Parameters `String ctxAttrName` verändert wird, obwohl dies nicht zu einem Compilerfehler führen würde. Dieser Fall tritt dann auf, wenn eine Methode von einer abgeleiteten Klasse überschrieben und diese wiederum überschrieben wird. Die folgende Abbildung verdeutlicht diese Vererbungsstruktur.

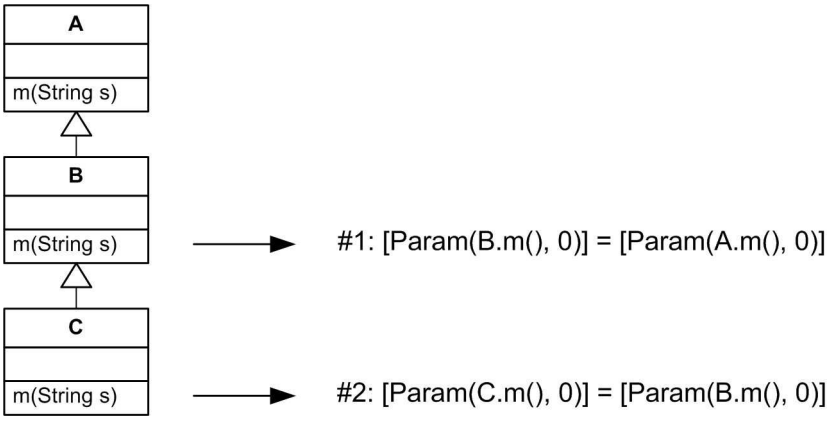

Abbildung 48: Vererbungsstruktur und erzeugte *Type Constraints*

Gemäß der *Constraint*-Regel (8) werden unter anderem *Constraints* für die Parameter aller beiden überschreibenden Methoden `B.m()` und `C.m()` erstellt. Diese stellen sicher, dass die Typen der Parameter jeweils übereinstimmen. Es wird nun das *Constraint* #1 invertiert, das die Typgleichheit der Methodenparameter von Klasse `B` und Klasse `A` sicherstellt. Verletzt wird dieses *Constraint* beispielsweise durch eine Ersetzung des Parametertypen von `B.m()` durch `java.lang.Object`. Diese Ersetzung sei im konkreten Quelltext auch problemlos möglich und führe nicht zu einem Compilerfehler. Dennoch verhindert das *Constraint* #2 der Methode aus Klasse `C` diese Ersetzung, welches eine Typgleichheit der Parameter von `B.m()` und `C.m()` fordert.

Zuletzt genanntes *Constraint* soll sicherstellen, dass die Methode `C.m()` die Methode `B.m()` überschreibt. Die Verletzung dieses *Constraint* führt damit nicht zu Compilerfehlern. Da es auf Basis einer semantischen *Constraint*-Regel erstellt wurde, soll es lediglich eine Veränderung des Programmverhaltens verhindern (siehe Definition 1).

Zwei ähnliche Fälle sind für das Projekt "Draw2d" zu beobachten. Auch hier wird die Erzeugung von Mutanten verhindert, da semantisch-orientierte *Constraints* nicht mit der Änderung konform sind. Im Unterschied zu den zuvor betrachteten Fällen, basieren diese allerdings auf der *Constraint*-Regel (6), die den Zugriff auf versteckte Attribute regelt.

Eine Möglichkeit, um diese kompilierbaren – aber dennoch abgewiesenen – Mutanten mit dem vorliegenden Ansatz erstellen zu können, ist die Verfeinerung der *Constraint*-Prüfung. So könnten die *Constraints* bei der Lösung des *Constraintsystems* ignoriert werden, die auf Basis von semantischen Regeln erstellt wurden.

6.3 Bewertung des Ansatzes

Auf dem ersten Blick erscheinen die Ergebnisse der Testläufe in Abbildung 33 ernüchternd: In nur knapp 10% der Fälle kann aus einem invertierten *Constraint* auch ein Mutant erzeugt werden. Bei mehreren Projekten konnte kein einziger Mutant generiert werden. Dies hängt damit zusammen, dass Änderungen an Typdeklarationen nur unter bestimmten Bedingungen zu gültigen Mutanten führen. Oft ist es nicht möglich, den Typ eines Parameters oder einer Variable zu verändern, da an späterer Stelle im Quelltext auf Methoden oder Attribute zugegriffen wird, die nur in der ursprünglich deklarierten Klasse verfügbar sind.

