Technical Reports and Working Papers

Angewandte Datentechnik (Software Engineering)
Dept. of Electrical Engineering and Information Technology
The University of Paderborn

Mutationstest einer C++ Implementierung

Verlässliches Programmieren in C/C++

Sebastian Dröge, Carsten Rösnick, Elena Wagner,
Martin Walter

Betreuer: Dipl.-Inform. Axel Hollmann

Technical Report 2010/4 (Version 1.0, July 2010)

D-33095 Paderborn, Pohlweg 47-49 Tel.:++49-5251-603446 Fax: 603246
http://adt.upb.de email: hollmann@adt.upb.de

Gliederung

1. Einleitung

Martin Walter, Elena Wagner

Der *Mutationstest* gehört zu diversitären Testmethoden und ist keine Testtechnik im eigentlichen Sinne. Er bietet die Möglichkeit, die Leistungsfähigkeit von Testtechniken unter kontrollierten Bedingungen systematisch zu vergleichen.
Diese Methode basiert auf der Annahme, dass ein erfahrener Programmierer fast korrekte Programme erstellt, die sich nur durch bestimmte Fehlertypen von der Spezifikation unterscheiden. Diese Fehlertypen werden zur Spezifizierung von Testfällen benutzt. Dabei werden diversitäre Software-Versionen aus einer Originalversion hergeleitet, indem man kleine Fehler dieser Fehlertypen (*Mutationen*) künstlich einfügt. Jede mutierte Version (*Mutant*) enthält einen genau bekannten Fehler. Für die Originalversion der Software können vollständige Testfallmengen entsprechend unterschiedlichen Testtechniken gebildet werden. Wenn diese Testfallmengen über die Originalversion und alle entsprechend einer bestimmten Mutationstransformation gebildeten Mutanten ausgeführt werden, ist zu erwarten, dass bei einer gewissen Mutantenanzahl andere Reaktionen erzeugt werden als bei der Originalversion (Abb. 1.1) [1].

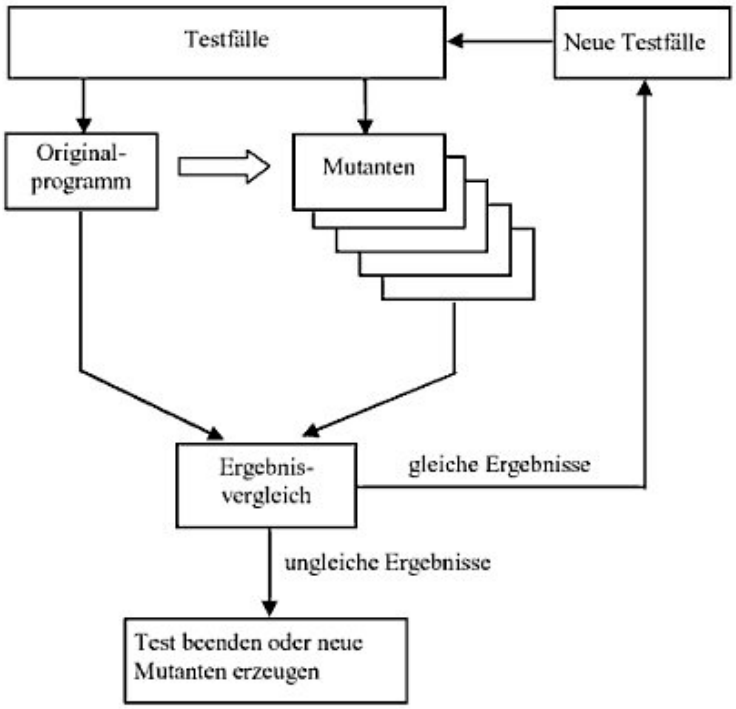

Abbildung 1.1.: Prinzip des Mutationstests [2]

Die Güte von Testfallmengen kann daran gemessen werden, inwieweit sie in der Lage sind, das Originalprogramm von seinen Mutanten zu unterscheiden. Falls Testergebnisse eines Mutanten mit dem Originalprogramm identisch sind, ist das ein Hinweis, dass die entsprechende Testfallmenge nicht effizient genug ist und durch weitere Testfälle ergänzt werden muss [2].
Im Rahmen dieser Projektarbeit wurde basierend auf einer C++ Implementierung ein umfangreicher Mutationstest durchgeführt. Dazu wurde eine Open Source C++ Software ausgewählt, zwei Testfallmengen systematisch und eine intuitiv erstellt und geeignete Mutationsoperatoren entworfen. Unter Benutzung der entworfenen Operatoren wurden Mutanten aus der Originalversion der gewählten Software generiert. Die implementierten Tests wurden mit der Originalversion und den Mutanten ausgeführt. Danach wurden Testfallmengen um Testfälle erweitert, die die nicht aufgedeckten Mutanten erkennen, und wieder zum Testen benutzt. Anschließend wurde die Güte jeder Testfallmenge analysiert und Mutationsoperatoren bewertet.

2. Software: muParser

Elena Wagner, Sebastian Dröge

Der Mutationstest wurde anhand der existierenden Open Source Software *muParser* durchgeführt, die sich unter [4] finden lässt. *muParser* ist eine C++ Bibliothek zum Parsen und Auswerten von mathematischen Ausdrücken. Ein Ausdruck kann hierzu in Bytecode umgewandelt werden um eine mehrfache Auswertung zu beschleunigen und konstante Teile des Ausdrucks nur einmal zu berechnen.

muParser bietet Parser für Fließkommazahlen, komplexe Zahlen und ganze Zahlen. Dieser Mutationstest beschränkt sich allerdings auf den Parser für Fließkommazahlen.

