Fachhochschule Nordwestschweiz
Hochschule für Technik

MASTER OF SCIENCE
IN ENGINEERING

FHNW Brugg/Windisch

Projektbericht P8

AutoFeedback – Korrektheit

Student	Joel Emmenegger joel.emmenegger@fhnw.ch
Advisor	Prof. Dr. Christoph Stamm christoph.stamm@fhnw.ch

Emmenegger Joel
20.8.2021

SUMMARY

Die Arbeit beschäftigt sich mit der automatischen Evaluation studentischer Java-Programmierlösungen bezüglich Korrektheit. Dafür soll ein Evaluationstool erstellt werden, welches die mathematische Korrektheit gegenüber der Spezifikation von verschiedenen Java-Programmierlösungen überprüfen kann.

Verschiedene bestehende Evaluationstools wurden auf ihre Tauglichkeit geprüft. Dafür wurden Verifizierungsbeispiele erstellt. Das sind einfache Java-Programme mit ihrer jeweiligen Spezifikation bestehend aus Vor- und Nachbedingungen und Invarianten. Anhand der Verifizierungsbeispiele wurde OpenJML als das am besten passenden Tool ermittelt.

OpenJML ist ein Programm-Verifizierungstool für Java Programme, welches, die in der Java Modeling Language (JML) annotierte Spezifikation, überprüft. Die Java Modeling Language ist eine Spezifikationssprache für Java Programme, welche dem Design by contract Paradigma folgt. Für die Spezifikation werden Vor- und Nachbedingungen und Invarianten im Hoare-Style verwendet.

Da auch OpenJML nicht alle Anforderungen erfüllt, wird OpenJML mit einem Wrapper umschlossen. Dieser ermöglicht es automatisch, einfache For-Schleifen auszurollen, Overflows zu ignorieren und logische Vorbedingungen hinzuzufügen. Dafür wird die studentische Java-Programmierlösung vom Wrapper eingelesen und überarbeitet. Anschliessend wird der Code mit der überarbeiteten Spezifikation von OpenJML auf mathematische Korrektheit gegenüber der Spezifikation überprüft und die Resultate werden zu einem Feedback aufbereitet, welches den Studenten präsentiert werden kann.

Um das erstellte Tool benutzbar zu machen, wurde dieses in das abgeschlossene Lehrfonds-Projekt «Automatisches Feedback zu Programmieraufgaben» des Instituts für Mobile und Verteilte Systeme integriert.

INHALTSVERZEICHNIS

1. EINLEITUNG

Für Programmieranfänger ist es wichtig Programmieraufgaben selbständig lösen zu können und für ihre Lösungen ein sinnvolles und individuelles Feedback zu erhalten. Im Studiengang Informatik an der Hochschule für Technik fallen ins erste Studienjahr Module, in welchen die Grundlagen der Programmierung mit Java vermittelt werden. Das Betreuungsintensive aber besonders Wertvolle an diesen Modulen ist, das Begutachten der Lösungen von Programmieraufgaben, welche die Studierenden im Rahmen des begleiteten Selbststudiums erarbeiten. Die Beurteilung beinhaltet Hinweise zu falschen, verbesserungswürdigen und guten Teilen in ihren Programmen. Diese Beurteilung sollte den Studierenden möglichst zeitnah zur Verfügung gestellt werden. Um bei wachsender Anzahl Studierenden die personellen Ressourcen für die enge Betreuung nicht zu überbeanspruchen, soll ein automatisches Feedbacksystem eingesetzt werden.

Im abgeschlossenen Lehrfonds-Projekt «Automatisches Feedback zu Programmieraufgaben» des Instituts für Mobile und Verteilte Systeme und in der Projektarbeit «AutoFeedback – Performance» [1] sind bereits erste Teile eines automatischen Feedbacksystems, AutoFeedback, umgesetzt worden. Daraus resultierte eine Webapplikation, welche Studentenprogramme serverseitig überprüft. Die Studentenprogramme können anhand von vordefinierten Unit Tests auf Korrektheit und anhand von vordefinierten Performance Tests auf Performance analysiert werden.

1.1. AUTOFEEDBACK PROJEKT

1.1.1. AUFBAU DER WEBAPPLIKATION

Es gibt zwei verschiedene Benutzertypen: «Dozenten» und «Studenten». Mit einem Dozentenaccount können neue Aufgaben «Tasks» erstellt werden. Ein Task beinhaltet einen Titel, eine Beschreibung, eine Zip-Datei mit den Unit Tests und eine Liste von Klassennamen, welche beim Testen ausgeführt werden sollen. Um das automatische Testen von Studentenlösungen zu ermöglichen, definiert der Dozent Interfaces, welchen die Studentenlösungen entsprechen müssen.

Um das Benutzen der Applikation für die Studenten zu vereinfachen, werden die Tasks in «Modulen» zusammengefasst. Dabei ist ein Modul eine sortierte Sammlung von Tasks, die beispielsweise während eines Semesters von den Studenten gelöst werden sollen. Ein Task kann dabei in verschiedenen Modulen enthalten sein. Die Zugriffsrechte auf Tasks sind auf der Ebene der Module geregelt, damit Studenten nicht zu jedem Task separat eingeladen werden müssen.

Wenn ein Student seine Lösung zu einer Aufgabe testen möchte, wählt er den dazugehörigen Task aus und lädt seine Java-Klassen als Zip-Datei hoch. Anschliessend wird der Code kompiliert und evaluiert. Nach der Evaluation werden die Resultate dem Studenten mit einem Feedback im Task angezeigt.

1.1.2. EVALUATION DER PROGRAMME

Die Evaluation einer Studentenlösung auf dem Server läuft wie folgt ab: Sobald eine noch nicht evaluierte Studentenlösung auf dem Server verfügbar ist, wird diese von einem Hintergrundprozess bearbeitet. Dabei werden in einem ersten Schritt sowohl die Java-Klassen der Tests als auch der Lösung in einem Docker-Container kompiliert. Anschliessend wird eine Java-Applikation ausgeführt, welche mittels Reflexion die Tests vom Dozenten und die Studentenlösung lädt. Die Tests werden ausgeführt und die Resultate inklusive Feedbacks in eine Datenbank geschrieben. Es können dabei sowohl Unit Tests als auch Performancetests überprüft werden. Das Feedback kann anschliessend vom jeweiligen Studenten in der Webapplikation eingesehen werden.

1.2. Ziele

Das Ziel der Projektarbeit P8 ist die Evaluation von Java-Programmierlösungen der Studenten bezüglich Korrektheit sowie die Integration der Code-Verifikation in die AutoFeedback-Applikation. Dies umfasst die Beantwortung folgender Fragen:

1. Welche Tools zur Verifikation von Java-Programmcode sind verfügbar? Was leisten sie? Wo sind ihre Grenzen?
2. Wie kann die Korrektheit von Java-Programmen effizient und mathematisch korrekt überprüft werden? Kann ein Ansatz, basierend auf Hoare-Triples verwendet werden?
3. Lassen sich alle Arten von Programmen verifizieren oder muss auf eine vordefinierte Untermenge von Java-Programmen zurückgegriffen werden?

Folgende neuen Funktionen soll die AutoFeedback-Applikation durch die Erweiterung dieser P8 Arbeit erhalten:

- Verifizierung der Studentenprogramme mittels Vor- und Nachbedingungen unter Einhaltung von Invarianten.
- Berechnung der schwächsten Vorbedingung (Weakest Precondition).
- Vor- und Nachbedingungen für Teile eines Programmes validieren, nicht nur für komplette Methoden.
- Erkennen von möglichen Fehlern aufgrund von Variablenüberläufen (Overflows).
- Bei inkorrekten Programmen aufzeigen von Bedingungen, unter welchen die Spezifikation nicht eingehalten werden kann.

Die bestehenden Tools werden anhand einer nachfolgend definierten Kriterienliste evaluiert. Das Erreichen der Ziele Nummer zwei und drei wird anhand der folgenden Verifizierungsbeispiele gemessen.

1.3. Aufbau des Berichts

Im folgenden Bericht werden im Kapitel 2 verschiedene Verifizierungsbeispiele definiert und beschrieben. Diese dienen zur Bewertung der entwickelten Verifikation und werden teilweise auch für die Analyse der bestehenden Tools eingesetzt. Anschliessend werden im Kapitel 3 die bestehenden Tools für die Verifizierung von Java-Programmcodes analysiert. Im Kapitel 4 wird das entwickelte Verifizierungs-Tool erläutert und anhand der Verifizierungsbeispiele getestet. Im Kapitel 5 werden alle entwickelten und geänderten Projekte beschrieben. Dies beinhaltet einerseits, wie die Verifikation in das AutoFeedback Projekt integriert wird und andererseits wird gezeigt wie Dozenten und Studenten die Applikation verwenden können, sowie welche Anforderungen an ihre Projekte gestellt werden. Der Bericht wird mit der Diskussion im Kapitel 6 abgeschlossen.

2. VERIFIZIERUNGSBEISPIELE

Für die Überprüfung des Verifizierungstools und für die Evaluation er bestehenden Tools werden repräsentative Verifizierungsbeispiele benötigt. Zu jedem dieser Beispiele liegt eine Lösung mit korrekten Vor- und Nachbedingungen und allen erforderlichen Invarianten vor. Anhand dieser Programme kann gezeigt werden, ob diese zuverlässig und ohne zusätzliche Vorbedingungen und Invarianten verifiziert werden können. Die Beispiele bestehen aus der Überprüfung von einzelnen Anweisungen und Kontrollstrukturen und einfachen Beispielprogrammen, welche in den Programmier-Grundlagen-Modulen der ersten Studienjahre auch verifiziert werden könnten.

Die folgenden Programme wurden ausgewählt: eine Zuweisung, ein Swap, eine If-Else Anweisung, eine For-Schleife, eine Multiplikation, eine Integer Division, eine Integer Quadratwurzelberechnung, eine Fakultätsberechnung und eine Fibonacci Berechnung. Der Code zu den Verifikationsbeispielen ist im Projekt AutoFeedback Library[1] zu finden. Die Beispiele befinden sich im Testordner im Package *ch.fhnw.autofeedback.verificationExamples* und sind nach ihrem englischen Namen benannt.

In den folgenden Kapiteln werden die einzelnen Verifizierungsbeispiele mit ihren Vor-, Nachbedingungen und Invarianten beschrieben. Es wird ebenfalls mit Hilfe der Hoare Logic gezeigt, dass die jeweiligen Vor- und Nachbedingungen und Invarianten valide sind.

2.1. HOARE LOGIC

Die Programme werden anhand der Hoare Logic mittels Vor- und Nachbedingungen überprüft. Dies geschieht mit sogenannten Hoare-Triplen $\{R\}\ P\ \{S\}$. Ein Hoare-Triple ist gültig, wenn die Vorbedingung R vor der Ausführung des Programms P erfüllt ist und nach der Ausführung des Programmes P die Nachbedingung S erfüllt ist. Je schwächer die Vorbedingung R ist desto besser ist dies, weil weniger einschränkend. Die schwächste Vorbedingung heisst Weakest Precondition (WP). [2, 3]

Ablauf einer Verifikation

Aus S und P kann die schwächste Vorbedingung $WP \equiv wp(P,S)$ berechnet werden. Falls nun $\{R \Longrightarrow WP\}$ gilt, erfüllt das Programm P die Nachbereitung S bei eingehaltener Vorbedingung R. Die WP kann ausgehend von der Nachbedingung S Anweisung für Anweisung rückwärts berechnet werden.

WP-Berechnungsregel für Zuweisung

Zuweisung $P \equiv \{x \coloneqq f\}$ wobei x eine Variabel und f ein Ausdruck ist.

$WP \equiv wp(P,S) \equiv wp\big((x \coloneqq f), S\big) \equiv S'$ alle ungebundenen x werden durch f ersetzt.

Beispiel: $\{x + 1 = 42\}\ y \coloneqq x + 1\ \{y = 42\}$

WP-Berechnungsregel für Komposition

Bei sequenziellen Programmen können einzelne Tripel mit folgender Regel verknüpft werden: Wenn $\{R\}\ P_1\ \{S_1\}$ und $\{S_1\}\ P_2\ \{S_2\}$ gilt dann gilt auch $\{R\}\ P_1; P_2\ \{S_2\}$.

$WP \equiv wp(P,S) \equiv wp(P_1, wp(P_2, S))$

[1] Projekt AutoFeedback Library: https://gitlab.fhnw.ch/autofeedback/autofeedback-library

WP-Berechnungsregel für Selektion (if-then-else-Regel)

Für die Selektion gibt es zwei Auswahlkonstrukte für die beiden Auswahlmöglichkeiten $\{R \wedge B\}\ P_1\ \{S\}$ und $\{R \wedge \neg B\}\ P_2\ \{S\}$ für das Programm $\{R\}\ if\ B\ then\ P_1\ else\ P_2\ \{S\}$. B ist ein logischer Ausdruck.

$$WP \equiv wp(P,S) \equiv wp\big((if\ B\ then\ P_1\ else\ P_2), S\big) = \big(B \Longrightarrow wp(P_1,S) \wedge \neg B \Longrightarrow wp(P_2,S)\big)$$
$$\equiv \big(\neg B \vee wp(P_1,S)\big) \wedge \big(B \vee wp(P_2,S)\big)$$
$$\equiv \neg B \wedge B \vee \neg B \wedge wp(P_2,S) \vee B \wedge wp(P_1,S) \vee wp(P_1,S) \wedge wp(P_2,S)$$
$$\equiv false \vee \neg B \wedge wp(P_2,S) \vee B \wedge wp(P_1,S)$$
$$\equiv B \wedge wp(P_1,S) \vee \neg B \wedge wp(P_2,S)$$

WP-Berechnungsregel für Iteration (While-Regel)

Für die Verifizierung einer Schleife wird eine Schleifeninvariante I benötigt. Diese muss vor, während als auch nach der Ausführung der Schleife gültig sein. Die Invariante I muss manuell ermittelt werden und wird wie die Vor- und Nachbedingung zur Verifizierung eines Programms benötigt. Das Hoare-Triple $\{I \wedge B\}\ P\ \{I\}$ gilt für Schleifen $\{I\}\ while\ B\ do\ P\ done\ \{I \wedge \neg B\}$. B ist ein logischer Ausdruck.

$$WP \equiv wp(P,S) \equiv I \wedge \big((I \wedge B) \Longrightarrow wp(P,I)\big) \wedge \big((I \wedge \neg B) \Longrightarrow S\big)$$

Die Invariante ist korrekt, wenn $\{(I \wedge B) \Longrightarrow wp(P,I)\}$ gilt.

Die Nachbedingung ist erfüllt, wenn $\{(I \wedge \neg B) \Longrightarrow S\}$ gilt.

2.2. Zuweisung

```
pre true;
x := 5;
post x = 5;
```

Code 1 Zuweisung

```
pre true;
// VC1
// {5 = 5}
x := 5;
// {x = 5}
post x = 5;
```

Code 2 WP Berechnung Zuweisung

Damit das Zuweisungsprogramm valide ist, muss die Verifikationsbedingung (VC1) in Formel 1 valide sein. Dies ist in diesem Fall trivial und kann bestätigt werden.