Das Ziel, nur gültige Mutanten zu erzeugen, wurde im ersten Schritt leider verfehlt. In Abschnitt 6.1 wurde allerdings diskutiert, dass die Ursachen für die aufgetretenen Compilerfehler lösbar sind. Der vorliegende Ansatz, die *Type Constraints* zur Generierung von Mutanten zu verwenden, ist demnach grundsätzlich in der Lage, die Kompilierbarkeit der Mutanten sicherzustellen. Bei der Implementierung kann dann auf die Mutantenprüfung per Compiler verzichtet werden, wodurch eine erhebliche Laufzeitersparnis erreicht wird: Laut der Auswertung in Abbildung 35 benötigt die Überprüfung per Compiler das Sechsfache der Zeit (69 Sekunden) gegenüber der Prüfung des *Constraintsystems* (11 Sekunden).

Das zweite Ziel des Ansatzes, ausschließlich relevante Mutanten (also solche, die eine Änderung des Bindungsverhalten bewirken) zu erzeugen, wurde erfüllt (vgl. relevante Mutanten in Abbildung 41). Ob Mutanten gemäß dieser Definition allerdings auch eine echte Verhaltensänderung bewirken, lässt sich zunächst grundsätzlich durch die Abbildung 33 beurteilen: In knapp 5% der Fälle wurde ein gültiger Mutant durch die automatisierten Tests "getötet". Bei diesen Fällen handelt es sich in jedem Fall um Mutationen, die eine echte – da bemerkbare – Verhaltensänderung vorgenommen haben. Ein hoher Anteil an "getöteten" Mutanten könnte also für eine hohe Qualität der Mutanten sprechen. Im Gegenzug kann der vorliegende geringe Wert allerdings auch an einer geringen Abdeckung der automatisierten Tests liegen. Klarheit bringt die Analyse in Abschnitt 5.3: Sie kommt zu dem Ergebnis, dass etwa 50% der Mutanten eine Verhaltensänderung darstellen und damit von den automatisierten Tests "getötet" werden könnten. Im Vergleich zu den in Abschnitt 2.3 genannten Auswertungen zur Mutationsgenerierung per "MuJava" [Ma et al. 2009] ist dies ein deutlich besserer Wert. Bei der Verwendung von "MuJava" sind mehr als 86% der erzeugten Mutanten äquivalent zum Ausgangsprogramm. Während der Entwickler dort also mindestens acht Mutanten untersuchen muss, um einen hochwertigen Mutanten – und damit einen potenziell fehlenden Test – zu finden, sind es beim vorliegenden Ansatz nur zwei.

7 Erweiterung

In diesem Kapitel wird eine Erweiterung des in Kapitel 3 eingeführten Ansatzes vorgestellt und diskutiert. Sie verfolgt das Ziel, eine höhere Zahl an Mutanten zu generieren und soll damit einen Vergleich mit den vorliegenden Ergebnissen ermöglichen.

7.1 Prinzip

In Abschnitt 3.2 wurden einzelne *Type Constraint*-Regeln und die durch sie erzeugbaren *Constraints* untersucht. Es wurde dabei überprüft, ob es eine Typersetzung geben kann, die das untersuchte *Constraint* verletzt und alle anderen *Constraints* erfüllt. Das Schema dieser Typersetzung sollte dann auf einem konkreten Programmcode angewendet werden, um einen Mutanten zu erzeugen. Mithilfe der Gesamtmenge an generierten *Type Constraints* ließe sich dabei beurteilen, ob eine konkrete Typersetzung den Programmcode kompilierbar halten oder zu Compilerfehlern führen würde.