Weiterhin bietet *muParser* einige eingebaute Funktionen (z.B. sin() oder acosh()) an, einige eingebaute Konstanten (z.B. e oder π) und die wichtigsten mathematischen Operatoren (z.B. + für die Addition oder / für die Division). Es ist auch möglich eigene Funktionen, Konstanten oder Operatoren zu definieren und Variablen zu nutzen.

Die Gründe für die Wahl waren eine genügend hohe Komplexität, ein übersichtlicher Code und der Einsatz genügend vieler C++ Sprachkonstrukte. Abb. 2.1 zeigt ein UML-Klassendiagramm der Software, welches alle Klassen, die im Rahmen dieses Mutationstests getestet wurden, darstellt.

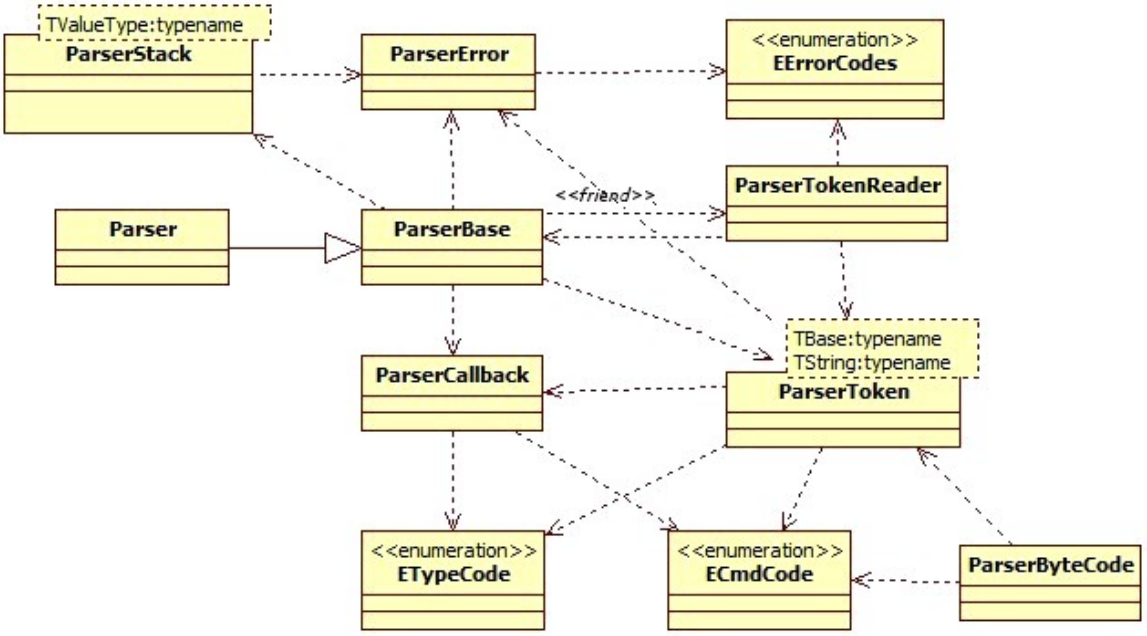

Abbildung 2.1.: UML-Klassendiagramm von muParser

3. Testfallmengen und Unit-Testing-Framework

3.1. Unit-Testing-Framework: CppUnit

Elena Wagner, Sebastian Dröge

Zur Implementierung von Testfällen wurde *CppUnit* eingesetzt. *CppUnit* ist ein Open Source C++ Framework zur Programmierung von Software-Tests für C++ nach dem Prinzip der Unit-Tests und kann unter [5] gefunden werden.

Es gibt sehr viele alternative C++ Unit Testing Frameworks, z.B. *Boost::Test*[6] oder *GoogleTest*[7]. Allerdings bieten alle Frameworks die gleich Grundfunktionalität an und im Rahmen dieses Mutationstests wurden keine der erweiterten Funktionen der Frameworks benötigt, weswegen ein beliebiges Framework gewählt wurde.

Eine Auflistung von verschiedenen C/C++ Unit Testing Frameworks mit kurzer Beschreibung zu jedem kann auf *www.opensourcetesting.org*[1] gefunden werden.

Vom Prinzip her wird *CppUnit* so verwendet, dass eine C++ Klasse als Sammlung von Tests geschrieben wird, in der verschiedene Methoden die verschiedenen Testfälle implementieren. Zur Überprüfung, ob ein Test fehlschlägt bietet *CppUnit* verschiedene Assertion Makros an, um z.B. zu testen ob eine Bedingung erfüllt ist oder ob ein Wert mit höchstens um eine bestimmte Differenz vom erwarteten Wert abweicht. Abb. 3.1 zeigt ein kleines Beispiel.