$true$	Vorbedingung
$\Longrightarrow 5 = 5$	Weakest Precondition

Formel 1 Zuweisung Verifikationsbedingung 1

2.3. Swap

```
pre true;
t := x;
x := y;
y := t;
post y = a && x = b;
```

Code 3 Swap

```
pre true;
// {x = x~ && y = y~}
t := x;
// {t = x~ && y = y~}
x := y;
// {t = x~ && x = y~}
y := t;
// {y = x~ && x = y~}
post y = x~ && x = y~;
```

CODE 4 WP BERECHNUNG SWAP

Damit das Swap-Programm valide ist, muss die Verifikationsbedingung (VC1) Formel 2 valide sein. Die Notation ~ steht für den Wert, welche die Variable zum Zeitpunkt der Vorbedingung hat. In diesem Beispiel stehen $x\sim$ und $y\sim$ für die Werte der Variablen x und y vor dem Swap. Dies ist in diesem Falle trivial und kann bestätigt werden.

$true$	Vorbedingung
$\Longrightarrow x = x\sim \wedge y = y\sim$	Weakest Precondition

FORMEL 2 SWAP VERIFIKATIONSBEDINGUNG 1

2.4. IF-ELSE ANWEISUNG

```
pre x > 1 || x < 0;
if x < 0 then
    x := -x;
else
    x := x -1;
end
post x > 0;
```

CODE 5 IF-ELSE ANWEISUNG

```
pre x > 1 || x < 0;
// VC1
// {(x < 0 && -x > 0) || (!(x < 0) && x -1 > 0)}
if x < 0 then
    // {-x > 0}
    x := -x;
    // {x > 0}
else
    // {x -1 > 0}
    x := x -1;
    // {x > 0}
end
post x > 0;
```

CODE 6 WP BERECHNUNG IF-ELSE ANWEISUNG

Damit das If-Else Anweisungsprogramm valide ist, muss die Verifikationsbedingung (VC1) Formel 3 valide sein. Dies ist mithilfe einer Vereinfachung der Weakest Precondition anschliessend trivial und kann bestätigt werden.

$(x > 1 \vee x < 0)$	Vorbedingung
$\Longrightarrow (x < 0 \wedge -x > 0) \vee (x \not< 0 \wedge x - 1 > 0)$	Weakest Precondition
$\equiv \big(x < 0 \ \vee (\ x \geq 0 \ \wedge x > 1)\big)$	Vereinfachen
$\equiv (x < 0 \ \vee x > 1)$	Vereinfachen

FORMEL 3 IF-ELSE VERIFIKATIONSBEDINGUNG 1

2.5. For-Schleife

```
pre x = -15;
for i in 1..5 do
    x := x + i;
end
post x = 0;
```

Code 7 For-Schleife

```
pre x = -15;
// VC1
// {x + 5 + 4 + 3 +2 + 1 = 0}
for i in 1..5 do
    // {x + i = 0}
    x := x + i;
    // {x = 0}
end
post x = 0;
```

Code 8 WP Berechnung For-Schliefe

Damit das For-Schleife-Programm valide ist, muss die Verifikationsbedingung (VC1) Formel 4 valide sein. Dies ist mithilfe einer Vereinfachung der Weakest Precondition anschliessend trivial und kann bestätigt werden.

$x = -15$	Vorbedingung
$\Longrightarrow x + 5 + 4 + 3 + 2 + 1 = 0$	Weakest precondition
$\equiv x = -15$	Vereinfachen

Formel 4 For-Schleife Verifikationsbedingung 1

2.6. Multiplikation

```
pre x >= 0;
a := x; b := y; r := 0;
while a != 0 do
    inv a >= 0 && a*b + r = x*y;
    if a & 1 = 1 then
        r := r + b;
    end
    b := b * 2;
    a := a / 2;
end
post r = x*y;
```

Code 9 Multiplikation

```
pre x >= 0;
// VC1
// {x >= 0 && x*y = x*y}
a := x;
// {a >= 0 && a*y = x*y}
b := y;
// {a >= 0 && a*b = x*y}
// {a >= 0 && a*b + 0 = x*y}
r := 0;
// {invar: a >= 0 && a*b + r = x*y}
while a != 0 do
    inv a >= 0 && a*b + r = x*y;
    // {invar && guard: (a >= 0 && a*b + r = x*y) && (a != 0)}
    // VC2
    // {((a&1 =1) => ((a/2) >= 0 && (a/2)*2*b + r + b = x*y))) && (!(a&1 = 1) => ((a/2) >= 0 &&
(a/2)*2*b + r = x*y))}
    if a & 1 = 1 then
        // {(a/2) >= 0 && (a/2)*2*b + (r+b) = x*y}
        r := r + b;
        // {(a/2) >= 0 && (a/2)*2*b + r = x*y}
    End
    // {(a/2) >= 0 && (a/2)*2*b + r = x*y}
    b := b * 2;
    // {(a/2) >= 0 && (a/2)*b + r = x*y}
    a := a / 2;
    // {invar: a >= 0 && a*b + r = x*y}
end
// {invar && not guard: a >= 0 && a*b + r = x*y && !(a != 0)}
// VC3
post r = x*y;
```

CODE 10 WP BERECHNUNG MULTIPLIKATION

Damit das Multiplikationsprogramm valide ist müssen alle Verifikationsbedingung (VC1, VC2 und VC3) valide sein. Dies ist bei allen drei der Fall und das Programm ist damit valide.

VC1: In der VC1 (Formel 5) kann die Weakest Precondition vereinfacht werden. Sie entspricht der Vorbedingung und kann bestätigt werden.

$x \geq 0$	Vorbedingung
$\Longrightarrow x \geq 0 \ \wedge xy + 0 = xy$	Weakest Precondition
$\equiv x \geq 0$	Vereinfacht

FORMEL 5 MULTIPLIKATION VERIFIKATIONSBEDINGUNG 1

VC2: In der VC2 (Formel 6) können sowohl der ungerade (Formel 7) als auch der gerade Fall (Formel 8) zum selben Term vereinfacht werden. Anschliessend können die beiden Fälle zusammengefasst werden und entsprechen der Vorbedingung.

$a \geq 0 \ \wedge ab + r = xy \ \wedge a \ \neq 0$	Invariante und Guard
$\Longrightarrow OddCase \wedge EvenCase$	Weakest Precondition
$\equiv \frac{a}{2} \geq 0 \ \wedge ab + r = xy$	Einsetzten des Odd- und Evencases

FORMEL 6 MULTIPLIKATION VERIFIKATIONSBEDINGUNG 2

$Odd(a) = \frac{a}{2} \geq 0 \wedge \left\lfloor \frac{a}{2} \right\rfloor 2b + r + b = xy$	Weakest Precondition in Oddcase
$\equiv \frac{a}{2} \geq 0 \wedge \frac{a-1}{2} 2b + r + b = xy$	Vereinfachen
$\equiv \frac{a}{2} \geq 0 \wedge (a-1)b + r + b = xy$	Vereinfachen
$\equiv \frac{a}{2} \geq 0 \wedge ab + r = xy$	Vereinfachen

FORMEL 7 MULTIPLIKATION ODDCASE

$Even(a) = \frac{a}{2} \geq 0 \wedge \left\lfloor \frac{a}{2} \right\rfloor 2b + r = xy$	Weakest Precondition in Evencase
$\equiv \frac{a}{2} \geq 0 \wedge ab + r = xy$	Vereinfachen

FORMEL 8 MULTIPLIKATION EVENCASE

VC3: In der VC3 (Formel 9) kann die Invariante und nicht Guard vereinfacht werden. Sie entspricht der Nachbedingung und kann bestätigt werden.

$a \geq 0 \wedge ab + r = xy \wedge a = 0$	Invariante und nicht Guard
$\equiv a = 0 \wedge 0b + r = xy$	Einsetzen von $a = 0$
$\equiv a = 0 \wedge r = xy$	Vereinfachen
$\Longrightarrow r = xy$	Nachbedingung

FORMEL 9 MULTIPLIKATION VERIFIKATIONSBEDINGUNG 3

2.7. INTEGER DIVISION

```
pre a >= 0 && b > 0;
q := 0;
r := a;
while r >= b do
    inv a = b *q + r && 0 <= r;
    q := q + 1;
    r := r - b;
end
post a = b * q + r && 0 <= r && r < b;
```

CODE 11 INTEGER DIVISION

```
pre a >= 0 && b > 0;
// VC1
// {a = b*0 + a && 0 <= a}
q := 0;
// {a = b*q + a && 0 <= a}
r := a;
// {invar: a = b*q + r && 0 <= r}
while r >= b do
    inv a = b*q + r && 0 <= r;
    // {invar && guard: a = b *q + r && 0 <= r && r >= b}
    // VC2
    // {a = b*(q+1) + (r-b) && 0 <= r - b}
    q := q + 1;
    // {a = b*q + (r-b) && 0 <= r - b}
    r := r - b;
    // {invar: a = b*q + r && 0 <= r}
end
// {invar && not guard: a = b*q + r && 0 <= r && !(r >= b)}
// VC3
post a = b*q + r && 0 <= r && r < b;
```

CODE 12 WP BERECHNUNG INTEGER DIVISION

Damit die Integer Division valide ist, müssen alle Verifikationsbedingungen (VC1, VC2 und VC3) valide sein. Dies ist bei allen drei der Fall und das Programm ist damit valide.

VC1: In der VC1 (Formel 10) kann die Weakest Precondition vereinfacht werden. Sie entspricht der Vorbedingung und kann somit bestätigt werden. Der zweite Teil der Vorbedingung $b > 0$ ist nicht in der Weakest Precondition enthalten. Falls jedoch $b \leq 0$ wäre, würde die Schleife nicht terminieren.

$a \geq 0 \wedge b > 0$	Vorbedingung
$\Longrightarrow a = b \cdot 0 + a \wedge a \leq 0$	Weakest Precondition
$\equiv a = a \wedge a \leq 0$	Vereinfachen
$\equiv a \geq 0$	Vereinfachen

FORMEL 10 INTEGER DIVISION VERIFIKATIONSBEDINGUNG 1

VC2: In der VC2 (Formel 11) kann die Weakest Precondition vereinfacht werden. Sie entspricht der Invariante und Guard und kann bestätigt werden.

$a = b \cdot q + r \wedge 0 \leq r \wedge r \geq b$	Invariante und Guard
$\Longrightarrow a = b \cdot (q + 1) + (r - b) \wedge 0 \leq (r - b)$	Weakest Precondition
$\equiv a = b \cdot q + b + r - b \wedge 0 \leq r - b$	Vereinfachen
$\equiv a = b \cdot q + r \wedge 0 \leq r - b$	Vereinfachen

FORMEL 11 INTEGER DIVISION VERIFIKATIONSBEDINGUNG 2

VC3: In der VC3 (Formel 12) entspricht die Invariante und nicht Guard der Nachbedingung und VC3 kann bestätigt werden.

$a = b \cdot q + r \wedge 0 \leq r \wedge r \not\geq b$	Invariante und nicht Guard
$\Longrightarrow a = b \cdot q + r \wedge 0 \leq r \wedge r < b$	Nachbedingung

FORMEL 12 INTEGER DIVISION VERIFIKATIONSBEDINGUNG 3

2.8. INTEGER QUADRATWURZEL

```
pre a >= 0;
d := 1;
s := 1;
r := 0;
while s <= a do
    inv d = 2*r + 1 && s = (r+1)*(r+1) && r*r <= a;
    d := d + 2;
    s := s + d;
    r := r + 1;
end
post r*r <= a && a < (r+1)*(r+1)
```

CODE 13 INTEGER QUADRATWURZEL

```
pre a >= 0;
// VC1
// {0 <= a}
// {1 = 1 && 0 <= a}
d := 1;
// {d = 1 && 0 <= a}
// {d = 1 && 1 = 1 && 0 <= a}
s := 1;
// {d = 1 && s = 1 && 0 <= a}
// {d = 2*0 + 1 && s = (0+1)*(0+1) && 0*0 <= a}
r := 0;
// {invar: d = 2*r + 1 && s = (r+1)*(r+1) && r*r <= a}
while s <= a do
    inv d = 2*r + 1 && s = (r+1)*(r+1) && r*r <= a;
    // {invar && guard: d = 2*r + 1 && s = (r+1)*(r+1) && r*r <= a && s <= a}
    // VC2
    // {d + 2 = 2*r + 3 && s + d + 2 = (r+2)*(r+2) && (r+1)*(r+1) <= a}
    d := d + 2;
    // {d = 2*r + 3 && s + d = (r+2)*(r+2) && (r+1)*(r+1) <= a}
    s := s + d;
    // {d = 2*r+3 && s = (r+2)*(r+2) && (r+1)*(r+1) <= a}
    // {d = 2*(r+1) + 1 && s = ((r+1)+1)*((r+1)+1) && (r+1)*(r+1) <= a}
    r := r + 1;
    // {invar: d = 2*r + 1 && s = (r+1)*(r+1) && r*r <= a}
end
// {invar && not guard: d = 2*r + 1 && s = (r+1)*(r+1) && r*r <= a && !(s <=a)}
// VC3
post r*r <= a && a < (r+1)*(r+1)
```

CODE 14 WP BERECHNUNG INTEGER QUADRATWURZEL

Damit die Integer Quadratwurzel valide ist, müssen alle Verifikationsbedingungen (VC1, VC2 und VC3) valide sein. Dies ist bei allen drei der Fall und das Programm ist damit valide.

VC1: In der VC1 (Formel 13) entspricht die Weakest Precondition der Vorbedingung und kann somit bestätigt werden.

$a \geq 0$	Vorbedingung
$\Longrightarrow 1 = 1 \wedge 0 \leq a$	Weakest Precondition

FORMEL 13 INTEGER QUADRATWURZEL VERIFIKATIONSBEDINGUNG 1

VC2: In der VC2 (Formel 14) kann die Weakest Precondition vereinfacht werden. Sie entspricht der Invariante und Guard und kann bestätigt werden.

$d = 2 \cdot r + 1 \wedge s = (r+1) \cdot (r+1) \wedge r \cdot r \leq a \wedge s \leq a$	Invariante und Guard
$\Longrightarrow d + 2 = 2 \cdot r + 3 \wedge s + d + 2 = (r+2) \cdot (r+2) \wedge (r+1) \cdot (r+1) \leq a$	Weakest Precondition
$\equiv d = 2 \cdot r + 1 \wedge s + d + 2 = (r+2) \cdot (r+2) \wedge (r+1) \cdot (r+1) \leq a$	vereinfachen
$\equiv d = 2 \cdot r + 1 \wedge s + 2r + 1 + 2 = r^2 + 4r + 4 \wedge (r+1) \cdot (r+1) \leq a$	d ersetzen und Binom auflösen
$\equiv d = 2 \cdot r + 1 \wedge s = r^2 + 2r + 1 \wedge (r+1) \cdot (r+1) \leq a$	vereinfachen
$\equiv d = 2 \cdot r + 1 \wedge s = (r+1) \cdot (r+1) \wedge (r+1) \cdot (r+1) \leq a$	Binom einsetzen

FORMEL 14 INTEGER QUADRATWURZEL VERIFIKATIONSBEDINGUNG 2

VC3: In der VC3 (Formel 15) kann die Invariante und nicht Guard vereinfacht werden. Sie entspricht der Nachbedingung und kann bestätigt werden.