Bei zwei *Constraints* sind anstelle von Typersetzungen andere Quelltext-Mutationen vorgestellt worden, die zur Verletzung des *Constraint* führen würden. Da die *Type Constraints* allerdings keine Aussage über die Gültigkeit dieser Mutationen treffen können, sind sie für den ursprünglichen Ansatz nicht nutzbar gewesen. Es handelt sich in beiden Fällen um Mutationen, die Deklarationselemente aus dem Ursprungsquellcode löschen. Im einen Fall (*Constraint* 3 aus Abbildung 12 in Abschnitt 3.2.1.1) sollen Aufrufe auf eine überschriebene Methode umgebunden werden, indem die überschreibende Methode entfernt wird. Im anderen Fall (*Constraint* 1 aus Abbildung 18 in Abschnitt 3.2.1.2) soll der Zugriff auf ein verstecktes Attribut umgebunden werden, indem das versteckende Attribut entfernt wird.

Im Rahmen einer Erweiterung des Ansatzes sollen diese Mutationen zum Einsatz kommen. Die jeweils identifizierten *Constraint*-Regeln (10) und (11) sollen zur Erzeugung von Mutanten genutzt werden, indem die durch sie generierten *Constraints* invertiert werden. Abweichend zum ursprünglichen Ansatz können die Mutationen allerdings nicht durch die weiteren *Constraints* des *Constraintsystem* überprüft werden. Es ist somit möglich, dass die erzeugten Mutanten Compilerfehler enthalten. Diese sollen – wie in der ursprünglichen Implementierung – durch den Java Compiler identifiziert werden.

Mit dieser Erweiterung soll überprüft werden, ob durch eine Abwandlung des vorliegenden Ansatzes eine größere Zahl an Mutanten generiert werden kann. Obwohl die dabei jeweils verwendeten Mutationen (Typersetzungen im ursprünglichen Ansatz und das Löschen von Programmelementen in der Erweiterung) wenig gemein haben, so erzielen sie dennoch den gleichen Effekt: Bei überschreibenden Methoden wird der Zugriff auf die ursprünglich überschriebene Methode umgebunden und bei versteckenden Attributen auf das ursprünglich versteckte Attribut.

Die angewendeten Mutationen lassen sich bekannten Mutationsoperatoren aus [Ma et al. 2002] zuordnen. Das Entfernen von überschreibenden Methoden entspricht dem Schema des Operators IOD (*Overriding method deletion*). Das Entfernen von versteckenden Attributen entspricht hingegen dem Operator IHD (*Hiding variable deletion*).

Die Erweiterung nutzt – wie auch der ursprüngliche Ansatz – das Programmverständnis der *Type Constraints* aus, um Mutanten mit einer Bindungsänderung zu erzeugen. Die Ansätze unterscheiden sich allerdings darin, welche Mutationen sie zu diesem Zweck zulassen. Die Erweiterung sieht die Durchführung von Mutationen vor, die sich nicht durch das *Constraint*-

system prüfen lassen und verletzt damit die Forderung des ursprünglichen Ansatzes, ausschließlich gültige Mutanten zu erzeugen.

7.2 Auswertungen

Nachfolgend wird gezeigt, wie viele Mutanten im Rahmen von überschriebenen Methoden generiert wurden. Die Übersicht zeigt die Ergebnisse des ursprünglichen Ansatzes sowie der Erweiterung:

Projekt	PPD-Mutanten (Ursprünglicher Ansatz)			IOD-Mutanten (Erweiterung)		
	erzeugt	**gültig**	**getötet**	**erzeugt**	**gültig**	**getötet**
JUnit 3.7.2	0	0	0	22	22	8
Jester 1.37b	0	0	0	8	8	2
Commons Codec 1.3	0	0	0	0	0	0
Commons IO 1.4	0	0	0	69	69	35
Draw2d 3.4.2	8	6	0	266	264	25
HTML Parser 1.6	6	4	4	179	175	77
Jaxen 1.1.1	1	1	0	74	74	17
JHotDraw 6.01b	28	27	0	276	274	3
jpaul 2.5.1	0	0	0	86	68	0
gesamt	**43**	**38** **88,4%**	**4** **10,5%**	**980**	**954** **97,3%**	**167** **17,5%**

Abbildung 49: Übersicht der generierten Mutanten (Überschriebene Methoden)