3.2. Testfallmengen

Für den Mutationstest wurden drei Testfallmengen (zwei systematisch und eine intuitiv) erstellt, deren Leistungsfähigkeit geprüft werden sollte:

- Funktionsüberdeckung
- Grenzwertanalyse
- Intuitive Erstellung

[1] http://www.opensourcetesting.org/unit_c.php

```
#include <cppunit/ui/text/TestRunner.h>
#include <cppunit/extensions/TestFactoryRegistry.h>
#include <cppunit/extensions/HelperMacros.h>

#include <muParser/muParser.h>

#define _USE_MATH_DEFINES
#include <cmath>

class ParserTest : public CppUnit::TestFixture {
public:
    CPPUNIT_TEST_SUITE (ParserTest);
    CPPUNIT_TEST (testAllocateDynamic);
    CPPUNIT_TEST (testCopy);
    //...
    CPPUNIT_TEST_SUITE_END ();

    void testAllocateDynamic () {
        mu::Parser *parser = new mu::Parser ();
        CPPUNIT_ASSERT (parser != NULL);
        delete parser;
    }

    void testCopy () {
        mu::Parser parser;
        double res, x;

        parser.SetExpr ("x = 2 + 2");
        parser.DefineVar("x", &x);

        mu::Parser parser2 = parser;
        res = parser2.Eval ();
        CPPUNIT_ASSERT_DOUBLES_EQUAL (4, res, 0.000001);
        CPPUNIT_ASSERT_DOUBLES_EQUAL (4, x, 0.000001);
    }
```

Abbildung 3.1.: Einsatz von CppUnit

3.2.1. Funktionsüberdeckung

Sebastian Dröge

Bei der *Funktionsüberdeckung* handelt es sich um ein systematisches Blackbox-Verfahren. Hierbei wird jede Funktion der zu testenden Software auf korrektes Verhalten auf korrekte und falsche Eingabewerte getestet. Das Ziel dieses Verfahren ist es, sicherzustellen, daß alle Funktionen der Software implementiert sind und sich zumindest auf den implementierten Testfällen korrekt verhalten.

Zum Anwenden dieses Verfahrens müssen zuerst alle zu testenden Funktionen identifiziert, das erwartete Verhalten dieser Funktionen bestimmt und korrekte sowie falsche Eingaben

erzeugt werden. Anschließend werden zu jeder dieser Funktionen einige Testfälle für verschiedene Eingaben erstellt.

Falls die Software zu viele Funktionen bereitstellt, um sinnvoll Testfälle für alle Funktionen zu erstellen, ist es empfehlenswert eine Auswahl zu treffen, wobei die ausgewählten Funktionen die nicht (direkt) getesteten Funktionen überdecken sollten.

Im Rahmen dieses Mutationstest wurde folgendes getestet

- alle öffentlichen C++ Methoden der `mu::Parser` Klasse
- alle eingebauten mathematischen Funktionen, Operatoren und Konstanten
- eigene mathematische Funktionen, Operatoren, Konstanten und Variablen in allen möglichen Varianten
- verschieden komplexe mathematische Ausdrücke

3.2.2. Kombination Grenzwertanalyse mit Bedingungsüberdeckung

Carsten Rösnick

Die *Grenzwertanalyse* als Blackbox-Verfahren sieht das Testen von Funktionsblöcken vor, indem die Grenzen des Wertebereichs der Eingabeparameter angenommen werden. Hierunter fallen ebenso sehr große Eingabewerte (u.a. von Interesse bei Funktionen wie `asinh`), als auch sehr kleine (kleinstmögliches ϵ, so dass $1 + \epsilon > 1$).
In Kombination zur Grenzwertanalyse findet die *Bedingungsüberdeckung* Anwendung. Die Idee dieses C2/C3-Tests [12] ist es alle Pfade eines Funktionsblocks mindestens einmal zu durchlaufen. Hierzu werden die Belegungen aller Bedingungen, sowie aller derer Unterbedingungen, in Abfragen (Verzweigungsknoten) durchlaufen. Der Sinn erklärt sich, da Prädikate in Bedingungen, die somit nun mindestens einmal wahr und unwahr werden (C2-Test), oder aber alle ihrer möglichen Werte durchlaufen (C3-Test), durchaus in einem Zweig erneut auftauchen und den Ausgang eines Zweiges entsprechend beeinflussen. Resultat ist die Diversifizität der Ergebnisse eines jeden Zweiges.
Zwar handelt es sich bei der Bedingungsüberdeckung um ein Whitebox-Verfahren, jedoch ergänzen sich beide Verfahren (zumindest im Kontext von muParser) bei näherer Betrachtung: Differente Pfade innerhalb eines Funktionsblocks können zu verschiedenen zulässigen Wertebereichen bezüglich der Eingabe führen.
Getestet wurden, entsprechend obiger Richtlinien:

- Funktionen mit mehreren Parametern und diversen assoziierten Wertebereichen (bspw. `asinh` und `atanh`), sowie dessen Löschung;

- reinitialisieren, überschreiben und löschen existenter und nicht existenter Konstanten;
- Präzedenzen eingebauter und benutzerdefinierter Operatoren;
- eigene Funktionen zur benutzerdefinierten Werteerkennung in diversen Fehlersituationen.

3.2.3. Intuitive Testfallmenge

Elena Wagner, Martin Walter

Im Gegensatz zu den vorangegangenen zwei Testfallmengen, die jede für sich eine systematische Aufstellung von Testfällen darstellt, erfolgt die Erstellung der 3ten Testfallmenge intuitiv. In diesem Fall bedeutet das, dass die Aufstellung der Testfälle kein konkretes Ziel zu verfolgen scheinen. Es ist nicht abzusehen ob z.B. der gesamte Quellcode überdeckt wird oder alle Funktionen ausgeführt werden. Dennoch sollte auch der Anspruch an die intuitive Testfallmenge sein, dass jeder Fehler im Programm gefunden wird. Um das sicher zu stellen werden verschieden Ansätze verfolgt, die in ihrer Kombination zu einer guten Fehlererkennung führen sollen.