$d = 2 \cdot r + 1 \wedge s = (r+1) \cdot (r+1) \wedge r \cdot r \leq a \ \wedge s \nleq a$	Invariante und nicht Guard
$\equiv d = 2 \cdot r + 1 \wedge s = (r+1) \cdot (r+1) \wedge r \cdot r \leq a \ \wedge s > a$	vereinfachen
$\equiv d = 2 \cdot r + 1 \wedge s = (r+1) \cdot (r+1) \wedge r \cdot r \leq a \ \wedge (r+1) \cdot (r+1) > a$	s ersetzten
$\Longrightarrow r \cdot r \leq a \wedge \ a < (r+1) \cdot (r+1)$	Nachbedingung

FORMEL 15 INTEGER QUADRATWURZEL VERIFIKATIONSBEDINGUNG 3

2.9. FAKULTÄT

```
pre n >= 0;
i := 0;
a := 1;
while i < n do
    inv 0 <= i && i <= n && a = fact(i);
    i := i + 1;
    a := a * i;
end
post a = fact(n);
```

CODE 15 FAKULTÄT

```
pre n >= 0;
// VC1
// {0 <= n && 1 = fact(0)}
// {0 <= 0 && 0 <= n && 1 = fact(0)}
i := 0;
// {0 <= i && i <= n && 1 = fact(i)}
a := 1;
// {invar: 0 <= i && i <= n && a = fact(i)}
while i < n do
    inv 0 <= i && i <= n && a = fact(i);
    // {invar && guard: 0 <= i && i <= n && a = fact(i) && i < n}
    // VC2
    // {0 <= i + 1 && i + 1 <= n && a * (i + 1) = fact(i + 1)}
    i := i + 1;
    // {0 <= i && i <= n && a * i = fact(i)}
    a := a * i;
    // {invar: 0 <= i && i <= n && a = fact(i)}
end
// {invar && not guard: 0 <= i && i <= n && a = fact(i) && !(i < n)}
// VC3
post a = fact(n);
```

CODE 16 WP BERECHNUNG FAKULTÄT

Damit die Integer Quadratwurzel valide ist, müssen alle Verifikationsbedingungen (VC1, VC2 und VC3) valide sein. Dies ist bei allen drei der Fall und das Programm ist damit valide.

VC1: In der VC1 (Formel 16) entspricht die Weakest Precondition der Vorbedingung und kann somit bestätigt werden.

$n \geq 0$	Vorbedingung
$\Longrightarrow 0 \leq n \wedge 1 = fact(0)$	Weakest Precondition

FORMEL 16 FAKULTÄT VERIFIKATIONSBEDINGUNG 1

VC2: In der VC2 (Formel 17) kann die Weakest Precondition vereinfacht werden. Sie entspricht der Invariante und Guard und kann bestätigt werden.

$0 \leq i \wedge i \leq n \wedge a = fact(i) \wedge i < n$	Invariante und Guard
$\equiv 0 \leq i \wedge i < n \wedge a = fact(i)$	überflüssige Bedingung entfernen
$\Longrightarrow 0 \leq i+1 \wedge i+1 \leq n \wedge a \cdot (i+1) = fact(i+1)$	Weakest Precondition
$\equiv 0 \leq i+1 \wedge i+1 < n+1 \wedge a \cdot (i+1) = fact(i) \cdot (i+1)$	Fakultät anwenden
$\equiv 0 \leq i+1 \wedge i < n \wedge a = fact(i)$	vereinfachen

FORMEL 17 FAKULTÄT VERIFIKATIONSBEDINGUNG 2

VC3: In der VC3 (Formel 18) kann die Invariante und nicht Guard vereinfacht werden. Sie entspricht der Nachbedingung und kann bestätigt werden.

$0 \leq i \wedge i \leq n \wedge a = fact(i) \wedge i \not< n$	Invariante und nicht Guard
$\equiv 0 \leq i \wedge i = n \wedge a = fact(i)$	vereinfachen
$\Longrightarrow a = fact(n)$	Nachbedingung

FORMEL 18 FAKULTÄT VERIFIKATIONSBEDINGUNG 3

2.10. FIBONACCI

```
pre n >= 0;
a := 0;
b := 1;
i := 0;
while i < n do
    inv 0 <= i && i <= n && a = fib(i) && b = fib(i+1);
    h := a;
    a := b;
    b := h + b;
    i := i + 1;
end
post a = fib(n);
```

CODE 17 FIBONACCI

```
pre n >= 0;
// VC1
// {0 <= n && 0 = fib(0) && 1 = fib(1)}
a := 0;
// {0 <= n && a = fib(0) && 1 = fib(1)}
b := 1;
// {0 <= n && a = fib(0) && b = fib(1)}
// {0 <= 0 && 0 <= n && a = fib(0) && b = fib(0+1)}
i := 0;
// {invar: 0 <= i && i <= n && a = fib(i) && b = fib(i+1)}
while i < n do
    inv 0 <= i && i <= n && a = fib(i) && b = fib(i+1);
    // {invar && guard: 0 <= i && i <= n && a = fib(i) && b = fib(i+1) && i < n}
    // VC2
    // {0 <= i+1 && i+1 <= n && b = fib(i+1) && a+b = fib(i+2)}
    h := a;
    // {0 <= i+1 && i+1 <= n && b = fib(i+1) && h+b = fib(i+2)}
    a := b;
    // {0 <= i+1 && i+1 <= n && a = fib(i+1) && h+b = fib(i+2)}
    b := h + b;
    // {0 <= i+1 && i+1 <= n && a = fib(i+1) && b = fib(i+2)}
    // {0 <= i+1 && i+1 <= n && a = fib(i+1) && b = fib(i+1+1)}
    i := i + 1;
    // {invar: 0 <= i && i <= n && a = fib(i) && b = fib(i+1)}
end
// {invar && not guard: 0 <= i && i <= n && a = fib(i) && b = fib(i+1) && !(i < n)}
// VC3
post a = fib(n);
```

CODE 18 WP BERECHNUNG FIBONACCI

Damit die Integer Quadratwurzel valide ist, müssen alle Verifikationsbedingungen (VC1, VC2 und VC3) valide sein. Dies ist bei allen drei der Fall und das Programm ist damit valide.

VC1: In der VC1 (Formel 19) entspricht die Weakest Precondition der Vorbedingung und kann somit bestätigt werden.

$n \geq 0$	Vorbedingung
$\Longrightarrow 0 \leq n \wedge 0 = fib(0) \wedge 1 = fib(1)$	Weakest Precondition

FORMEL 19 FIBONACCI VERIFIKATIONSBEDINGUNG 1

VC2: In der VC2 (Formel 20) kann die Weakest Precondition vereinfacht werden. Sie entspricht der Invariante und Guard und kann bestätigt werden.

$0 \leq i \ \wedge i \leq n \wedge a = fib(i) \wedge b = fib(i+1) \wedge i < n$	Invariante und Guard
$\equiv 0 \leq i \wedge i < n \wedge a = fib(i) \wedge b = fib(i+1)$	Vereinfachen
$\Longrightarrow 0 \leq i+1 \wedge i+1 \leq n \ \wedge b = fib(i+1) \wedge a+b = fib(i+2)$	Weakest Precondition
$\equiv 0 \leq i+1 \wedge i+1 < n+1 \wedge b = fib(i+1) \wedge a+b = fib(i)+fib(i+1)$	Fibonacci anwenden
$\equiv 0 \leq i+1 \wedge i < n \wedge b = fib(i+1) \wedge a+b = fib(i)+b$	Vereinfachen und b einsetzen
$\equiv 0 \leq i+1 \wedge i < n \wedge b = fib(i+1) \wedge a = fib(i)$	Vereinfachen

FORMEL 20 FIBONACCI VERIFIKATIONSBEDINGUNG 2

VC3: In der VC3 (Formel 21) kann die Invariante und nicht Guard vereinfacht werden. Sie entspricht der Nachbedingung und kann bestätigt werden.

$0 \leq i \wedge i \leq n \wedge a = fib(i) \wedge b = fib(i+1) \wedge i \not< n$	Invariante und nicht Guard
$\equiv 0 \leq i \wedge i = n \wedge a = fib(i) \wedge b = fib(i+1)$	Vereinfachen
$\Longrightarrow a = fib(n)$	Nachbedingung

FORMEL 21 FIBONACCI VERIFIKATIONSBEDINGUNG 3

3. Analyse bestehender Tools

Im Rahmen einer Voranalyse wurden bereits verschiedene Tools begutachtet. Damit konnte die Verwendung von diversen Tools aus verschiedenen Gründen verworfen werden.

- In der VerifyThis Competition[2] der ETH Zürich wurden von verschiedenen Teilnehmern Programme formal verifiziert. Dabei wurden den Teams bezüglich Programmiersprache und Verifizierungstechnologie keine Grenzen gesetzt. In der Competition wurden unter anderem Tools wie OpenJML[3] oder VerCors[4] eingesetzt, welche Java Code verifizieren können[5]. VerCors [4] ist spezialisiert auf die Verifikation von parallelen und nebenläufigen Programmen. Diese Spezialisierung führt zu einem signifikanten Mehraufwand, weil Berechtigungen explizit spezifiziert werden müssen. Deshalb wird, falls nur sequenzielle Algorithmen verifiziert werden sollen, auf die Tools OpenJML und KeY[6] verwiesen [5]. Im Rahmen der Programmier-Grundlagen-Module der ersten Studienjahre werden nur sequenzielle Algorithmen verifiziert. Deshalb wird von der Verwendung von VerCors abgesehen. OpenJML und KeY werden in den folgenden Abschnitten behandelt.
- Eine ähnliche Veranstaltung ist die SV-Comp [6]. Diese Competition soll einen Überblick über den aktuellen Stand von verschiedenen Software-Verifizierungs-Tools, anhand derselben Probleme geben. Dabei gibt es auch eine Kategorie für Programme die Java Code verifizieren können.
  Die meisten geprüften Java-Verifizierungs-Tools funktionieren mit symbolischer Ausführung des Codes (Java Ranger[7], JDart[8], COASTAL [7] und Symbolic Pathfinder [8]). Die anderen Tools sind Model Checker welche die Absenz von Benutzerdefinierten Assertion Verletzungen prüfen (JMBC[9] und JHorn[10]). Diese Programme eignen sich für das schnelle Finden von Null-Pointern oder Deadlocks. Sie sind nicht geeignet, um Vor- und Nachbedingungen zu überprüfen.
- Weiter wurden in einer Fallstudie zur formalen Verifizierung von Java Programmen von 2018 [9] verschiedene Verifizierungstools untersucht und bewertet. LOOP und Jack sind veraltete Programme, für welche kein Code mehr zur Verfügung steht. Es wurden zwei Bounded Model Checker (Taco und Sireum/Kiasan) überprüft, welche für dieses Projekt nicht geeignet sind. Ebenfalls wurden die beiden automatischen SMT solver ESC/Java2 und OpenJML untersucht, welche sich potenziell für dieses Projekt eignen würden. ESC/Java2 wird jedoch nicht mehr weiterentwickelt und wurde von OpenJML abgelöst. OpenJML wird in Betracht gezogen und im Kapitel 3.2 weiterverfolgt. Als letztes wurden zwei interaktive Tools (KeY und Krakatoa[11]) bewertet. Diese versuchen den Code automatisch zu verifizieren, ist dies jedoch nicht möglich kann der Benutzer die Verifizierung interaktiv vervollständigen. Für die Verwendung innerhalb der AutoFeedback Webseite ist eine interaktive Verifizierung nicht möglich. Die Ersteller der

[2] VerifyThis: https://www.pm.inf.ethz.ch/research/verifythis.html
[3] OpenJML: https://www.openjml.org/
[4] VerCors: https://vercors.ewi.utwente.nl/publications
[5] VerifyThis Results 2019: https://www.pm.inf.ethz.ch/research/verifythis/Archive/2019.html
[6] KeY: https://www.key-project.org/
[7] Java Ranger: https://github.com/vaibhavbsharma/java-ranger
[8] JDart: https://github.com/psycopaths/jdart
[9] JBMC: https://github.com/diffblue/cbmc/tree/develop/jbmc
[10] JHorn: https://jayhorn.github.io/jayhorn/
[11] Krakatoa: http://krakatoa.lri.fr/

Fallstudie kamen zum Schluss, dass die getesteten Tools im Allgemeinen in einem schlechten Zustand sind.

Als Fazit der Voranalyse konnte festgestellt werden, dass nur das Tool OpenJML vertieft betrachtet werden soll. Die restlichen Tools sind für die geplante Verwendung nicht geeignet. Sie sind entweder schlecht und/oder veraltet dokumentiert, schwierig zu installieren oder komplex in der Anwendung.

An der FHNW wurde von Christoph Stamm bereits mit der Implementierung eines simplen Tools zur Verifizierung von Code begonnen. Dieser WP Calculator wird ebenfalls untersucht.

Vorgehen in der Analysephase:

- Tools die evaluiert werden
  - OpenJML
  - WP Calculator[12] von Christoph Stamm
- Für Erweiterungen oder eine Eigenentwicklung wird möglicherweise ein Java Code Parser benötigt. Dafür wurden zwei mögliche Kandidaten vorevaluiert (werden nur analysiert, falls einer verwendet wird):
  - JavaParser[13]
  - Antlr[14]
- Für Erweiterungen oder eine Eigenentwicklung wird möglicherweise ein automatischer Theorembeweiser benötigt. Dafür wurden vier Kandidaten und ein Interface für die Verwendung dieser Tools voranalysiert. (werden nur analysiert, falls einer verwendet wird):
  - JavaSMT (Interface für Z3, MathSAT, SMTInterpol, Princess usw.) [10]
  - Z3[15]
  - SMTInterpol[16]
  - MathSAT5[17]
  - Princess[18]

[12] WP Calculator: https://gitlab.fhnw.ch/christoph.stamm/wpcalc
[13] JavaParser: https://javaparser.org/
[14] Antlr: https://www.antlr.org/
[15] Z3: https://github.com/Z3Prover/z3
[16] SMTInterpol: https://ultimate.informatik.uni-freiburg.de/smtinterpol/
[17] MathSAT5: https://mathsat.fbk.eu/
[18] Princess: http://www.philipp.ruemmer.org/princess.shtml

3.1. Kriterienliste

Dies sind die Kriterien zur Evaluation von bestehenden Tools zur Überprüfung der mathematischen Korrektheit von Java Programmen. Das Verhalten und die Resultate der bestehenden Tools werden anhand der Verifizierungsbeispiele evaluiert.