Weitergehend wird ausgewertet, wie viele Mutanten im Rahmen von versteckten Attributen generiert wurden:

Projekt	PMD-Mutanten (Ursprünglicher Ansatz)			IHD-Mutanten (Erweiterung)		
	erzeugt	**gültig**	**getötet**	**erzeugt**	**gültig**	**getötet**
JUnit 3.7.2	0	0	0	6	5	0
Jester 1.37b	0	0	0	0	0	0
Commons Codec 1.3	0	0	0	4	4	0
Commons IO 1.4	0	0	0	6	6	0
Draw2d 3.4.2	0	0	0	7	6	0
HTML Parser 1.6	0	0	0	0	0	0
Jaxen 1.1.1	0	0	0	30	27	0
JHotDraw 6.01b	0	0	0	66	64	0
jpaul 2.5.1	0	0	0	23	0	0
gesamt	**0**	**0**	**0**	**142**	**112**	**0**
		-	**-**		**78,9%**	**0,0%**

Abbildung 50: Übersicht der generierten Mutanten (Versteckte Attribute)

Die nachfolgende Tabelle fasst die Ergebnisse zusammen:

Ansatz	Mutanten				
	erzeugt	**gültig**	**ungültig**	**getötet**	**nicht getötet**
Ursprünglicher Ansatz	43	38	5	4	34
		88,4%	11,6%	10,5%	89,5%
Erweiterung	1122	1066	56	167	899
		95,0%	5,0%	15,7%	84,3%

Abbildung 51: Zusammenfassung der Ergebnisse

7.3 Diskussion

Die Ergebnisse der Testläufe zeigen, dass durch die Erweiterung eine deutlich höhere Zahl an Mutanten generiert werden kann. Im Gegensatz zum ursprünglichen Ansatz ist auch die Menge an "getöteten" Mutanten stark gestiegen. Diese Mutanten haben damit in jedem Fall eine Veränderung des Programmverhaltens bewirkt, der auch von den automatisierten Tests festgestellt wurde. Vergleicht man diesen Wert ("167") mit dem des ursprünglichen Ansatzes ("34", siehe Abbildung 41) wird deutlich: Durch die Anwendung der neuen Mutationen werden mindestens viermal so viele Mutanten generiert, die auch eine Veränderung des Programmverhaltens bewirken.

Diese Tatsache zeigt, dass Typersetzungen im Quelltext deutlich seltener durchführbar sind als das Entfernen von Methoden oder Attributen. Die folgenden Ursachen können für diesen Umstand verantwortlich gemacht werden:

- Die Typersetzung schlägt fehl, da nachfolgende Methodenaufrufe oder Attributzugriffe auf dieser Instanz nur für den ursprünglichen Typen aufgelöst werden können.
- Es ist keine Typersetzung möglich, da keine Typdeklaration vorhanden ist. Dies ist besonders bei überschreibenden Methoden zu beobachten, die **keine** Parameter besitzen.

Die Erweiterung hat allerdings den Nachteil, dass sie auch ungültige Mutanten erzeugt, da eine Prüfung der Mutanten durch die *Type Constraints* nicht möglich ist. Die Mutationen können hingegen dem Bereich der Sichtbarkeiten zugeordnet werden (siehe Abschnitt 3.3 von [Steimann & Thies 2010]): Eine entfernte Methode oder ein gelöschtes Attribut ist nicht mehr sichtbar an den auf sie zugreifenden Stellen. Ob dieser Zugriff an eine andere Methode bzw. ein anderes Attribut gebunden wird, lässt sich somit durch die *Accessibility Constraints* beantworten. Diese stellen eine Möglichkeit dar, um die Qualität der per Erweiterung generierten Mutanten zu verbessern.

8 Schlussbetrachtungen

8.1 Zusammenfassung

Das Ziel des vorliegenden Ansatzes ist die Erzeugung von hochwertigen Quelltext-Mutanten, die eine Überprüfung der automatisierten Tests ermöglichen. Durch die Verwendung von *Type Constraints* sollen die Mutanten zwei Anforderungen gerecht werden: Zum einen soll der veränderte Quelltext konform mit der Sprachspezifikation und damit kompilierbar sein (gültiger Mutant). Zum anderen soll durch die Mutation eine Veränderung des Programmverhaltens bewirkt werden, das auf eine abweichende Bindung von Deklarationselementen zurückgeführt wird (relevanter Mutant).