Für den mathematischen Parser *muParser* ergeben sich verschieden Möglichkeiten intuitiv geeignete Eingaben zu finden um eine möglichst gute Fehlererkennung zu erreichen.

Als ersten Ansatz wird die Eingabe von sehr komplexen mathematischen Begriffen verfolgt. Ein Beispiel hierfür wäre z.B. folgender Ausdruck: *max(abs(4,5,6),cosh(9),sqrt(12))*. Hierbei verfolgt man die Annahme, dass mit der Komplexität der Ausdrücke auch die Wahrscheinlichkeit zunimmt, dass viele Funktionen im Programm ausgeführt werden. Um diesen Effekt zu verstärken können die Ausdrücke stark verschachtelt werden. Um hierbei Ergebnisse zu erhalten, die im Vorfeld einfach zu bestimmen sind, damit man einen Vergleich mit dem errechneten Ergebnis des mathematischen Parser unkompliziert vornehmen kann, bieten sich verschachtelte Funktionen an, die sich gegenseitig wieder aufheben. Ein Beispiel für einen derartigen Ausdruck stellt $sinh(asinh(sqrt((5)^2)))$ dar. Das zu erwartende Ergebnis ist die 5, weil sich alle äußeren Funktionen gegenseitig aufheben. Auch wenn dieses Ergebnis sehr leicht bestimmt werden kann, bedeutet er für den mathematischen Parser jedoch einen großen Rechenaufwand, weil dieser die einfachen Zusammenhänge nicht erkennen kann.

Ein anderer Ansatz stellt das Erzwingen von Fehlermeldungen und Fehlfunktionen dar. Die Motivation für diese Testfälle liegt darin, dass das Erkennen von fehlerhaften Eingaben eine gewisse *Intelligenz* vom Programm voraussetzt. Die Annahme liegt also nahe, dass durch das Erzwingen der Fehlerbehandlung ein Großteil von zuvor nicht überprüftem

Quellcode ausgeführt wird. Beispiele hierfür sind folgende:

- Das Verwenden von fehlerhaften Ausdrücken:
  - fehlerhafte arithmetische Operatoren wie *2x3* anstelle von *2*3*,
  - fehlende Klammern wie *sin3* anstelle von *sin(3)*,
  - Funktionsaufrufe mit falscher Parameterzahl.
- Das Verwenden von gelöschten Funktionen oder Konstanten
  - in *muParser* können benutzerdefinierte Konstanten oder Funktionen angelegt werden. Werden diese wieder gelöscht führt das Verwenden dieser Konstrukte zu Systemfehlern.
- Das Ausloten von Grenzfällen
  - durch das Überschreiten des zulässigen Genauigkeitsbereich können Funktionen wie *floor(4,499999999999999)* zu unerwartenden Ergebnissen führen
- Das Überschreiben von Konstanten
  - aufgrund der Möglichkeit in *muParser* eigene Konstanten und Funktionen zu definieren, lassen sich Fehler provozieren, indem man die globalen Konstanten wie *Pi* oder *e* durch benutzerdefinierte Konstanten überschreibt.

3.3. Ergebnis der Testfallmengen

Sebastian Dröge

Alle drei erstellten Testfallmengen wurden darauf getestet, wie viel der zu testenden Software sie überdecken. Explizit von der Codeüberdeckung und -Analyse ausgenommen sind der Integer-, sowie der Complex-Parser von *muParser*. Hierbei wurde zwischen den überdeckten Code-Zeilen, Funktionen und (Teil-) Bedingungen unterschieden. Die Ergebnisse sind in Abb. 3.2 zu finden. Diese Statistik wurde mit Hilfe der Open Source Software *lcov*[8] und *gcov*[9] erstellt.

Bei den Ergebnissen fällt auf, dass die Funktionsüberdeckung der Testfallmenge 1, die nach dem Verfahren der Funktionsüberdeckung gebildet wurde, nur knapp 70% der Funktionen der Software überdeckt sind. Der Grund hierfür ist, dass die Funktionsüberdeckung nur alle öffentlichen Funktionen getestet hat, diese Statistik allerdings auch alle internen Funktionen enthält. Die internen Funktionen, die von der Testfallmenge nicht überdeckt wurden, werden nur in bestimmten Sondernfällen ausgeführt, allerdings kann diese Testfallmenge natürlich auch keine Fehler in diesen Sonderfällen entdecken. Eine bessere

	Zeilen	Funktionen	Bedingungen
Testfallmenge 1: „Funktionsüberdeckung“	73.5%	69.6%	60.5%
Testfallmenge 2: „Grenzwertanalyse“	65.9%	61.5%	55.3%
Testfallmenge 3: „Intuitiv“	61.2%	62.2%	51.0%
$\sum$	78.9%	73.9%	66.5%

Abbildung 3.2.: Überdeckung von *muParser* durch die Testfallmengen

Überdeckung wäre in jedem Falle wünschenswert, für die Testfälle entwickelt werden müssen, die genau diese Sonderfälle provozieren. Ebenso verhält es sich mit der Testfallmenge 2, die zwar nach dem Verfahren der Bedingungsüberdeckung gebildet wurde, aber nur knapp 55% der Bedingungen überdeckt.