Was	Mögliche Punkte
Lizenz	- Ist es möglich dieses Tool gratis zu testen und zu verwenden?
Aktualität	- Für welche Java-Version wurde es entwickelt? - Wann wurde es das letzte Mal verbessert (Bugfixes und neue Funktionen)? - Wird das Tool noch weiterentwickelt oder wurde die Entwicklung eingestellt?
Liegt Code zu diesem Tool vor	- Ist eine Version zum Testen vorhanden? - Kann der Code eingesehen werden (GitHub repository)?
Dokumentation	- Ist eine Dokumentation vorhanden? - Wie einfach kann das Tool mit der Dokumentation verwendet werden?
Einschränkungen	- Was kann das Tool nicht? - Welcher Code kann nicht analysiert werden?
Qualität der Resultate	- Kann das Tool die Verifizierungsbeispiele verifizieren? - Werden zusätzliche Invarianten oder Vorbedingungen benötigt? - Gibt es sinnvolle Fehlermeldungen bei nicht validen Programmen? - Kann die schwächste Vorbedingung (Weakest Precondition) anhand der Nachbedingung und der Invarianten berechnet werden?
Integration	- Wie gross ist der Aufwand der Integration in die bestehende AutoFeedback-Applikation?
Erweiterbarkeit	- Kann das Tool auf die Bedürfnisse des AutoFeedback Projektes angepasst werden?

Tabelle 1 Kriterienliste für die Analyse der bestehenden Tools

3.2. OpenJML

OpenJML ist ein Programm-Verifizierungstool für Java Programme, welches die in der Java Modeling Language (JML) annotierte Spezifikation überprüft. Dabei ist OpenJML selbst kein Theorembeweiser oder Model Checker. Es übersetzt die JML Spezifikation in das SMT-LIB Format und übergibt das Verifizierungsproblem and den SMT Solver. Der SMT Solver selbst ist nicht Teil von OpenJML und es können verschiedene Tools wie Z3, CVC4 oder Yices verwendet werden. In der aktuellen OpenJML Version wird der Z3 v4.3.2 Solver empfohlen und auch mitgeliefert.

Was	Mögliche Punkte
Lizenz	Das Tool ist unter der GPLv2 lizenziert und kann damit gratis eingesetzt werden.
Aktualität	Der letzte OpenJML Release ist vom 12. Juli 2021 V0.8.55. Die Version wird laufend weiterentwickelt und funktioniert nur mit Java 8. Ebenfalls wird momentan an einer Alphaversion von OpenJML entwickelt, welche auf OpenJDK 16 basiert. Der letzte Release OpenJML 0.16.0-alpha-2 ist vom 15. Juli 2021. Dieser funktioniert momentan noch nicht für Windows. Für dieses Projekt wird auf die stabile Version V0.8.55 gesetzt.
Liegt Code zu diesem Tool vor	Das Github repository ist öffentlich zugänglich unter https://github.com/OpenJML/OpenJML/.
Dokumentation	Eine Dokumentation ist vorhanden https://www.openjml.org/documentation/. Sie ist jedoch unübersichtlich und unvollständig.
Einschränkungen	OpenJML kann keine Weakest Precondition berechnen. Es kann nur Code, welcher Java 8 kompatibel ist überprüft werden. Es existiert bereits eine Alphaversion, welche mit Java 16 kompatibel ist, auf welche in Zukunft gesetzt werden könnte. OpenJML kann nur mit einem Java 8 gestartet werden, weil OpenJML den OpenJDK Parser erweitert, damit dieser neben dem Quellcode auch die JML Spezifikation in einen Abstract Syntax Tree parsen kann. OpenJML basiert auf einigen Teilen der OpenJDK Implementation, welche als nicht *public* und mit der Intension diese zu ändern markiert sind. Ein Beispiel dafür ist die Klasse *com.sun.tools.javac.main.Main* welche verwendet wird.
Qualität der Resultate	Mit OpenJML können alle Verifizierungsbeispiele verifiziert werden. Bei einigen mussten kleine Anpassungen vorgenommen werden. Die Resultate der Verifizierung sind in Tabelle 3 unten zu sehen. Fehlermeldungen bei nicht validen Programmen sind vorhanden und es werden auch Gegenbeispiel angegeben. Diese sind nützlich, jedoch schwierig zu verstehen und zu beheben.

Integration	Im Benutzerhandbuch wird angegeben das OpenJML als Kommandozeilenprogramm, als Eclipse Plugin und auch direkt in einem Benutzerprogramm integriert benutzt werden kann. Wie OpenJML in einem Benutzerprogramm verwendet werden kann, wird im Benutzerhandbuch Kapitel 17 beschrieben [11]. Dieses Kapitel ist aber noch nicht vollständig und es wird nicht beschrieben, wie dies funktioniert. Deshalb wird in dieser Arbeit auf das Kommandozeilenprogramm gesetzt. Dieses kann gut in AutoFeedback integriert werden, da keine direkte Integration notwendig ist.
Erweiterbarkeit	Das Tool könnte nur schwer erweitert werden, weil OpenJML komplex und sehr umfangreich ist. Zusätzlich müssten die vorgenommenen Änderungen bei Updates jeweils wieder in die neue Version eingepflegt werden.

TABELLE 2 KRITERIENLISTE OPENJML

Mit OpenJML konnten alle Verifizierungsbeispiele verifiziert werden. Die jeweiligen Ausführungszeiten und Änderungen, welche an den Beispielen vorgenommen wurden, sind in Tabelle 3 zu sehen.

Test	Ausführungsdauer	Bemerkung
Zuweisung	2.4 s	Keine Änderungen nötig.
Swap	2.5 s	Keine Änderungen nötig.
If-Else Anweisung	2.5 s	Die Verifizierung des If-Else Anweisung schlägt ohne zusätzliche Vorbedingungen fehl. Dies liegt an der Behandlung des Primitiven Datentyps *int* in Java. Wenn der minimale Wert für *int* ($-2'147'483'648$) mit minus eins multipliziert wird resultiert daraus wegen eines Überlaufs wieder $-2'147'483'648$. Dieser Fehler kann mit einer zusätzlichen Vorbedingung, x muss grösser als der minimale Wert für *int* sein, behoben werden.
For Schleife	2.6 s	Eine For-Schleife wird in OpenJML wie eine While-Schleife behandelt. Eine While-Schleife benötigt zur Verifizierung eine Invariante. Diese musste in diesem Beispiel noch hinzugefügt werden. `//@ loop_invariant 2*x + 30 == i * (i-1) && i <= 6;`

Multiplikation	2.6 s	Die Multiplikation musste mit einigen zusätzlichen Vorbedingungen erweitert werden. Da das Resultat der Multiplikation immer noch im Wertebereich des Datentype *int* liegen muss.
		`//@ requires Integer.MIN_VALUE <= x * y;`
		`//@ requires Integer.MAX_VALUE >= x * y;`
		Zusätzlich kann es innerhalb der While-Schleife zu einem Variablenüberlauf der Variable b kommen. Nach dem Überlauf wird die Schleife jedoch immer verlassen. Daher kann dieser Überlauf ignoriert werden.
		`//@ assume Integer.MIN_VALUE <= b *2 <= Integer.MAX_VALUE;`

Integer Division	2.5 s	Keine Änderungen nötig.
Integer Quadratwurzel	3.0 s	Keine Änderungen nötig.

Fakultät	11.0 s	Für die Verifizierung der Fakultät wird in der Spezifikation (siehe Kapitel 3) die Fakultät als *fact* verwendet. Diese Funktion gibt es in OpenJML nicht. Sie muss deshalb hinzugefügt werden.
		`/*@ requires n > 0 && n <= 20;`
		`ensures 0 <= \result && \result <= Long.MAX_VALUE;`
		`ensures n > 0 ==> \result == n * spec_factorial(n-1);`
		`also`
		`requires n == 0;`
		`ensures \result == 1;`
		`public model function static pure long spec_factorial(int n) {`
		`if (n == 0) {`
		`return 1;`
		`} else {`
		`assert n * spec_factorial(n-1) <= Long.MAX_VALUE;`
		`return n * spec_factorial(n-1);`
		`}`
		`}@*/`
		Zusätzlich muss das Resultat der Fakultät in den Datentypen *int* passen und kann deshalb nicht grösser als 20 sein. Dies muss als zusätzliche Vorbedingung angegeben werden.

Fibonacci	52.2 s	Für die Verifizierung von Fibonacci wird in der Spezifikation (siehe Kapitel 3) die Fibonacci Funktion als *fib* verwendet. Diese Funktion gibt es in OpenJML nicht. Sie muss deshalb hinzugefügt werden.
		`/*@ requires 46 >= n >= 0;`
		`  ensures 0 <= \result && \result <= Integer.MAX_VALUE;`
		`  ensures n > 1 ==> \result == spec_fib(n-1) + spec_fib(n-2);`
		`also`
		`  requires n == 1;`
		`  ensures \result == 1;`
		`also`
		`  requires n == 0;`
		`  ensures \result == 0;`
		`public model function static pure int spec_fib(int n) {`
		`if (n == 0) {`
		`return 0;`
		`}else if(n == 1){`
		`return 1;`
		`}else {`
		`assume Integer.MIN_VALUE <= spec_fib(n-1) + spec_fib(n-2) <= Integer.MAX_VALUE;`
		`return spec_fib(n-1) + spec_fib(n-2);`
		`}`
		`}@*/`
		Zusätzlich muss das Resultat der Fibonacci in den Datentypen *int* passen und kann deshalb nicht grösser als 46 sein. Dies muss als zusätzliche Vorbedingung angegeben werden.

TABELLE 3 RESULTATE DER VERIFIZIERUNG DER VERIFIZIERUNGSBEISPIELEN MIT OPENJML

Fazit: OpenJML kann den entwickelten Code verifizieren, jedoch nur mit zusätzlichen Bedingungen. Um dies zu ändern kann entweder OpenJML entsprechend angepasst oder ein Wrapper um das Tool herum gebaut werden. Um Anpassungen an OpenJML vornehmen zu können ist ein grundlegendes Verständnis des Tools notwendig, was sehr aufwändig zu erlangen wäre. Zudem müssten bei Updates die vorgenommenen Änderungen jeweils wieder in OpenJML eingepflegt werden. Ebenfalls müsste sichergestellt werden, dass OpenJML trotz dieser Änderungen nach wie vor korrekt funktioniert. Die besser abschätzbare und sicherere Variante ist der Bau eines Wrappers um OpenJML. Dieser kann einfacher auf Versionsänderungen angepasst und auf konkrete Anforderungen dieses Projektes eingehen. Weiter ist auch die Korrektheit von OpenJML immer noch gegeben.

3.3. WP CALCULATOR VON CHRISTOPH STAMM

Der WP Calculator wurde als Testprojekt für die Verifizierung von Code entwickelt.

Was	Mögliche Punkte
Lizenz	Das Tool ist eine Eigenentwicklung von Christoph Stamm. Der Code ist gratis und voll umfänglich zugänglich für dieses Projekt.
Aktualität	Das Tool wird aktuell nicht mehr weiterentwickelt. Die aktuellste Version ist zwei Jahre alt.
Liegt Code zu diesem Tool vor	Der Code liegt im FHNW GitLab vor und kann eingesehen werden https://gitlab.fhnw.ch/christoph.stamm/wpcalc.

Dokumentation	Eine Dokumentation ist nur in Form von JavaDoc Kommentaren vorhanden.
Einschränkungen	Der WP Calculator berechnet, wie es der Name schon sagt, eine Weakest Precondition. Eine Überprüfung, ob diese Weakest Precondition unter Einhaltung der Vorbedingung valide ist, wird nicht vorgenommen. Momentan kann kein Java Code überprüft werden, sondern nur eine vereinfachte Version von Java ähnlicher Syntax.
Resultate Qualität	Das Programm kann keine Beispiele verifizieren. Es kann nur Weakest Precondition für die Verifizierungsbeispiele berechnen. Dies funktioniert jedoch nicht bei allen Verifizierungsbeispielen. Für die Verifizierung des Fibonacci- und des Fakultätbeispiels werden die rekursiven Implementierungen benötigt. Diese können jedoch nicht an den WP Calculator übergeben werden und eine Weakest Precondition Berechnung ist nicht möglich. Für die anderen Verifizierungsbeispiele konnte eine korrekte Weakest Precondition berechnet werden. Diese könnte in einigen Fällen noch logisch weiter vereinfacht werden. Ebenfalls werden keine Fehlermeldungen bzw. Gegenbeispiele angezeigt, weil gar keine Verifizierung erfolgt und damit auch keine Fehler festgestellt werden können. Auch eine Überprüfung auf Variablenüberläufe wird nicht vorgenommen.
Integration	Der WP Calculator kann gut direkt in AutoFeedback integriert werden.
Erweiterbarkeit	Das Tool könnte einfach erweitert werden, weil die Komplexität überschaubar und verständlich ist. Das Tool wurde an der FHNW entwickelt. Eine Zusammenarbeit mit dem Entwickler wäre sehr einfach möglich.

TABELLE 4 KRITERIENLISTE WP CALCULATOR

Fazit: Der WP Calculator kann keine Verifizierung vornehmen. Diese müsste mit Hilfe eines Theorem Provers noch implementiert werden, um den WP Calculator wirklich nutzen zu können. Ebenfalls müsste der WP Calculator angepasst werden damit dieser mit validem Java Code und nicht nur mit einer vereinfachten Grammatik funktioniert. Der Vorteil des WP Calculator ist die Kompaktheit und die Nachvollziehbarkeit des Codes.

3.4. FAZIT DER TOOLANALYSE

OpenJML erfüllt mit einem Wrapper zusammen fast alle gestellten Anforderungen. Es wird einzig keine Weakest Precondition berechnet. Auch in Bezug auf Zeitaufwand und Komplexität ist dies eine realistisch umsetzbare Lösung. Der WP Calculator wäre die flexiblere aber auch die aufwendigere und unsicherere Variante. Deshalb wird die Variante OpenJML mit Wrapper weiterverfolgt und umgesetzt.

3.5. Evaluation eines Java Code Parsers

Für die Implementierung eines Wrappers um OpenJML muss der Programmcode der Studenten analysiert und manipuliert werden können. Dafür eignet sich ein Parser, der den Code zu einem Abstract Syntax Tree (AST) parst, sehr gut. Zwei mögliche Kandidaten wurden vorevaluiert: der JavaParser und Antlr. Beide Tools sind frei zugänglich, werden noch weiterentwickelt und verfügen über die Möglichkeit Code zu parsen und diesen zu manipulieren. Antlr ist das flexiblere Tool. Damit können nicht nur Java Codes, sondern auch andere Grammatiken geparst werden. Der JavaParser wirkt moderner und weniger kompliziert, weil er weniger umfangreiche Funktionen abdeckt. Der Code kann einfach eingelesen, manipuliert und wieder ausgegeben werden. Es ist ebenfalls einfach möglich neuen Code automatisch zu erstellen, welcher anschliessend ausgeführt werden kann. Das Benutzen des JavaParsers wurde durch das Einlesen in das Buch JavaParser: Visited, welches auf der Webseite zur Verfügung gestellt wird, einfach ermöglicht [12]. Weil der JavaParser alle Anforderungen erfüllt, wurde auf eine genauere Analyse von Antlr verzichtet und der JavaParser verwendet.