In Kapitel 3 wurde untersucht, welche *Type Constraint*-Regeln einen möglichen Ansatzpunkt zur Erstellung der Mutanten liefern. Konkrete *Constraints* sollen dabei absichtlich verletzt werden und somit eine Veränderung des ursprünglichen Programmverhaltens fordern. Eine Lösung des dadurch erzeugten *Constraintsystems* besteht aus veränderten deklarierten Typen. Diese können durch entsprechende Mutationen am Quelltext hervorgebracht werden, wie in mehreren Fällen gezeigt wurde.

Die praktische Umsetzung des Ansatzes in Form eines *Plugin* für die Entwicklungsumgebung "Eclipse" wurde in Kapitel 4 beschrieben. Dieses erprobt das vorgestellte Vorgehen, indem es Typersetzungen durchführt, die mit dem *Constraintsystem* konform sind. Die erstellten Mutanten werden anschließend von den automatisierten Tests überprüft. Es wurden bestimmte Fälle identifiziert, in denen es trotz der Verwendung von *Type Constraints* zu nicht-kompilierbarem Quelltext kommen kann. Um diese Fälle erkennen zu können, findet eine Überprüfung aller Mutanten mit dem Compiler statt.

Anschließend wurde der Ansatz an einer Reihe von *Open Source* Projekten angewendet und die Ergebnisse diskutiert. Ungültige Mutanten, die Compilerfehler verursachen, wurden untersucht und Lösungen zu ihrer Vermeidung vorgestellt. Eine Analyse der Mutanten brachte hervor, dass knapp die Hälfte eine echte Veränderung des Programmverhaltens bewirkt hat und somit durch die automatisierten Tests hätte festgestellt werden können. Abschließend konnte ermittelt werden, in welchen Fällen die *Type Constraints* eine Mutation verhindern, obwohl diese zu gültigem Quelltext führen würde.

Da sich durch die Anwendung des vorliegenden Ansatzes eine relativ geringe Zahl an Mutanten ergab, wurde in Kapitel 7 eine Erweiterung vorgestellt und diskutiert. Diese ermöglicht den Einsatz von Mutationen, die eine ähnliche Verhaltensänderung bewirken können wie die ursprünglich verwendeten Typersetzungen. Im Unterschied dazu können diese zu Compilerfehlern führen, da sie nicht durch das System der *Type Constraints* überprüfbar sind. Es wurde gezeigt, dass durch die Erweiterung eine deutlich höhere Zahl an Mutanten erzeugt werden kann und der ursprüngliche Ansatz somit in seinem Einsatzbereich beschränkt ist.

8.2 Fazit

Der vorliegende Ansatz ist grundsätzlich in der Lage, hochwertige Mutanten zu erzeugen. Er kann – durch eine Kombination mit den *Accessibility Constraints* – sicherstellen, dass ausschließlich kompilierbare Mutanten generiert werden. Bezogen auf die erzielte Änderung des Programmverhaltens ließ sich folgender Nutzen nachweisen: 50% der in Testläufen generierten Mutanten können durch automatisierte Tests erkannt werden.

Die auf Basis der *Type Constraints* erzeugbaren Mutationen sind allerdings nur in seltenen Fällen anwendbar. Im Vergleich zu alternativen Mutationen aus dem Bereich der Sichtbarkeit wird eine deutlich geringe Menge an Mutanten erzeugt. Dies ist eine Schwäche des vorliegenden Ansatzes und mindert seinen Nutzen in der Praxis. Die vorgestellte Erweiterung zeigt einen Weg, wie der Einsatzbereich vergrößert werden kann. Ob diese durch die Kombination mit den *Accessibility Constraints* zuverlässig kompilierbare Mutanten erzeugen kann, ist allerdings noch zu untersuchen.