Im Vergleich zu den systematisch gebildeten Testfallmengen 1 und 2, hat die intuitiv gebildete Testfallmenge 3 die geringste Überdeckung, allerdings nur etwas geringer als die der Testfallmenge 2. Eine intuitiv gebildete Testfallmenge hat also keinen Vorteil gegenüber systematisch gebildeten Testfallmengen, kann unter Umständen allerdings vergleichbare Überdeckungen erreichen wie systematisch gebildete Testfallmengen.

Weiterhin fällt auf, dass alle drei Testfallmengen Teile der Software testen, die von keiner anderen Testfallmenge getestet werden, was später bei den Mutationstests ein Grund für unterschiedliche Ergebnisse der Testfallmengen ist. Mutanten in Teilen der Software, die von keiner der Testfallmengen überdeckt werden, können natürlich nicht erkannt werden.

4. Mutationsoperatoren

Martin Walter

Die Mutationsoperatoren definieren die Veränderungen im originalen Quellcode. Die Anwendung eines Mutationsoperators an einer Stelle im Quellcode führt zu einem definierten und bekannten Fehler in der Software. Diese fehlerhaften Ableger der originalen Software sind die Mutanten.

Die Menge der Mutationsoperatoren kann unterteilt werden in die Bereiche auf die sie wirken. Abbildung 4.1 stellt diese Bereiche hierarchisch dar.

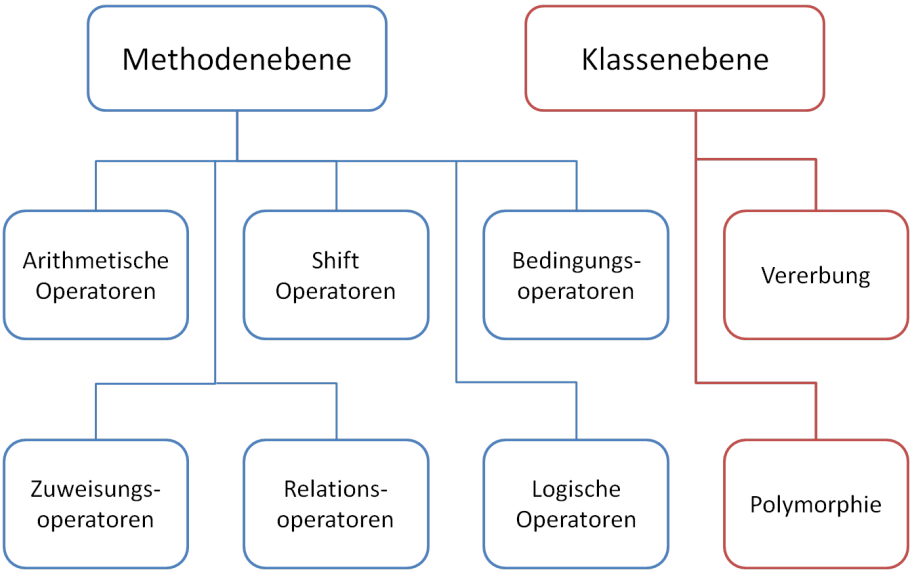

Abbildung 4.1.: Anwendungsgebiete der Mutationsoperatoren

Als erste grobe Unterscheidung dient die Anwendung der Mutationsoperatoren auf die Methoden- bzw. die Klassenebene. Auf Methodenebene werden die Mutationsoperatoren ausschließlich auf die Operatoren der Programmiersprache angewendet. Auf Klassenebene unterteilen sich die Operatoren auf die beiden Eigenschaften von objektorientierten Programmiersprachen, die Vererbung und die Polymorphie. Die folgende Auflistung benennt und charakterisiert die verwendeten Mutationsoperatoren die von diesen [10] [11] Java-Mutationsoperatoren abgeleitet sind:

- Mutationsoperatoren auf Methodenebene:
  - Anwendung auf arithmetische Operatoren:
    Die arithmetischen Operatoren bestehen aus den fünf zweiseitigen Operatoren

$+, -, *, /, \%$, den zwei einseitigen Operatoren $+$ *und* $-$, sowie den abgekürzten Operatoren $op++$, $++op$, $op--$, $--op$. Mutationsoperatoren :

  * AOR : *Arithmetic Operator Replacement*
    arithmetischen Operator durch anderen arithmetischen Operator ersetzen.
  * AOI : *Arithmetic Operator Insertion*
    arithmetischen Operator einfügen (nur für einseitigen oder abgekürzten arithmetischen Operator möglich).
  * AOD : *Arithmetic Operator Deletion*
    arithmetischen Operator entfernen (nur für einseitigen oder abgekürzten arithmetischen Operator möglich).