3.6. Evaluation eines automatischen Theorembeweisers

Bei der Analyse der Tools wurde OpenJML als beste Variante evaluiert und in dieser Arbeit verwendet. In der aktuellen OpenJML Version wird der Z3 v4.3.2 Solver empfohlen und auch mitgeliefert. Deshalb wurde auf eine Evaluation von anderen Theorembeweisern verzichtet und der Z3 Theorembeweiser innerhalb von OpenJML verwendet.

4. VERIFIZIERUNGS-TOOL

In diesem Kapitel wird in mehreren Schritten erklärt, wie die mathematische Korrektheit einer Studentenlösung überprüft wird. Dies bildet die Basis für die Umsetzung des Verifizierungs-Tools. Der Aufbau des Verifizierungs-Tools ist in Abbildung 1 zu sehen. Die einzelnen Schritte werden in den folgenden Kapiteln genauer beschrieben. Als erstes wird in Kapitel 4.1 erklärt wie die Studentenprogrammen mit Hilfe der Java Modeling Language beschrieben werden können. Im nächsten Teil (Kapitel 4.2) wird auf die Umsetzung der Verifizierungs-Tools selbst eingegangen und zum Schluss werden die Testresultate des Verifizierungs-Tools präsentiert.

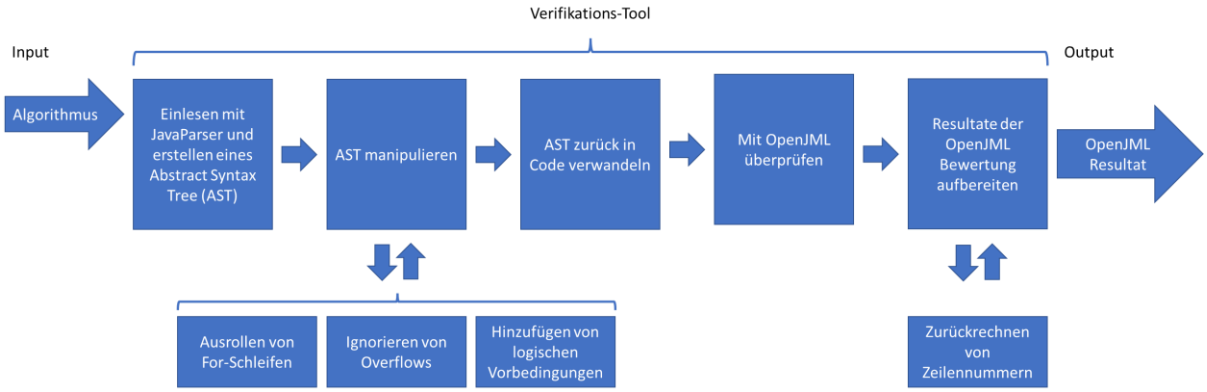


ABBILDUNG 1 AUFBAU VERIFIZIERUNGS-TOOL

Der implementierte Code zum Verifizierungs-Tool befindet sich im Projekt AutoFeedback Library[19]. Die Klassen des Tools befinden sich im Package *ch.fhnw.autofeedback.verificationevaluation* und die Tests im Package *ch.fhnw.autofeedback.verification*.

4.1. JAVA MODELING LANGUAGE

Die Java Modeling Language (JML) ist eine Spezifikationssprache für Java Programme, welche dem Design by contract Paradigma folgt. Für die Spezifikation werden Vor- und Nachbedingungen und Invarianten im Hoare-Style verwendet. Die Spezifikation wird mittels Kommentarannotationen direkt in die Java Quelldatei geschrieben. Die Java Kommentare welche mit einem @ beginnen werden als JML Annotationen interpretiert. Die beiden möglichen Annotationsvarianten sind in Code 19 zu sehen. Dabei wurde das @ Symbol etwas unglücklich gewählt, weil dieses mittlerweile auch für Java Annotationen verwendet werden kann. Dies bedeutet, dass auskommentierte Java Annotationen als JML Spezifikation interpretiert werden, was zu Fehler führen kann [13].

```
//@ <JML spezifikation>
/*@ <JML spezifikation> */
```

CODE 19 JML SPEZIFIKATIONSBEISPIELE

JML Syntax stellt unter anderem die Schlüsselbegriffe in Tabelle 5 zur Verfügung. Eine JML Anweisung besteht aus einem Schlüsselbegriff und einem logischen Ausdruck. Die Anweisung wird mit einem Semikolon abgeschlossen. Die logischen Ausdrücke können mit den JML Ausdrücken aus Tabelle 6 ergänzt werden. Ein Beispiel dafür ist in Code 20 zu sehen. Die folgende Methode liefert ein Resultat über zehn, wenn der Input x kleiner gleich zehn ist und b unter null.

[19] AutoFeedback Library: https://gitlab.fhnw.ch/autofeedback/autofeedback-library

JML Schlüsselbegriff	Bedeutung
requires	Definiert eine Vorbedingung für die folgende Methode.
ensures	Definiert eine Nachbedingung für die folgende Methode.
invariant	Definiert eine Invariante für die Klasse.
loop_invariant	Definiert eine Invariante für eine Schleife.
also	Kombiniert mehrere Spezifikationsfälle.
assert	Definiert eine JML Assertion.
assume	Teilt OpenJML mit, dass eine Aussage als wahr angesehen werden soll und nicht überprüft werden muss. Diese Aussage kann von OpenJML für spätere Überprüfungen genutzt werden.
pure	Definiert, dass eine Methode Seiten Effekt frei ist. Ebenfalls wird erwartet, dass eine pure Methode immer terminiert oder eine Exception wirft.
assignable	Definiert welche Felder dieser Methode verändert werden dürfen.
model	Mit dem Begriff Model können Felder, Methoden oder Typen definiert werden. Alle mit Model definierten Features sind nur in der Spezifikation vorhanden.

TABELLE 5 JML SCHLÜSSELBEGRIFFE

JML Ausdruck	Bedeutung
\result	Ein Identifikator für den Rückgabewert der folgenden Methode.
\old(<expression>)	Ein Identifikator, welcher für den Wert der *<expression>* zum Eintrittszeitpunkt in die Methode steht.
$a \Longrightarrow b$	a impliziert b.
$a \Longleftarrow b$	a wird von b impliziert.
Java Syntax	Es kann die Standard Java Syntax für logische Ausdrücke wie «und», «oder» oder «nicht», für mathematische Ausdrücke wie «Addition», «Multiplikation», «Division» oder «Subtraktion» und für Vergleichsoperatoren wie «grösser» oder «kleiner gleich» verwendet werden.

TABELLE 6 JML AUSDRÜCKE

```
//@ requires x >= 10 && b < 0;
//@ ensures \result >= 10;
```

CODE 20 JML BEISPIEL

In der Spezifikation des Fibonacci und des Fakultätsbeispiels wurden die Definition der Fakultät bzw. von Fibonacci verwendet. Diese sind beide in OpenJML nicht vorhanden. Sie können mittels einer Model

Funktion eingefügt werden. Dies ist eine Funktion, die nur in der JML Spezifikation vorhanden ist und benutzt werden kann, um die Java Implementierung zu validieren. Im Code 21 ist die Implementierung einer solchen Model Funktion für die Fakultät zu sehen. Es wird eine rekursive Variante der Fakultät mit Vor- und Nachbedingungen implementiert. Der Schlüsselbegriff *spec_factorial* kann nun für die Verifizierung der Java Methode verwendet werden.

```
/*@  requires n > 0 && n <= 20;
       ensures 0 <= \result && \result <= Long.MAX_VALUE;
       ensures n > 0 ==> \result == n * spec_factorial(n-1);
     also
       requires n == 0;
       ensures \result == 1;
  public model function static pure long spec_factorial(int n) {
    if (n == 0) {
      return 1;
    } else {
      assert n * spec_factorial(n-1) <= Long.MAX_VALUE;
      return n * spec_factorial(n-1);
    }
  }@*/
```

CODE 21 BEISPIEL MODEL FUNKTION DER FAKULTÄT

4.2. WRAPPER

Mit OpenJML können alle Verifizierungsbeispiele verifiziert werden. In der Evaluation von OpenJML in Kapitel 3.2 wurde gezeigt, dass für die Verifizierungsbeispiele zusätzliche Bedingungen benötigt werden. Die Ergebnisse der Validierung und die zusätzlichen Bedingungen können in Tabelle 3 betrachtet werden. Ebenfalls wurde ein Wrapper um OpenJML als beste Variante um dies zu ändern bestimmt.

Der Wrapper um OpenJML ist wie folgt aufgebaut:

1. Einlesen des Studentencodes mithilfe eines Parsers und Erstellen eines Abstract Syntax Tree (AST)
2. Manipulation des ASTs
3. Den manipulierten AST wieder zurück in validen Java Code verwandeln (Pritty printing).
4. Den manipulierten Code an OpenJML zur Bewertung übergeben.
5. Die Resultate der OpenJML Bewertung aufbereiten und für die Anzeige speichern.

Die einzelnen Schritte des Wrappers werden in den folgenden Kapiteln erklärt. Die Implementierungen der einzelnen Schritte befinden sich im Package *ch.fhnw.autofeedback.verificationevaluation* des Projektes AutoFeedback Library.

4.2.1. EINLESEN DES STUDENTENCODES

Der Studentencode wird mit dem JavaParser eingelesen und der Abstract Syntax Tree des Codes wird vom Parser erstellt.

4.2.2. MANIPULATION DES ASTS

Der eingelesene AST kann nun manipuliert werden und die nötigen Änderungen für OpenJML können vorgenommen werden. Es werden drei Bereiche des Codes angepasst: Es werden fehlende OpenJML Vorbedingungen hinzugefügt, unnötige Overflow-Warnungen ignoriert und einfache For-Schleifen ausgerollt.

Für jede Manipulation des ASTs wurden Beispiele und Tests geschrieben. Sowohl die Tests als auch die Implementationen befinden sich im Testordner. Die Beispiele sind im Package *ch.fhnw.autofeedback.verificationExamples* zu finden. Die Klassen beginnen mit Examples. Die dazugehörigen Tests befinden sich

im Package *ch.fhnw.autofeedback.verifiaction*. Alle Methoden sind in der AutoFeedback Library zu finden.

4.2.2.1. Ausrollen von For-Schleifen

Eine For-Schleife wird in OpenJML wie eine While-Schleife behandelt. Eine While-Schleife benötigt für die Verifizierung eine Invariante. Bei simplen For-Schleifen ist dies für die Verifizierung nicht nötig. Sie können ausgrollt und damit eliminiert werden. Somit wird keine Schleifen-Invariante benötigt, weil keine Schleife mehr vorhanden ist. Ein Beispiel für eine einfache For-Schleife und die ausgerollte Version davon ist im Code 22 und Code 23 zu sehen.

```
for (int i = 1; i < 3; i++)
    x += i;
```

Code 22 Einfache For-Schleife

```
{   int i = 1;
    x += i;
    i++;
    x += i;
    i++;
    x += i;
    i++;
}
```

Code 23 Ausgerollte einfache For-Schleife

```
For (Initialisierung; Abbruchbedingung; Veraenderung Laufvariable)
    //Anweisungen
```

Code 24 Aufbau einer For-Schleife

Wie im Beispiel zu sehen ist, wird die Schleife durch einen Block ersetzt, der mit der Initialisierung der Schleife beginnt. Danach folgen die Schleifenanweisungen und das Verändern der Laufvariablen. Diese werden so oft eingefügt, wie die Schleife ausgeführt wird. Der Code für den Ersatzblock wird durch die Ausführung der For-Schleife generiert. Der Inhalt der Schleife muss dafür so verändert werden, dass er nicht ausgeführt, sondern auf die Konsole geschrieben wird. Am Ende der Schleife soll auch die Veränderung der Laufvariable auf die Konsole geschrieben werden. Damit im Ersatzblock auch die Initialisierung vorhanden ist, wird diese bereits vor der Schleife auf die Konsole geschrieben. Für die einfache For-Schleife in Code 22 ist die Generierung der ausgerollten For-Schleife im Code 25 zu sehen. Die einfache Schleife kann nun durch den Konsolenoutput ersetzt werden.

```
System.out.println("int i = 1;");
for (int i = 1; i < 3; i++) {
    System.out.println("x += 1;");
    System.out.println("i++;");
}
```

Code 25 Generierung der ausgerollten For-Schleife

Mit dieser Methode können For-Schleifen ausgerollt werden, bei welchen in der Abbruchbedingung und in der Laufvariablenveränderung nur Variablen benutzt werden, welche in der Initialisierung initialisiert werden. Zusätzlich funktioniert der Ersatzblock nur, wenn in der Schleife kein *continue* oder *break* Statement verwendet wird. Dies weil die beiden Anweisungen nicht innerhalb eines Blockes verwendet werden können. Falls beim Ausrollen ein Fehler auftritt, wird die originale For-Schleife im Code belassen.

Bei Schleifen mit vielen Anweisungen oder mit vielen Iterationen wird der Ersatzblock sehr lange. Dies kann bei der Verifizierung mit OpenJML zu extrem langen Ausführungszeiten führen. Deshalb wurde ein konfigurierbares Maximum an Codezeilen für Ersatzblöcke festgelegt, welches standartmässig bei 1000

Zeilen liegt. Falls diese Anzahl an Zeilen überschritten wird, wird die For-Schleife nicht ersetzt. Die Ausführungsdauer für das Ausrollen und die Verifizierung von For-Schleifen mit verschiedenen Längen ist in Tabelle 7 zu sehen. Ausgerollt wurde die For-Schleife im Code 22 mit angepasster Abbruchbedingung, um die gewünschte Anzahl an Iterationen zu erhalten. Der Zeitaufwand für die Verifizierung kann bei anderem Schleifeninhalt anders ausfallen. Die Zahl von standartmässig 1000 Zeilen kam durch Experimentieren zustande, weil der Aufwand für die Verifizierung einer Code Zeile nicht oder nur schwer allgemeingültig berechenbar ist.

Länge der Ausgerollten For-Schleife	Schleifen-Iterationen	Verifizierungszeit
200 Zeilen	100 Iterationen	14s
600 Zeilen	300 Iterationen	25s
1000 Zeilen	500 Iterationen	49 s
1600 Zeilen	800 Iterationen	1m 46s
2000 Zeilen	1000 Iterationen	3m 29s

TABELLE 7 VERIFIZIERUNGSDAUER EINER AUSGEROLLTEN FOR-SCHLEIFE

4.2.2.2. OVERFLOWS IGNORIEREN

OpenJML überprüft den Code automatisch auf Overflows. Das ist erwünscht und in fast allen Fällen sinnvoll. Es gibt jedoch Ausnahmen, in denen ein Overflow keinen Einfluss auf das Resultat der Methode hat. Dies ist beim Multiplikationsbeispiel (Kapitel 2.6) der Fall. Wenn eins mit $2'147'483'647$ multipliziert werden soll, kommt es in der Variablen b zu einem Overflow. Das ist nicht mehr relevant, weil die Schleife nach dem Overflow immer verlassen wird. Der Verlauf der einzelnen Variablen kann in Tabelle 8 betrachtet werden.