A. Inhalt der beiliegenden CD

Verzeichnis	Inhalt
doc	Die vorliegende Abschlussarbeit im PDF-Format
src	Der Quellcode des *Plugin* "Type Constraints Tester" mitsamt den Projektdateien
javadoc	Die Dokumentation des Quellcodes im HTML-Format
jar-default	Das *Plugin* "Type Constraints Tester" als jar-Datei zum Installieren für die Entwicklungsumgebung "Eclipse"
jar-extension	Das *Plugin* der in Kapitel 7 vorgestellten Erweiterung (Bezeichnung des *Plugin*: "Type Constraints Tester (Extension)")
jar-compiletest	Das *Plugin* zum Durchführen der in Abschnitt 5.4 vorgestellten Überprüfung der abgewiesenen Mutanten per Compiler (Bezeichnung des *Plugin*: "Type Constraints Tester (Compile Test)")
projects	Die in Kapitel 5 verwendeten *Open Source* Projekte
eclipse	Eine Installation von "Eclipse" (Version 3.4.2) inklusive des *Plugin* "Type Constraints Tester" für Windows

B. Installation des Plugin

Auf der beiliegenden CD befindet sich eine Installation der Entwicklungsumgebung "Eclipse" inklusive des *Plugin* "Type Constraints Tester". Diese kann in einen Ordner auf der Festplatte kopiert und von dort gestartet werden. Um das *Plugin* einer bestehenden "Eclipse"-Installation hinzuzufügen, muss die Datei "tct_1.0.0.jar" aus dem Verzeichnis "jar-default" in den dort vorhandenen Unterordner "plugins" kopiert werden.

Nach dem anschließenden Start von "Eclipse" kann die Oberfläche des *Plugin* durch die folgenden Schritte angezeigt werden:

- Wählen Sie im Menü den Eintrag "Window", anschließend "Show View" und darunter "Other…".
- Es öffnet sich das Fenster "Show View". Wählen Sie hier die Kategorie "General" und darunter den Eintrag "Type Constraints Tester". Klicken Sie anschließend auf "OK".
- Die Oberfläche wird neben den bekannten *Views* "Tasks", "Console" und "Error Log" als Registerreiter angezeigt.

Die Bedienung der Oberfläche wird in Abschnitt 4.6 beschrieben.

Zur Installation eines der weiteren mitgelieferten *Plugins* "tct_extension_1.0.0.jar" oder "tct_compiletest_1.0.0.jar" gehen Sie bitte wie folgt vor: Löschen Sie die vorhandene *Plugin*-Datei aus dem Verzeichnis "plugins" unterhalb von "Eclipse". Kopieren Sie anschließend das gewünschte *Plugin* in dieses Verzeichnis und befolgen die oben genannten Installationsanweisungen.

Bitte beachten Sie die folgenden Hinweise:

- Das *Plugin* greift während der Verarbeitung auf die Ressourcen des ausgewählten Projektes zu. Bitte stellen Sie sicher, dass die Ressourcen in dieser Zeit nicht im Editor geöffnet sind.
- Der Installationspfad von "Eclipse" darf keine Leerzeichen enthalten.
- Bei der Mutationsgenerierung von großen Projekten (wie beispielsweise "JHotDraw") wird unter Umständen mehr Arbeitsspeicher benötigt, als standardmäßig maximal für Java-Programme bereitgestellt werden (128 MB). Starten Sie "Eclipse" mit folgenden Parametern, um den maximal nutzbaren Arbeitsspeicher auf 512 MB zu erweitern: eclipse.exe -vmargs -Xms64m -Xmx512m