- Anwendung auf Shift Operatoren ($<<$, $>>$):
  * SOR : *Shift Operator Replacement*
    Shift Operator durch anderen Shift Operator ersetzen.
- Anwendung auf Bedingungsoperatoren (*& &*, $||$, *&*, $|$, *!*):
  * COR : *Conditional Operator Replacement*
    Bedingungsoperator durch anderen Bedingungsoperator ersetzen.
  * COI : *Conditional Operator Insertion*
    Bedingungsoperator einfügen (nur für den einseitigen Bedingungsoperator ! möglich).
  * COD : *Conditional Operator Deletion*
    Bedingungsoperator entfernen (nur für den einseitigen Bedingungsoperator ! möglich).
- Anwendung auf Zuweisungsoperatoren ($+=$, $-=$, $*=$, $/=$, ...):
  * ASR : *Short-Cut Assignment Operator Replacement*
    Zuweisungsoperator durch anderen Zuweisungsoperator ersetzen.
- Anwendung auf Relationsoperatoren ($>$, $>=$, $<$, $<=$, $==$, $!=$):
  * ROR : *Relational Operator Replacement*
    Relationsoperator durch anderen Relationsoperator ersetzen.
- Anwendung auf logische Operatoren (*&* , $|$, $\tilde{}$):
  * LOR : *Conditional Operator Replacement*
    logischen Operator durch anderen logischen Operator ersetzen.
  * LOI : *Conditional Operator Insertion*
    logischen Operator einfügen (nur für den einseitigen logischen Operator $\tilde{}$ möglich).

* LOD : *Conditional Operator Deletion*
  logischen Operator entfernen (nur für den einseitigen logischen Operator ~ möglich).

- Mutationsoperatoren auf Klassenebene:
  - Anwendung auf Vererbung
    * IOD : *Overriding method deletion*
      Eine Methodendeklaration aus einer erbenden Klasse, die die Methode aus der vererbenden Klasse überschreibt, entfernen.

Durch diese Mutationsoperatoren wurden 270 Mutanten gebildet. Die Anzahl der Mutanten die durch die einzelnen Mutationsoperatoren gebildet wurden können der Abbildung 4.2 entnommen werden.

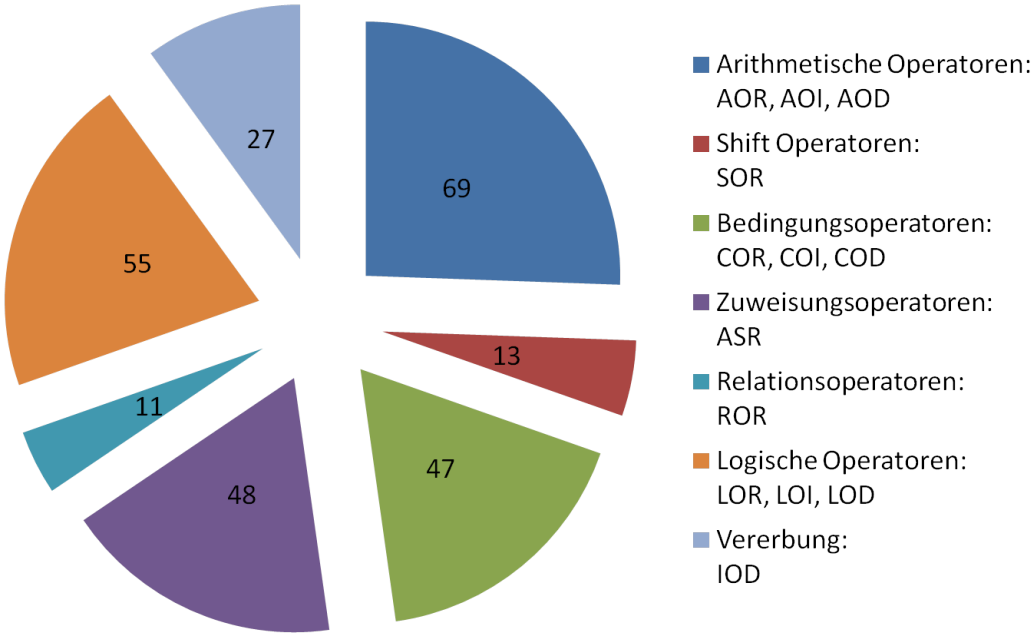

Abbildung 4.2.: Anzahl der Mutanten zu den genannten Mutationsoperatoren

5. Auswertung

Aufbauend auf Abschnitt 3.3 werden nachfolgend die Ergebnisse der Codeüberdeckung und Mutantenterkennung auf ihre Implikationen analysiert.

5.1. Anzahl erkannter Mutanten

Sebastian Dröge

Nach Erstellung der 270 Mutanten durch die im vorigen Kapitel vorgestellten Mutationsoperatoren, wurden getestet, wie viele der Mutanten durch die Testfallmengen gefunden werden. Mutanten befinden sich dabei im gesamten betrachteten Code, nicht nur in Codeteilen, die von den Testfallmengen explizit überdeckt werden. Die Ergebnisse sind in Tabelle 5.1 dargestellt.
Es fällt auf den ersten Blick auf, dass Testfallmenge 1 mit der höchsten Überdeckung am meisten Mutanten erkennt, Testfallmenge 2 mit der zweithöchsten Überdeckung am zweit meisten Mutanten erkennt und Testfallmenge 3 am wenigsten. Allerdings werden verhältnismäßig wenig Mutanten überhaupt erkannt, im Fall von Testfallmenge 2 und 3 gerade einmal die Hälfte. Auch bei diesen Ergebnissen zeigt sich wieder, dass jede Testfallmenge Code überdeckt, der von keiner der anderen Testfallmengen überdeckt wird, da die Summe der erkannten Mutanten deutlich höher ist, als die Anzahl der Mutanten, die von Testfallmenge 1 entdeckt wurden.