Position in der Methode	a	b	r
Werte vor dem While-Loop	1	2'147'483'647	0
Werte am Ende des ersten Durchgangs des While-Loops	0	-2	2'147'483'647
Werte vor dem Return	0	-2	2'147'483'647

TABELLE 8 MULTIPLIKATION VARIABELVERÄNDERUNGEN

OpenJML stellt in diesem Beispiel richtigerweise einen Overflow fest und kann damit die Methode nicht verifizieren. Diese Overflow-Warnung kann ignoriert werden, sofern der OpenJML Spezifikationsbefehl in Code 26 vor der Zeile, in welcher der Overflow passiert, geschrieben wird. Diesen Befehl davor hinzuschreiben ist fehleranfällig. Der Wrapper kann diese Zeile automatisch generieren und vor eine Codezeile schreiben.

```
/*@ assume Integer.MIN_VALUE <= b * 2 <= Integer.MAX_VALUE;*/
```

CODE 26 OVERFLOW IN MULTIPLIKATION IGNORIEREN

Dafür muss neu nur noch ein Kommentar, welcher mit einem *$*-Zeichen beginnt hinter die betroffene Codezeile geschrieben werden. Das Dollarzeichen muss zwingend ohne Abstand zu Beginn des Kommentars stehen. Mit dieser Annotation werden sämtliche Overflows ignoriert, welche von dieser Zeile produziert werden. Dies bedeutet nicht, dass die Methode dadurch immer verifiziert werden kann. Falls der Overflow einen Einfluss auf das Resultat der Methode hat, wird nun nicht mehr der Overflow bemängelt, sondern z.B., dass die Nachbedingungen nicht überprüft werden können. Ein Beispiel dafür ist die Methode mit dem dazugehörigen Feedback 1, welche in Code 27 zu sehen ist.

Das Ignorieren von Overflows funktioniert für die ganzzahligen Datentypen *byte*, *short*, *char*, *int* und *long*.

```
/*@ ensures \result == 5;*/
public int ignoreInfeasible(int x) {
    /*@ assume Integer.MIN_VALUE <= x + Long.MAX_VALUE <= Integer.MAX_VALUE;*/
    x += Long.MAX_VALUE;     //$ ignore Overflow
    return 5;
}
```

CODE 27 BEISPIEL METHODE IGNOREINFEASIBLE

```
ch.fhnw.autofeedback.verificationExamples.Test.ignoreInfeasible(int)
Attention the line numbers are calculated and could be wrong!
 Feasibility check #1 - end of preconditions   OK [0.00 secs]
 Feasibility check #2 - after explicit assume statement   infeasible [0.00 secs]
results\Test.java:6: warning There is no feasible path to program point after explicit assume statement in method ch.fhnw.autofeedback.verificationExamples.Test.ignoreInfeasible(int)
        /*@ assume Integer.MIN_VALUE <= x + Long.MAX_VALUE <= Integer.MAX_VALUE;*/
            ^
results\Test.java:6: Feasibility check #3 - at program exit  infeasible [0.00 secs]
results\Test.java:6: warning There is no feasible path to program point at program exit in method ch.fhnw.autofeedback.verificationExamples.Test.ignoreInfeasible(int)
    public int ignoreInfeasible(int x) {
               ^
```

FEEDBACK 1 OPENJML FEEDBACK FÜR DIE METHODE IGNOREINFEASIBLE

4.2.2.3. HINZUFÜGEN VON LOGISCHEN VORBEDINGUNGEN

Eine Schwierigkeit der Programmverifikation besteht in den Eigenschaften von elementaren Operationen, wie der Addition oder der Multiplikation von ganzen Zahlen. Diese Operationen haben in der Mathematik verschiedene Grundregeln und Eigenschaften. Es wird von einer unendlichen Menge an Ganzzahlen ausgegangen. Die Menge der Ganzzahlen ist in Computern jedoch nicht unendlich. Korrekt verifizierte Methoden können deshalb nur innerhalb der Limiten der jeweiligen Rückgabedatentypen funktionieren. Zum Beispiel kann die Multiplikation (Kapitel 2.6) nur Zahlen korrekt multiplizieren bei welchen das Resultat zwischen dem minimalen und dem maximalen Wert vom Rückgabetyp Integer liegt. Diese Vorbedingung ist für OpenJML nicht automatisch vorhanden. Sie muss manuell von den Studenten bzw. jetzt automatisch vom Wrapper hinzugefügt werden. Im Code 28 ist die Spezifikation der Multiplikation zu sehen.

```
//@ requires x >= 0;
//@ ensures \result == x * y;
static int multiply(int x, int y) {  …  }
```

CODE 28 ORIGINALE SPEZIFIKATION DER MULTIPLIKATION

Diese wird vom Wrapper eingelesen, die zusätzliche Vorbedingung wird generiert und zusammen werden sie wieder abgespeichert. Für die Generierung der neuen Vorbedingung wird sowohl die Nachbedingung als auch der Rückgabetyp der Methode eingelesen. Daraus wird eine neue Vorbedingung generiert, welche sicherstellt, dass die Resultate innerhalb der Limiten des Datentyps sind. Im Multiplikationsbeispiel würde dies bedeuten, dass das Resultat von $x \cdot y$ innerhalb vom Typ Integer liegt. Die neue Spezifikation ist im Code 29 zu sehen. Das Hinzufügen von logischen Vorbedingungen funktioniert für die ganzzahligen Datentypen *byte*, *short*, *char*, *int* und *long*.

```
//@ requires x >= 0;
//@ ensures \result == x * y; requires Integer.MIN_VALUE <= x * y <= Integer.MAX_VALUE;
static int multiply(int x, int y) {  …  }
```

CODE 29 MULTIPLIKATIONSSPEZIFIKATION NACH DEM HINZUFÜGEN VON LOGISCHEN VORBEDINGUNGEN

4.2.3. PRITTY PRINTING

Mit dem JavaParser kann Code als AST eingelesen, verändert und wieder zurück zu Code verwandelt werden. Der erhaltene Code wird, laut JavaParser erneut nach Java Konventionen gespeichert. Dies bedeutet, dass er wieder fast gleich aussehen sollte wie der Originalcode, falls dieser ebenfalls nach Java Konventionen formatiert wurde [12].

4.2.4. OPENJML BEWERTUNG

Der optimierte Studentencode kann jetzt an OpenJML zur Bewertung übergeben werden. Für die Ausführung von OpenJML sind verschiedene Parameter nötig. Bei der Ausführung der Konsolenapplikation muss die Jar Datei von OpenJML angegeben werden. Für diese gibt es verschiedene Parameter, die gesetzt werden. Sie werden sowohl für die Verifizierung als auch für die Generierung des Feedbacks benötigt. Für die Verifizierung muss der Statische Checker mit dem Parameter *-esc* aktiviert werden. Für die Feedback Generierung wird der Parameter *-verboseness 2* verwendet, damit die Resultate für alle geprüften Methoden ausgegeben werden und nicht nur die Fehler. Für die Analyse der Fehler wird die Verfolgung der Gegenbeispiele für alle fehlgeschlagenen *asserts* mit dem Parameter *-subexpressions* eingeschaltet. Ebenfalls wird mit dem Parameter *-checkFeasibility all* angegeben, dass alle Feasibility checks durchgeführt werden sollen. Dies ergibt für die Bewertung einer Klasse *StudentFile* den Befehl in Code 30.

```
java -jar openjml.jar -esc -verboseness 2 -subexpressions -checkFeasibility all StudentFile.java
```

CODE 30 OPENJML BEFEHL

Ein wichtiger Punkt für die Bewertung ist, dass OpenJML nur mit Java 8 Version funktioniert, nicht aber mit neueren oder älteren Java Versionen. Deshalb muss der *java* Befehl auf eine Java 8 Implementierung zeigen (siehe Kapitel 3.2).

4.2.5. OPENJML OUTPUT ZU FEEDBACK UMWANDELN

Die Konsolenausgabe vom OpenJML wird automatisch in ein Feedback umgewandelt, welches auf der Webseite angezeigt werden kann. Dargestellt wird der Name jeder Methode und welche Prüfungen durchgeführt wurden. Für das Multiplikationsbeispiel (Kapitel 2.6) ergibt dies das Feedback, welches im Feedback 2 zu sehen ist. Es wurden dabei zwei Methoden überprüft, einerseits der Standardkonstruktor und andererseits die *multiply* Methode. Bei beiden Methoden waren alle Checks gültig und damit ist auch die Klasse verifiziert.

```
ch.fhnw.autofeedback.verificationExamples.Multiply.Multiply()
 Feasibility check #1 - end of preconditions   OK [0.00 secs]
 Feasibility check #2 - at program exit   OK [0.00 secs]

ch.fhnw.autofeedback.verificationExamples.Multiply.multiply(int,int)
 Feasibility check #1 - end of preconditions   OK [0.00 secs]
 Feasibility check #2 - after explicit assume statement   OK [0.02 secs]
 Feasibility check #3 - at program exit   OK [0.01 secs]
```

FEEDBACK 2 MULTIPLIKATIONSFEEDBACK

Falls Bei der Überprüfung mit OpenJML ein Test fehlschlägt, wird ein entsprechendes Feedback gegeben. Im Beispiel Feedback 3 ist die Fehlermeldung zu sehen, welche an einen Studenten weitergeleitet würde, wenn ein If-Else Beispiel aus Kapitel 2.4 überprüft würde. Im ersten Teil des Feedbacks wird darauf hingewiesen, dass es in Zeile 10 zu einem Overflow kommen kann. Im Zweiten Teil wird ein Beispiel präsentiert in welchem dieser Overflow auftreten würde. Dies ist der Fall, wenn die Vorbedingung erfüllt ist, die If Bedingung wahr ist und der x Wert von $-2'147'483'648$ mit minus eins multipliziert wird. Dies ergibt in Java, wegen eines Overflows, immer noch $-2'147'483'648$ und damit erfüllt die Methode die Spezifikation nicht und ist nicht valide.

```
ch.fhnw.autofeedback.verificationExamples.IfElse.ifelse(int)
Attention the line numbers are calculated and could be wrong!
results\IfElse.java:10: warning The prover cannot establish an assertion (Arithmet-
icOperationRange) in method ifelse  int multiply overflow
            x = x * -1;
                  ^

results\IfElse.java:5:    requires (x > 1 || x < 0);
                    VALUE x       === ( - 2147483648 )
                    VALUE 1       === 1
                    VALUE x > 1   === false
                    VALUE x       === ( - 2147483648 )
                    VALUE 0       === 0
                    VALUE x < 0   === true
                    VALUE x > 1 || x < 0       === true
                    VALUE (x > 1 || x < 0)     === true
results\IfElse.java:1:    if (x < 0) ...
                    VALUE x       === ( - 2147483648 )
                    VALUE 0       === 0
                    VALUE x < 0   === true
                    VALUE (x < 0)       === true
                          Condition = true
results\IfElse.java:10:   x = x * -1
                    VALUE x       === ( - 2147483648 )
                    VALUE -1      === ( - 1 )
                    VALUE x * -1  === ( - 2147483648 )
                    VALUE x = x * -1    === 0
results\IfElse.java:10:   ArithmeticOperationRange assertion |#mul32ok#|(x, -1)
                    VALUE -1      === ( - 1 )
                    VALUE |#mul32ok#|(x_165_165___10, -1)   === false
results\IfElse.java:10:  Invalid assertion (ArithmeticOperationRange)
```

FEEDBACK 3 IF-ELSE-FEEDBACK

Im Feedback 3 ist zu sehen, dass angezeigt wird auf welcher Zeile der Fehler zu finden ist. Dies entspricht in den meisten Fällen der korrekten Zeile in der Datei, welche überprüft wurde. Im Beispiel Feedback 3 steht als Gegenbeispiel, dass die If-Anweisung auf Zeile 1 zu finden sei, diese falsche Angabe kommt von OpenJML. Alle anderen Zeilennummern im Beispiel sind korrekt. Beinhaltet die Datei nun aber eine Schleife, welche ausgerollt wird oder sonstige Optimierungen, stimmt die Zeilennummer nicht mehr mit der Originaldatei des Studenten überein. Um die Fehlersuche der Studenten zu erleichtern wird versucht die Zeilennummern zurück auf die Zeile im Originalcode zu rechnen. Dies geschieht in dem der Wrapper sich merkt, welche Zeilen an welcher Stelle im Programmcode verändert oder hinzufügt wurden. Mit dieser Information kann die Zeilennummer im OpenJML Resultat auf die Originalzeile zurückgerechnet werden. Weil auch das Pritty Printing einen Einfluss auf die Zeilennummer haben kann oder es einen Fehler in einer zusätzlich hinzugefügten Zeile geben kann, ist dies nicht immer exakt. Falls die Datei nach Java Konventionen formatiert ist, kommt es zu keiner Abweichung wegen des Pritty Printing. Falls die Zeile in der Originaldatei gar nicht vorhanden ist, wird das Rückrechnen in keinem Fall korrekt sein.

4.3. Resultate

Für die Überprüfung des Verifizierungs-Tools wurden wieder die Verifizierungsbeispiele, die spezifischen Tests für die drei Manipulation des ASTs (siehe Kapitel 4.2.2) und ein Test für die Validierung von Teilen eines Programms verwendet. Die Resultate der Überprüfung der Verifizierungsbeispiele mit dem Wrapper sind in Tabelle 9 zu sehen. Die Unit Tests der Verifizierungsbeispiele sind in der Klasse *TestVerification* im Package *ch.fhnw.autofeedback.verification* zu finden. Dabei konnten alle Beispiele korrekt verifiziert werden. Die Unit Tests für die Überprüfung der AST Manipulationen sind im selben Package zu finden und alle Methoden konnten, wie erwartet, verifiziert oder widerlegt werden.

Ein Ziel des Wrappers war, dass die Verifizierungsbeispiele ohne zusätzliche Bedingungen verifiziert werden können. Fast alle Beispiele können ohne zusätzliche Bedingungen verifiziert werden. Auf die drei Ausnahmen wird in der Bemerkungsspalte der Tabelle 9 eingegangen.

Mit *assume* und *assert* Statements und Schleifen-Invarianten können auch Teile einer Methode überprüft werden ohne, dass für die Methode selbst Vor- oder Nachbedingungen spezifiziert wurden. Um dies zu überprüfen, wurden für das Zuweisungsbeispiel, das Multiplikationsbeispiel und das Swap-Beispiel die Vor- und Nachbedingungen zu *assume* und *assert* Statements in der Methode umgewandelt. Die Implementation der drei Beispiele ist in der Klasse *ValidatePartOfMethods* im Package *ch.fhnw.autofeedback.verificationExamples* zu finden und der dazugehörige Unit Test in der Klasse *TestValidatePartOfMethods* im Package *ch.fhnw.autofeedback.verification*. Grundsätzlich wird in diesen Beispielen immer noch die gesamte Methode überprüft. Weil keine Vor- und Nachbedingungen vorhanden sind, werden diese in jedem Fall eingehalten. Falls in einem Teil der Methode ein Overflow auftritt, wird dieser als Fehler gemeldet.

Test	Ausführungsdauer der Unittests	Bemerkung
Zuweisung	4.6 s	Keine Änderungen nötig.
Swap	4.8 s	Keine Änderungen nötig.