Abbildungsverzeichnis

Abbildung 1: Beispiel eines äquivalenten Mutanten 4
Abbildung 2: Quellcode-Beispiel 6
Abbildung 3: Variablen der *Type Constraints* 7
Abbildung 4: Notation der *Type Constraints* 7
Abbildung 5: Funktion zur Methodenbestimmung 7
Abbildung 6: Regeln zur Generierung von *Type Constraints* in Java 8
Abbildung 7: Quelltext als Basis zum Generieren von *Type Constraints* 8
Abbildung 8: Generierte *Type Constraints* 9
Abbildung 9: Quellcode – Beispiel einer überschriebenen Methode 11
Abbildung 10: *Type Constraint*-Regeln – Überschreiben von Methoden 14
Abbildung 11: Quellcode – Überschreiben von Methoden 14
Abbildung 12: *Constraints* zum Überschreiben von Methoden im betrachteten Beispiel 14
Abbildung 13: Relevanter Mutant zum Quellcode aus Abbildung 11 15
Abbildung 14: JUnit-Test zum Zeigen des abweichenden Programmverhaltens 16
Abbildung 15: Beispiel eines PPD Mutanten 16
Abbildung 16: *Type Constraint*-Regel – Verstecken von Attributen 17
Abbildung 17: Quellcode – Verstecken von Attributen 17
Abbildung 18: *Constraint* zum Verstecken von Attributen im betrachteten Beispiel 18
Abbildung 19: *Type Constraint*-Regeln – Methodenaufruf 18
Abbildung 20: Quellcode – Überladene Methoden 19
Abbildung 21: *Constraints* zum Methodenaufruf im betrachteten Beispiel 19
Abbildung 22: Relevanter Mutant des Beispiel-Quellcodes 20
Abbildung 23: *Type Constraint*-Regel – Konstruktoraufruf 20
Abbildung 24: Quellcode – Konstruktoraufruf 21
Abbildung 25: *Type Constraint*-Regeln – Attributzugriff 21
Abbildung 26: Quellcode – Attributzugriff 22
Abbildung 27: *Constraints* zum Attributzugriff im betrachteten Beispiel 22
Abbildung 28: Übersicht der identifizierten *Constraint*-Regeln 23
Abbildung 29: Quellcode – Beispiel einer überschriebenen Methode 24
Abbildung 30: Oberfläche des "Type Constraints Tester" 27
Abbildung 31: Übersicht der zentralen Pakete und Klassen des *Plugin* 28
Abbildung 32: Übersicht der verwendeten *Open Source* Projekte 29
Abbildung 33: Übersicht der generierten Mutanten 30
Abbildung 34: Übersicht der generierten Mutanten gemäß der *Constraint*-Regeln 31
Abbildung 35: Laufzeit bei Generierung der Mutanten 32
Abbildung 36: Methode setLocation() der Klasse PrecisionPoint (Projekt Draw2d) 33
Abbildung 37: *Type Constraint* zur Methode setLocation() 33
Abbildung 38: Weitere *Type Constraints* zur Methode setLocation() 33
Abbildung 39: Methode putChildrenInto() der Klasse ScriptTag (Projekt HTML Parser) 34
Abbildung 40: Übersicht der ungültigen Mutanten 35
Abbildung 41: Übersicht der gültigen Mutanten 36
Abbildung 42: Übersicht der abgelehnten Mutanten 37

Abbildung 43: Methode getContent() aus ColorContentProducer (Projekt JHotDraw) 37
Abbildung 44: *Type Constraint* zur Methode getContent() 38
Abbildung 45: Weiteres *Type Constraint* zur Methode getContent() 38
Abbildung 46: Methode CompoundHorizontalPlacement.applyGPrime() (Projekt Draw2d) . 38
Abbildung 47: *Type Constraints* zur Methode applyGPrime() 39
Abbildung 48: Vererbungsstruktur und erzeugte *Type Constraints* 41
Abbildung 49: Übersicht der generierten Mutanten (Überschriebene Methoden) 44
Abbildung 50: Übersicht der generierten Mutanten (Versteckte Attribute) 45
Abbildung 51: Zusammenfassung der Ergebnisse 45

Literaturverzeichnis

Beck 2003

Beck, Kent: *Test Driven Development by Example*, Addison-Wesley, 2003.

Gosling et al. 2005

Gosling, James; Joy, Bill; Steele, Guy; Bracha, Gilad: *The Java Language Specification. 3. Auflage*, Addison-Wesley, 2005.
http://java.sun.com/docs/books/jls/download/langspec-3.0.pdf

Henrich 2001

Henrich, Andreas: *Kurs 01895: Management von Softwareprojekten*, FernUniversität in Hagen, 2001.