5.2. Analyse der nicht-erkannten Mutanten

Sebastian Dröge

	Prozentsatz getöteter Mutanten
Testfallmenge 1: „Funktionsüberdeckung“	65.6%
Testfallmenge 2: „Grenzwertanalyse“	53.7%
Testfallmenge 3: „Intuitiv“	49.3%
$\sum$	71.1%

Tabelle 5.1.: Prozentsatz getöteter Mutanten pro Testfallmenge

Bei der Analyse der nicht-erkannten Mutanten wurden 31% der nicht-erkannten Mutanten, oder etwa 9% aller Mutanten, als äquivalent zur Ausgangssoftware erkannt. Als äquivalente Mutanten werden solche bezeichnet, die sich genauso verhalten, wie die ursprüngliche Software. Andere Mutationstests kamen auf einen ähnlich hohen Anteil an äquivalenten Mutanten.

Für die restlichen, nicht-äquivalenten Mutanten wurde versucht, neue Tests zu erstellen. Für 16% dieser nicht-erkannten Mutanten war dies möglich, allerdings wurden durch diese neuen Tests keine Fehler in *muParser* gefunden.

Für die anderen 53% der nicht-erkannten Mutanten war es nicht möglich Tests zu erstellen, da kein Zustand erzeugt werden konnte, durch den der Code in dem die Änderung lag, ausgeführt wurde. Ein Großteil dieser Mutanten waren in der Methode `mu::ParserBase::ParseCmdCode`, für die es nicht möglich war im Rahmen dieses Projektes einen Test zu schreiben. Allerdings sollte es theoretisch möglich sein, diese Methode auszuführen, es ist allerdings nicht ersichtlich welcher Zustand hierfür erreicht werden muss.

5.3. Bewertung der Testfallmengen

Carsten Rösnick

Aufbauend auf Abschnitt 3.3 folgt nun eine genauere Aufschlüsselung der nicht erkannten, sowie äquivalenten Mutanten nach Mutationsoperatoren. An dessen Betrachtung schließt sich überdies eine Bewertung der gewählten Mutationsoperatoren bezüglich der untersuchten Software (muParser) an.
Nach Filterung der nicht erkannten Mutanten besteht die besondere Herausforderung in deren Charakterisierung, d.h. in der Bestimmung, ob es sich um von den Testfallmengen nicht abgedeckten Mutanten, oder gar um äquivalente handelt. Dabei ist die Bestimmung, ob es sich um einen äquivalenten Mutanten handelt im Allgemeinen ein hartes Problem [3]. Für dieses Projekt wurden demnach die entsprechenden Codeabschnitte (der Mutanten) händisch auf Äquivalenz zum Originalprogramm geprüft. Die Ergebnisse dieser Überlegungen sind in Tabelle 5.2 zusammengefasst.

	AOD	AOI	AOR	SOR	COI	COR	COD	ASR	LOD	LOI	LOR	ROR	IOD
alle	18	27	24	13	17	18	12	48	1	38	16	11	27
unerkannt	4	4	3	11	4	6	3	10	0	22	7	0	0
äquivalent	1	3	0	1	1	4	2	1	0	8	0	0	0
%unerkannt	22.2%	14.8%	12.5%	84.6%	23.5%	33.3%	25.0%	20.8%	0.0%	57.9%	43.8%	0.0%	0.0%
%äquivalent	5.6%	11.1%	0.0%	7.7%	5.9%	22.2%	16.7%	2.1%	0.0%	21.1%	0.0%	0.0%	0.0%

Tabelle 5.2.: Nicht erkannte und äquivalente Mutanten pro Mutationsoperator

Bemerkungen zum Mutationsoperator SOR

Der Mutationsoperator SOR (*Shift Operator Replacement*) sticht, was den prozentualen Anteil der nicht erkannten Mutanten angeht, hervor. Die Gründe hierfür sind vielfältig; einige davon werden nachfolgend benannt.
Zuvor sei jedoch noch eine Erklärung der durchgeführten Alternierung angeführt. SOR-Mutanten wurden ausschließlich im in Listing 5.1 eingefügten Codeabschnitt der ParserTokenReader-Klasse durch Austausch des binären Linksshift `<<` durch den binären Rechtsshift `>>` erzeugt. Da diese Fehlercodes der Erkennung von unerwarteten und nicht erlaubten Zeichen in von muParser zu parsenden Zeichenketten dienen, gleicht ein „auf Null setzen“ (was das Resultat des Rechtsshifts der 1 um $k \geq 1$ Bits darstellt) dem Herausnehmen des assoziierten Fehlercodes aus der Menge möglicher Fehler. Randbemerkung: Eine Ausnahme stellt hier der Fehlercode `noBO` dar.