If-Else Anweisung	4.7 s	Es existiert immer noch derselbe Fehler wie ohne Wrapper (siehe Kapitel 3.2). Die genaue Fehlermeldung kann in Feedback 3 angesehen werden. Dieser Variablenüberlauf wird als Programmierfehler eingestuft. Um diesen zu beheben, müsste die Implementierung oder die Spezifikation angepasst werden.
For-Schleife	5.2 s	Keine Änderungen nötig.
Multiplikation	4.7 s	Keine Änderungen nötig.
Integer Division	4.8 s	Keine Änderungen nötig.
Integer Quadratwurzel	4.7 s	Keine Änderungen nötig.
Fakultät	13.5 s	Für die Fakultät wird immer noch die Spezifikation von *fact* zur Verifizierung benötigt (siehe Kapitel 3.2). Diese kennt weder OpenJML noch der Wrapper automatisch. Damit die Spezifikation von *fact* korrekt genutzt werden kann, muss ebenfalls die Input Variable eingeschränkt werden. Diese hängt mit der Spezifikation von *fact* zusammen und kann nicht automatisch vom Wrapper eingefügt werden.
Fibonacci	103 s	Für das Fibonacci Beispiel gilt dasselbe wie für die Fakultät, einfach bezüglich der *fib* und nicht der *fact* Methode.

TABELLE 9 RESULTATE DER VERIFIZIERUNG DER VERIFIZIERUNGSBEISPIELEN MIT DEM WRAPPER

Fazit: Mit dem Verifizierungs-Tool können Studentenprogramme mittels Vor- und Nachbedingungen, unter Einhaltung von Invarianten, überprüft werden. Dabei werden auch Variablenüberläufe als Fehler erkannt und dem Benutzer gemeldet. Für einfache Beispiele werden keine zusätzlichen Bedingungen für die Verifizierung benötigt. Bei den beiden schwierigen Beispielen (Fibonacci und Fakultät) könnten die *spec* Model Methoden vom Dozenten zur Verfügung gestellt werden. Somit müsste der Student diese nur noch einfügen und korrekt verwenden. Den Studenten werden im Fehlerfall Gegenbeispiele als Feedback geliefert, die bei der Korrektur des Codes oder der Spezifikation sehr nützlich sein könnten. Wie ein solches Feedback mit Gegenbeispiel bei einem Variablenüberlauf aussieht, ist in Feedback 3 zu sehen. Mit *assume* und *assert* Statements und Schleifen-Invarianten können auch Teile einer Methode überprüft werden ohne, dass für die Methode selbst Vor- oder Nachbedingungen spezifiziert wurden. Momentan funktioniert OpenJML erst für Java 8 Codebeispiele und nicht für neuere Java Versionen. Ebenfalls kann mit dem Verifizierungs-Tool keine Weakest Precondition berechnet werden.

5. Integration in AutoFeedback

Um die Überprüfung der Korrektheit für Studenten nutzbar zu machen, soll diese ins Lehrfonds-Projekt «Automatisches Feedback zu Programmieraufgaben» integriert werden. Die Funktionsweise des bestehenden AutoFeedback Projekts wird im Kapitel 1.1 beschrieben. Der Code des AutoFeedback Projekts wurde in verschiedene Projekte unterteilt. In diesem Kapitel wird auf die Änderungen an den bestehenden Projekten und auf deren Zusammenspiel eingegangen. Wie die Projekte zusammenhängen, ist in Abbildung 2 dargestellt. Jedes der grossen viereckigen Kästchen stellt dabei ein Projekt dar. Die kleinen viereckigen Kästchen zeigen, das ein Projekt von einem anderen Projekt abhängig ist.

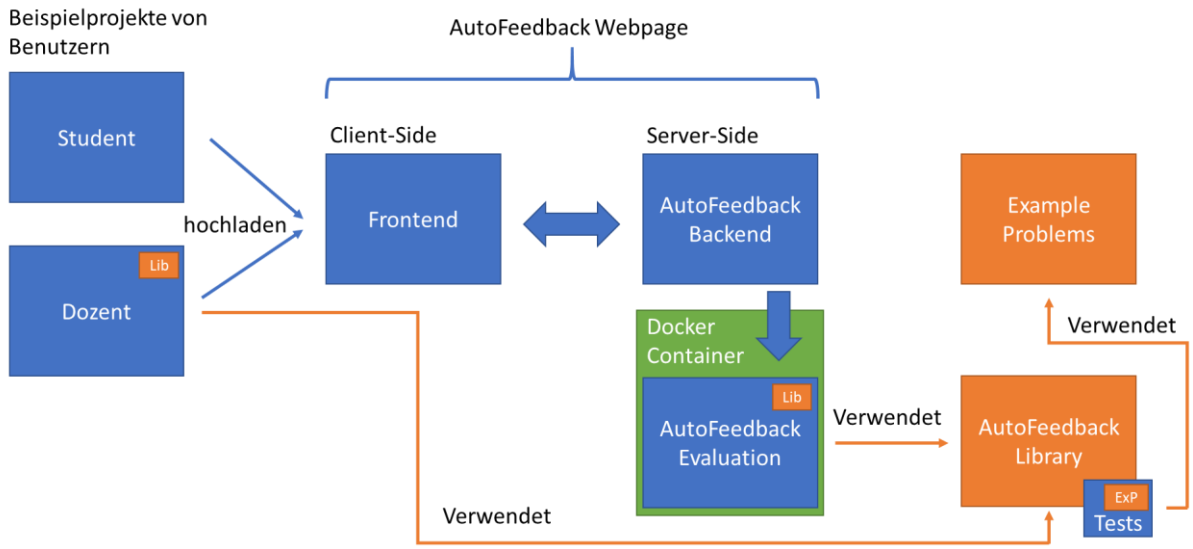


Abbildung 2: Übersicht über die AutoFeedback Projekte

5.1. Example Problems

Das Projekt Example Problems beinhaltet alle Beispielprobleme für die Performance Evaluation. Dazu gehören die Interfaces, die verschiedenen Implementierungen und die Testfälle für die Performance Evaluation. Auf die einzelnen Beispielprobleme und deren Implementierung wurde in meiner P7 Arbeit eingegangen [1]. Das Projekt Example Problems ist auf dem FHNW GitLab unter https://gitlab.fhnw.ch/autofeedback/performance/exampleproblems zu finden.

5.2. AutoFeedback Library

Das Projekt AutoFeedback Library beinhaltet alle Klassen, welche von verschiedenen anderen Projekten benötigt werden. Es besteht aus neuen Klassen für die Überprüfung der Korrektheit (Verifizierungs-Tool), aus bereits vorhandenen Klassen für die Performanceevaluation und aus Klassen für das Testen der Korrektheit von Studentenlösungen mit Unit Tests. Auf die Klassen in der Library kann sowohl vom AutoFeedback Evaluation Projekt als auch vom Dozenten Projekt zugegriffen werden.

Die Performanceevaluation wurde im Rahmen der P7 Arbeit entwickelt und kann anhand der Beispielprobleme getestet werden. Die dafür benötigten RPE-Klassen und die Tests befinden sich im Testordner der Library. Für die Ausführung der Tests wird das Projekt Example Problems mit den Implementierungen der Beispielprobleme benötigt.

Die Funktionsweise des Verifizierungs-Tools wurde im Kapitel 4 beschrieben und anhand der Verifizierungsbeispiele getestet. Sowohl das Tool selbst als auch die Verifizierungsbeispiele und die dazugehörigen Tests sind in diesem Projekt zu finden.

Das Projekt AutoFeedback Library ist auf dem FHNW GitLab unter https://gitlab.fhnw.ch/autofeedback/autofeedback-library zu finden.

5.3. Frontend

Im Projekt Frontend befindet sich die AutoFeedback React Webseite. Auf der Webseite können von Dozenten neue Tasks mit Testfällen erstellt werden. Anschliessend können die Studenten dort ihre Lösungen bewerten lassen.

Dafür werden die Unit Tests und die Performance Tests auf die Studentenlösung angewendet, welche vom Dozenten im Task eingerichtet wurden. Neu wird auch das Verifizierungs-Tool auf die Studentenlösung angewendet. Das Frontend Projekt ist auf dem FHNW GitLab unter https://gitlab.fhnw.ch/autofeedback/frontend zu finden.

Die Webseite wurde im Lehrfonds-Projekt «Automatisches Feedback zu Programmieraufgaben» erstellt. Die Webseite wurde damals nicht auf einen Stand gebracht, der veröffentlicht werden konnte. In einer ergänzenden Veranstaltung wurde begonnen die Webseite zu aktualisieren und fehlende Funktionen zu ergänzen. Das Editieren und Löschen von jeglichen Elementen, die Möglichkeit Accounts zu erstellen und erstellte Accounts wieder zu löschen sowie das Routing zwischen den verschiedenen Ansichten wurde hinzugefügt beziehungsweise verbessert. Eine generelle Benutzeranleitung sowie Beispiele für das Erstellen der Testfälle und das Einreichen von Studentenlösungen fehlt momentan noch. Die Verbesserung der Webseite ist nicht Teil dieser Arbeit, sondern der ergänzenden Veranstaltung.

Das Verifizierungs-Tool wurde ins Projekt integriert und die Resultate können auf der Webseite angezeigt werden. Die Anzeige der Testresultate wurde, wie in Abbildung 3 zu sehen, um die Verifizierungsresultate erweitert.

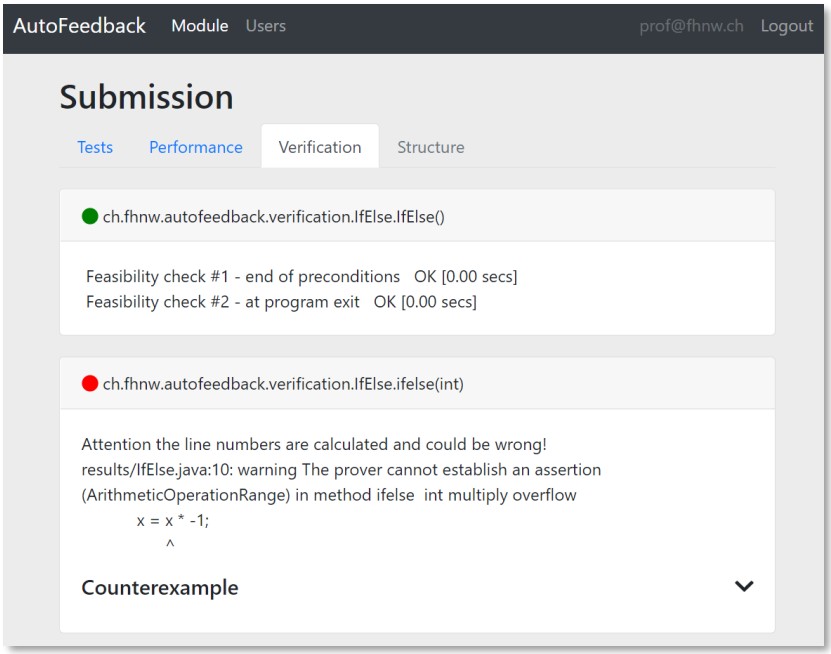


Abbildung 3: View Verifikationsresultate

In der Submissionsliste ist der aktuellen Status der Evaluation und eine Ampel, ob alle Tests korrekt waren, zu sehen. Die Liste wird jede Sekunde neu geladen, um das Resultat der noch laufenden Tests schnellstmöglich anzeigen zu können. Die Submissionsliste ist in Abbildung 4 zu sehen.

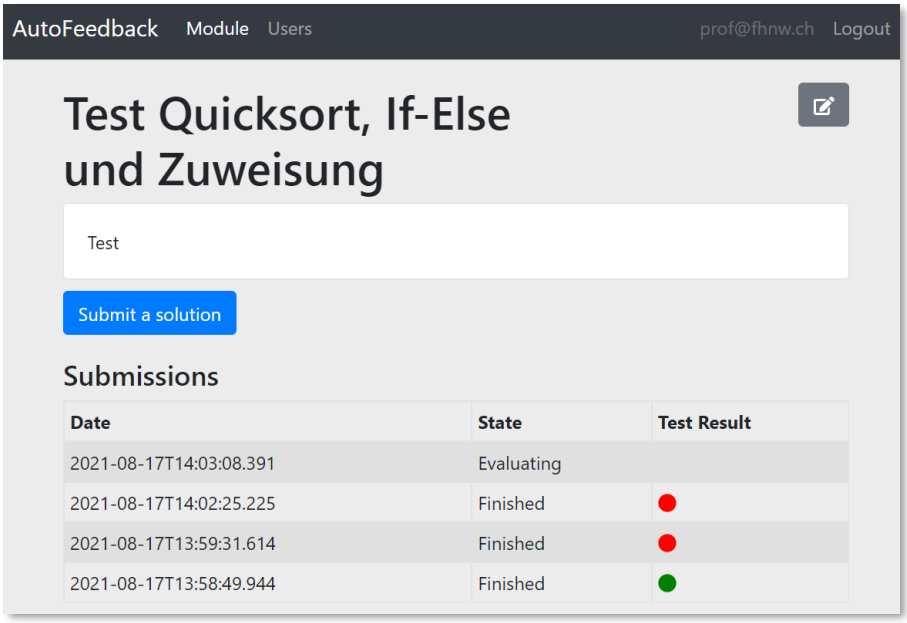


ABBILDUNG 4: VIEW SUBMISSIONSLISTE

5.3.1. ERSTELLEN UND ÄNDERN EINES TASKS

In einem Modul können neue Tasks erstellt oder bestehende Tasks geändert werden. Jeder Task hat einen Namen und eine Beschreibung. Dabei können unter «Modules» verschiedene Module ausgewählt werden, in welchen dieser Task angezeigt und von den Studenten angewendet werden kann. Anschliessend wird eine Zip Datei mit den Testfällen des Dozenten hochgeladen. Das neu entwickelte Verifizierungs-Tool kann mit der Checkbox eingeschalten werden und im Eingabe Feld dahinter kann die maximale Ausführungszeit für die Verifizierung angegeben werden. Unter «Test Classes» werden alle Klassen der zu verwendenden Tests, inklusive der maximalen Ausführungszeit für den Task angegeben. Das Formular für die Erstellung und Änderung eines Tasks ist in Abbildung 5 ersichtlich.