Jia & Harman 2010

Jia, Yue; Harman, Mark: *An Analysis and Survey of the Development of Mutation Testing*. King's College London, Technical Report, 2010.
http://www.dcs.kcl.ac.uk/technical-reports/papers/TR-09-06.pdf

JUnit 2010

JUnit Homepage, 2010.
http://www.junit.org (Stand: 30.07.2010)

Kegel 2007

Kegel, Hannes: *Constraint-basierte Typinferenz für Java 5*. FernUniversität in Hagen, Diplomarbeit, 2007.
http://www.fernuni-hagen.de/ps/docs/Diplomarbeit-Kegel.pdf

Kuhn & Thomann 2006

Kuhn, Thomas; Thomann, Olivier: *Abstract Syntax Tree.* Eclipse Corner Articles, 2006.
http://www.eclipse.org/articles/article.php?file=Article-JavaCodeManipulation_AST/index.html (Stand: 05.06.2010)

Ma et al. 2002

Ma, Yu-Seung ; Kwon, Yong-Rae; Offut, Jeff: *Inter-Class Mutation Operators for Java*. In: *13th International Symposium on Software Reliability Engineering*, 2002. S. 352-363.
http://www.cs.gmu.edu/~offutt/rsrch/papers/javamutop.pdf

Ma et al. 2005

Ma, Yu-Seung; Offut, Jeff; Kwon, Yong-Rae: *MuJava: An Automated Class Mutation System*. In: *Software Testing, Varification and Reliability*, 2005, S. 97-133.
http://cs.gmu.edu/~offutt//rsrch/papers/mujava.pdf

Ma et al. 2009

Ma, Yu-Seung; Kwon, Yong-Rae; Kim, Sang-Woon: *Statistical Investigation on Class Mutation Operators*. In: *ETRI Journal, vol.31, no.2*, 2009, S. 140-150. http://etrij.etri.re.kr/Cyber/servlet/GetFile?fileid=SPF-1239172363185

Offut et al. 2006

Offut, Jeff; Ma, Yu-Seung; Kwon, Yong-Rae: *The Class-Level Mutants of MuJava*. In: *Workshop on Automation of Software Test (AST 2006)*, 2006, S. 78-84. http://www.cs.gmu.edu/~offutt/rsrch/papers/ast-mujava.pdf

Palsberg & Schwartzbach 1993

Palsberg, Jens; Schwartzbach, Michael I.: *Object-Oriented Type Systems*, John Wiley & Sons, 1993.

Six & Winter 2002

Six, Hans-Werner; Winter, Mario: *Kurs 01793: Software Engineering I*, FernUniversität in Hagen, 2002.

Steimann & Thies 2009

Steimann, Friedrich; Thies, Andreas: *From Public To Private to Absent: Refactoring Java Programs under Constrained Accessibility*. In: *ECOOP 2009*, 2009, S. 419-443. http://www.fernuni-hagen.de/ps/pubs/ECOOP2009.pdf

Steimann & Thies 2010

Steimann, Friedrich; Thies, Andreas: From Behaviour Preservation to Behaviour Modification: Constraint-Based Mutant Generation. In: Proceedings of ICSE (2010), 2010, S. 425-434. http://www.fernuni-hagen.de/ps/pubs/ICSE2010.pdf

Tip et al. 2003

Tip, Frank; Kiezun, Adam; Bäumer, Dirk: *Refactoring for Generalization using Type Constraints*. In: *Object-Oriented Programming Systems, Languages and Applications* (*OOPSLA 2003*), 2003, S. 13-26. http://groups.csail.mit.edu/pag/pubs/refactoring-tip-ecoop2003.pdf

Tip et al. 2007

Tip, Frank; Fuhrer, Robert M.; Kiezun, Adam; Ernst, Michael D., Balaban, Ittai; De Sutter, Bjorn: *Refactoring Using Type Constraints*. IBM T.J. Watson Research Center, Technical Report, 2009. http://www.cs.washington.edu/homes/mernst/pubs/refactoring-type-constraints-rc24804.pdf

Erklärung

Hiermit erkläre ich, dass ich diese Abschlussarbeit selbstständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

Obertshausen, den 28. August 2010

Robert Bär