Listing 5.1: Auszug aus der Klasse ParserTokenReader

```
enum ESynCodes {
    noBO      = 1 << 0,  ///< to avoid i.e. "cos(7)("
    noBC      = 1 << 1,  ///< to avoid i.e. "sin)" or "()"
    noVAL     = 1 << 2,  ///< to avoid i.e. "tan 2" or "sin(8)3.14"
    noVAR     = 1 << 3,  ///< to avoid i.e. "sin a" or "sin(8)a"
    noARG_SEP = 1 << 4,  ///< to avoid i.e. ",," or "+," ...
    noFUN     = 1 << 5,  ///< to avoid i.e. "sqrt cos" or "(1)sin"
    noOPT     = 1 << 6,  ///< to avoid i.e. "(+)"
    noPOSTOP  = 1 << 7,  ///< to avoid i.e. "(5!!)" "sin!"
    noINFIXOP = 1 << 8,  ///< to avoid i.e. "++4" "!!4"
    noEND     = 1 << 9,  ///< to avoid unexpected end of formula
    noSTR     = 1 << 10, ///< to block numeric arguments on string functions
    noASSIGN  = 1 << 11, ///< to block assignement to constant i.e. "4=7"
    noANY     = ~0       ///< All of he above flags set
};
```

Wie bereits bemerkt sind die Gründe für die unerwartet hohe Anzahl nicht erkannter SOR-Mutanten vielfältig:

1. `m_iBrackets`, die Anzahl der öffnenden und schließenden Klammern, wird stets auf Balanciertheit geprüft und erzeugt entsprechend Exceptions. D.h., Klammerfehler werden bereits durch den internen Klammercheck, unabhängig von obigen Fehlercodes, abgedeckt.

2. Die nicht erlaubten Operationen, aufgelistet in `ParserTokenReader::ESynCodes` (siehe Listing 5.1), überlappen sich teilweise. D.h., fällt ein Fehlercode weg, so fangen die internen Tests in Kombination mit den verbliebenen Fehlercodes die meisten[1] Fehler dennoch ab.

[1] Im Rahmen dieser Auswertung wurde weder formal bewiesen noch widerlegt, dass jeweils ein Fehlercode entfernt werden kann, ohne jedoch die Fehlererkennung zu beeinflussen. Die Erkenntnisse der Codeanalyse deuten darauf hin, dass es kaum möglich ist Zeichenketten zu konstruieren, die Fehler enthalten, die nach einer SOR-Mutation jedoch nicht mehr gefunden werden.

3. Beispielausdruck: "()". Dieser resultiert in einem leeren Value- und Operator Stack, was in `ParserStack::pop()` zu einer Exception führt. Dieser Fehler würde auch ohne `noBC`-Check gefunden. D.h., auch die Anzahl der Argumente wird entsprechend, unabhängig von obigen Fehlercodes, geprüft.

Beispiel der eindeutigen Charaktierisierung als äquivalenter Mutant

Abschließend sei noch ein Beispiel für die positive Erkennung eines Mutanten als äquivalent angegeben. Als Beispiel diene ein AOI-Mutant. Der Aufruf von `ApplyFunc(*,*,2)` wurde hierbei durch `ApplyFunc(*,*,-2)` ersetzt. Werden im gesetzten Ausdruck nun benutzerdefinierte binäre Operatoren verwendet, so wird der dritte Parameter `a_iArgCount` (hier: -2 beim Mutanten) schlicht ignoriert. Der Mutant ist demnach äquivalent.

A. Testumgebung

Die Testumgebung beinhaltet nebst den Testfallmengen `testsuite-`n ($n = 1..3$) ebenso den *muParser* Source Code, die Mutanten (jeweils einer pro *git branch*[1]), sowie die Testfallmenge 4 (`testsuite-4`). In letzterer sind alle Tests enthalten, die gezielt nach Auswertung der nicht erkannten Mutanten geschrieben wurden um selbige zu erkennen.

A.1. Kompilieren der Testumgebung

Nachfolgenden seien die wesentlichen Kommandos zum bauen, ausführen und debuggen der Testumgebung aufgeführt.

- Zunächst `./autogen.sh [--enable-maintainer-mode]`
- ... gefolgt von `make`
- Ausführen der Tests: `make check`
- Debuggen einzelner Tests: `make testsuite-n/test-*.gdb`

[1] siehe dazu *branches.list*. Zum wechseln in einen branch diene *git checkout <branch name>*.

Literaturverzeichnis

[1] Liggesmeyer, P.: *Software-Qualität. Testen, Analysieren und Verifizieren von Software*, Spektrum Akademischer Verlag, 2009, ISBN-10 3827420563

[2] Roitzsch, P.: *Analytische Softwarequalitätssicherung in Theorie und Praxis*, Monsenstein und Vannerdat, 2005, ISBN-10 3865822029

[3] Petrank, E.; Roth, R.M.; Center, D. und Piscataway, NJ.: *Is code equivalence easy to decide?*, IEEE Transactions on Information Theory, 1997, Ausgabe 43:5, 1602–1604

[4] muParser: http://sourceforge.net/projects/muparser

[5] CppUnit: http://sourceforge.net/projects/cppunit

[6] Boost::Test: http://www.boost.org/doc/libs/1_43_0/libs/test/doc/html/index.html

[7] GoogleTest: http://code.google.com/p/googletest/

[8] lcov: http://ltp.sourceforge.net/coverage/lcov.php

[9] gcov: http://gcc.gnu.org/onlinedocs/gcc/Gcov.html

[10] Offutt, J.; Yu-Seung, M.: *Description of Class Mutation Mutation Operators for Java*, Information and Software Engineering George Mason University, November 7, 2005

[11] Offutt, J.; Yu-Seung, M.: *Description of Method-level Mutation Operators for Java*, Information and Software Engineering George Mason University, November 29, 2005

[12] Akinci, S: *Codeabdeckung anhand der Kontrollflussorientierten Software-Testmethoden*, Technische Universität Berlin, 2003; http://swt.cs.tu-berlin.de/lehre/seminar/ss03/folien/codeabdeckung.pdf