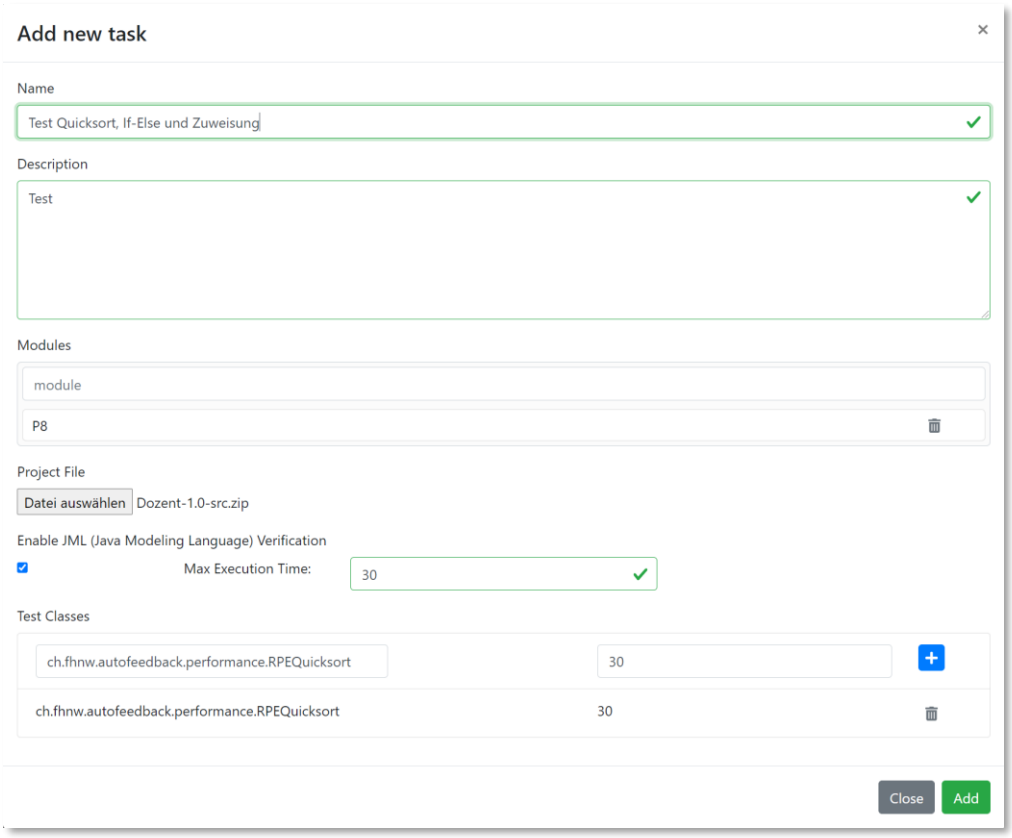


ABBILDUNG 5: TASK ERSTELLEN

5.3.2. *HOCHLADEN EINER STUDENTENLÖSUNG*

Ein Student kann seine Lösung als Zip Datei für einen bestimmten Test hochladen. Die Abgabe erfolgt über den «Submit a Solution» Button eines Tests, wie er in Abbildung 4 zu sehen ist. Anschliessend kann er seine Zip-Datei auswählen, mit «Submit» hochladen und testen lassen. Das Formular zum Hochladen einer Lösung ist in Abbildung 6 zu sehen.

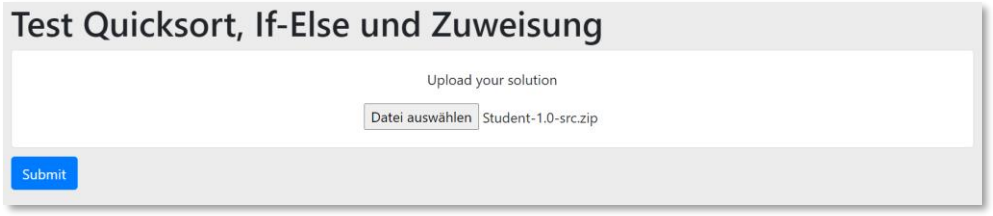


ABBILDUNG 6: HOCHLADEN EINER STUDENTENLÖSUNG

5.4. AUTOFEEDBACK BACKEND

Das AutoFeedback backend ist das Spring Boot Java backend zur React Webseite und wurde im Rahmen des Lehrfonds-Projekts «Automatisches Feedback zu Programmieraufgaben» entwickelt. Das AutoFeedback backend Projekt ist auf dem FHNW GitLab unter https://gitlab.fhnw.ch/autofeedback/autofeedback-backend zu finden.

Die Studentenlösung wird nicht direkt im Backend getestet. Die Studentenlösung und die Testfälle der Dozenten werden in einem Docker Container geladen. Dort werden sie kompiliert und vom Performance-Messtool, den Unittest und dem Verifizierungs-Tool getestet. Dadurch können Seiteneffekte des

Kompilierens und das Ausführen des Studenten-/Dozentencodes das Backend nicht beeinträchtigen. Weil die Evaluation der Lösungen nicht im Backend stattfindet, muss dieses für die Integration des Verifizierungs-Tools kaum angepasst werden.

5.5. AutoFeedback Evaluation

Die Studentenlösungen und die Testfälle der Dozenten werden wie bereits erwähnt, in einem Docker Container kompiliert und anschliessend vom der AutoFeedback Evaluation ausgeführt. Wie diese Studentenlösung und die Testfälle des Dozenten zu verwenden sind, wird aus dem Task bei welchem die Studentenlösung hochgeladen wurde ausgelesen. Als erstes wird überprüft ob für diesen Task die Verwendung des Verifizierungs-Tools eingeschalten wurde. Falls ja werden die Java Klassen der Studentenlösung auf JML Annotationen durchsucht und falls eine gefunden wird die Klasse mit dem Verifizierungs-Tool überprüft. Anschliessend werden die Testfälle des Dozenten abgearbeitet. Für jeden Test wird erst die Testklasse des Dozenten mit reflection geladen. Anschliessend wird überprüft, ob es sich um eine Performance oder einen Unit Test handelt. Der Test wird ausgeführt und die Resultate werden in die Datenbank gespeichert.

Die AutoFeedback Evaluation wurde um das in Kapitel 4 beschriebene Verifikations-Tool erweitert. Das AutoFeedback Evaluation Projekt ist auf dem FHNW GitLab unter https://gitlab.fhnw.ch/autofeedback/autofeedback-backend-evaluation zu finden.

5.6. Student

Im Projekt Student befindet sich ein Beispielprojekt eines Studenten. Dieses Projekt soll im Zusammenspiel mit dem Projekt Dozent (sehe Kapitel 5.7) zum Test der AutoFeedback Applikation dienen. Damit ein Studentenprojekt von der AutoFeedback Applikation bewertet werden kann, muss dieses auf die Webseite hochgeladen werden. Dafür wird der *src* Ordner mit allen Klassen und die *build.gradle* Datei zusammen gezipt und hochgeladen. Die hochgeladene ZIP Datei wird vom AutoFeedback Projekt in einem Docker Container entpackt und anhand der Gradle Datei kompiliert.

Dieses Projekt wurde mit der Zuweisung und dem If-Else Beispiel aus Kapitel 2.2 und 2.4 erweitert. Das Projekt Student ist auf dem FHNW GitLab unter https://gitlab.fhnw.ch/autofeedback/reflection-student zu finden.

5.7. Dozent

Im Projekt Dozent befindet sich ein Beispielprojekt eines Dozenten. Darin werden Tests für die Studentenlösung definiert, welche in der AutoFeedback Applikation getestet werden sollen. Damit ein Dozentenprojekt von der AutoFeedback Applikation zum Bewerten von Studentenlösungen verwendet werden kann, muss auf der Webseite ein neuer Test erstellt und das Projekt hochgeladen werden. Dieser Vorgang wird in Kapitel 5.3.1 erklärt. Von einem Dozentenprojekt wird der *src* Ordner mit allen Klassen und die *build.gradle* Datei zusammen gezipt und hochgeladen. Die hochgeladene ZIP Datei wird vom AutoFeedback Projekt in einem Docker Container entpackt und anhand der Gradle Datei kompiliert. Alle benötigten Klassen befinden sich im Projekt AutoFeedback Library. Dieses Projekt kann in der Gradle Datei als Abhängigkeit angegeben werden. Dabei ist zu beachten, dass alle Abhängigkeiten, die in der Gradle Datei definiert werden, auch im Docker Container geladen werden können. Das Projekt Dozent ist auf dem FHNW GitLab unter https://gitlab.fhnw.ch/autofeedback/reflection-teacher zu finden.

6. DISKUSSION

Für die Evaluation der bestehenden Tools und für das spätere Testen des Verifizierungs-Tools wurden verschiedene Verifizierungsbeispiele erstellt. Dank diesen Beispielen konnten die bestehenden Tools getestet und das Verifizierungs-Tool messbar und effizient verbessert werden.

Verschiedene bestehende Tools wurden im Rahmen einer Voranalyse auf ihre Tauglichkeit untersucht. Dabei konnten die meisten Tools bereits ausgeschlossen werden, weil sie nicht für einen Hoare-Triple-artige Verifizierung von Java Code konzipiert sind. Zusätzlich hat die Tatsache, dass die Tools nicht sehr weit verbreitet und oder schlecht dokumentiert waren, die Evaluation erschwert. Bereits die Inbetriebnahmen und das Testen der einfachsten Beispiele, wie der Zuweisung, war zeitaufwendig und nicht immer möglich. Mit OpenJML konnte jedoch ein Tool gefunden werden, welches die meisten Anforderungen erfüllt. OpenJML kann eine Verifizierung mit Vor- und Nachbedingungen und Invarianten vornehmen und bei Fehlern Gegenbeispiele liefern.

Es konnten jedoch nur etwa die Hälfte der Verifizierungsbeispiele ohne zusätzliche Bedingungen verifiziert werden. Dieser Umstand konnte mit dem Verifizierungs-Tool, einem zusätzlichen Wrapper um OpenJML, deutlich verbessert werden.

Das Verifizierungs-Tool nimmt die Studentenlösungen entgegen und versucht diese so aufzubereiten, dass sie möglichst einfach mit OpenJML verifiziert werden können. Dafür werden logische Vorbedingungen automatisch generiert und eingefügt, einfache For-Schleifen ausgerollt und gekennzeichnete Variablenüberläufe ignoriert. Die Resultate der Verifizierung werden anschliessend mit allfälligen Gegenbeispielen dem Studenten wieder zur Verfügung gestellt. Dabei werden die Lösungen sowohl auf ihre Korrektheit gegenüber der Spezifikation als auch auf Variablenüberläufe getestet. Ebenfalls können mit *assume* und *assert* Statements und Invarianten Teile einer Methode überprüft werden, ohne das Vor- oder Nachbedingungen für diese Methode spezifiziert wurden. OpenJML bringt nicht alle gewünschten Funktionen mit sich und hat auch Nachteile. Es können keine Weakest Preconditions berechnet werden. Der einschneidendste Nachteil von OpenJML ist, dass das Tool nur mit Java 8 funktioniert. Der Code, welcher neuere Konstrukte enthält, kann nicht getestet werden. Die aktuelle Weiterentwicklung von OpenJML sollte dies jedoch in Zukunft auch für neuere Java Versionen ermöglichen. Aktuell wird eine Alphaversion von OpenJML entwickelt, welche auf Java 16 basiert. Bei einem Release dieser Version wäre eine Aktualisierung der OpenJML Version im AutoFeedback Projekt ein Mehrwert. Durch das Aktualisieren der OpenJML Version kann das gesamte Verifizierungs-Tool relativ einfach auf eine neue Java Version aktualisiert werden.

Das erstellte Verifizierungs-Tool konnte, wie das Performance-Messtool im letzten Projekt, in das AutoFeedback Projekt integriert werden. Dies ermöglicht es den Studenten den entwickelten Code selbstständig online auf Korrektheit gegenüber ihrer Spezifikation zu verifizieren. Die Usability der Webseite wird momentan verbessert und die Webseite sollte anschliessend für Testnutzer online zugänglich gemacht.

ANHANG

LITERATUR

[1] J. Emmenegger, «AutoFeedback – Performance,» 2021.

[2] C. A. R. Hoare, «An Axiomatic Basis for Computer Programming,» *Commun. ACM,* Bd. 12, p. 576–580, 1969.

[3] C. Stamm, *Algorithmen und Datenstrukturen 1 Programmverifikation,* 2020.

[4] S. J. C. Joosten, W. Oortwijn, M. Safari und M. Huisman, «An Exercise in Verifying Sequential Programs with VerCors,» in *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*, New York, NY, USA, 2018.

[5] Univercity of Twente - Formal Methods and Tools, «VerCors Introduction,» [Online]. Available: https://github.com/utwente-fmt/vercors/wiki/Introduction. [Zugriff am 2021 April 26].

[6] L. C. Cordeiro, D. Kroening und P. Schrammel, «Benchmarking of Java Verification Tools at the Software Verification Competition (SV-COMP),» *SIGSOFT Softw. Eng. Notes,* Bd. 43, p. 56, 1 2019.

[7] W. Visser und J. Geldenhuys, «COASTAL: Combining Concolic and Fuzzing for Java (Competition Contribution),» in *Tools and Algorithms for the Construction and Analysis of Systems*, Cham, 2020.

[8] Y. Noller, C. S. Păsăreanu, A. Fromherz, X.-B. D. Le und W. Visser, «Symbolic Pathfinder for SV-COMP,» in *Tools and Algorithms for the Construction and Analysis of Systems*, Cham, 2019.

[9] D. Brizhinev und R. Goré, «A case study in formal verification of a Java program,» *CoRR,* Bd. abs/1809.03162, 2018.

[10] E. G. Karpenkov, K. Friedberger und D. Beyer, «JavaSMT: A Unified Interface for SMT Solvers in Java,» in *Verified Software. Theories, Tools, and Experiments - 8th International Conference, VSTTE 2016, Toronto, ON, Canada, July 17-18, 2016, Revised Selected Papers*, 2016.

[11] D. R. Cok, «Does your software do what it?,» 30 April 2021. [Online]. Available: https://github.com/OpenJML/OpenJML/releases/download/0.8.55/openjml-0.8.55-20210712.zip. [Zugriff am 10 August 2021].

[12] D. v. B. a. F. T. Nicholas Smith, JavaParser: Visited, https://leanpub.com/javaparservisited: Leanpub, 2021.

[13] E. P. C. C. Y. C. C. R. D. C. P. M. J. K. P. C. D. M. Z. W. D. Gary T. Leavens, «JML Reference Manual,» 31 Mai 2013. [Online]. Available: https://www.cs.ucf.edu/~leavens/JML/refman/jmlrefman.pdf. [Zugriff am 3 August 2021].

[14] B. Sitnikovski, «Tutorial on implementing Hoare logic for imperative programs in Haskell,» *CoRR,* Bd. abs/2101.11320, 2021.

[15] M. S. Nawaz, M. Malik, Y. Li, M. Sun und M. I. U. Lali, «A Survey on Theorem Provers in Formal Methods,» *CoRR,* Bd. abs/1912.03028, 2019.

[16] D. Matichuk, «Automatic Function Annotations for Hoare Logic,» in *Proceedings Seventh Conference on Systems Software Verification, SSV 2012, Sydney, Australia, 28-30 November 2012*, 2012.

[17] E. Lederer, «Beweisen statt Testen - Höhere Zuverlässigkeit durch die Java Modeling Langugage,» *IMVS Fokus Report,* 2008.

ABBILDUNGSVERZEICHNIS

CODEVERZEICHNIS

FORMELVERZEICHNIS

TABELLENVERZEICHNIS

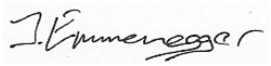
FEEDBACKVERZEICHNIS

EHRLICHKEITSERKLÄRUNG

Hiermit erkläre ich, den vorliegenden Projektbericht P8 AutoFeedback – Korrektheit für selbständig, ohne Hilfe Dritter und nur unter Benutzung der angegebenen Quellen verfasst zu haben.

Küttigen, 20.08.2021

Ort, Datum



Joel Emmenegger

ZEITPLANUNG