Karlsruhe Institute of Technology

Semantische Identifikation von Testklonen

Bachelorarbeit
von

Sven Scheu

An der Fakultät für Informatik
Institut für Programmstrukturen
und Datenorganisation (IPD)

Erstgutachter:	Prof. Dr. Walter F. Tichy
Zweitgutachter:	Prof. Dr. Ralf H. Reussner
Betreuender Mitarbeiter:	Dipl. Inform.-Wirt Mathias Landhäußer

Bearbeitungszeit: 17.05.2013 – 20.09.2013

Ich versichere wahrheitsgemäß, die Arbeit selbstständig angefertigt, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde.

Die Regeln zur Sicherung guter wissenschaftlicher Praxis im Karlsruher Institut für Technologie (KIT) habe ich befolgt.

Karlsruhe, 17.09.2013

. . . . . .

(**Sven Scheu**)

Kurzfassung

Diese Arbeit liefert einen Überblick über semantische Klonerkennungsverfahren, die allgemeine Quelltextklone identifizieren können. Anschließend wird ergründet, welche dieser Verfahren so angepasst werden können, dass das Klonerkennungswerkzeug *Jutci* um diese erweitert werden kann.

Einer der daraus resultierenden Ansätze, eine durch Programmabhängigkeitsgraphen unterstützte Klonerkennung, wird im Rahmen dieser Arbeit in *Jutci* integriert.

Die Güte dieser Modifikationen wird in einer Evaluation bewertet. Es wird dabei gezeigt, dass die modifizierte Klonerkennung eine bessere Präzision bei vergleichbarer Ausbeute erzielen kann.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einleitung

Ziel dieser Arbeit ist es, zu prüfen ob und wie semantische Verfahren verwendet werden können, um das Klonerkennungswerkzeug *Jutci* zu verbessern, das Johann Böhler in seiner Arbeit „Identifikation von Test-Klonen" [Bö13] vorstellt. *Jutci* ist in der Lage, mithilfe syntaktischer Analyseverfahren Testklone zu erkennen. In seiner Evaluation findet Böhler allerdings Fälle, bei denen Fehler in der Klonerkennung auftreten, weil Semantik und Syntax divergieren. Dies erfolgt hauptsächlich, wenn *Jutci* so konfiguriert wird, dass eine hohe Ausbeute erreicht wird. Dann kommt es durch die rein syntaktische Betrachtung zu einer geringen Präzision, die die Nutzbarkeit der Auswertung beeinträchtigt.

Die im Rahmen dieser Arbeit betrachteten Verfahren sollen eine gesteigerte Präzision bei gleichbleibender Ausbeute erzielen. Um dies zu erreichen, werden die Schwachstellen von *Jutci* im Hinblick auf die Klonerkennung untersucht und Möglichkeiten aufgezeigt, diese zu beseitigen. Die Praktikabilität dieser Verfahren wird dann am Beispiel einer auf Programmabhängigkeitsgraphen basierenden Analyse demonstriert, wobei diese Graphen zur Abstraktion der Semantik verwendet werden. In der anschließenden Evaluation wird das Resultat dieser Veränderungen veranschaulicht und die Güte der modifizierten Klonerkennung ermittelt. Für die dabei auffallenden möglichen Probleme werden mögliche Verbesserungen in Ausblick gestellt.

Diese Arbeit, wie auch die von Böhler, reiht sich damit in ein größeres Projekt ein, das zur Zeit am Institut für Programmstrukturen und Datenorganisation (IPD) des KIT durchgeführt wird und sich mit der automatischen Generierung von Testfällen beschäftigt [LT12]. Aktuell werden dazu aus bestehenden Testfällen Klone abgeleitet. Diese und verwandte Arbeiten können dabei helfen, ähnliche und damit für das Klonen geeignete Testfälle zu bestimmen, denn aus Analogien zwischen diesen Klonen und ihrem dazugehörigen Quelltext können möglicherweise Methoden abgeleitet werden, Testfälle für neuen Quelltext zu generieren.

2. Grundlagen

Dieses Kapitel liefert einen kurzen Überblick über die in dieser Arbeit verwendeten Begriffe.

2.1. Jutci

Jutci ist die Abkürzung für ein Werkzeug mit dem Namen *Java unit test clone investigator*, das von Böhler im Rahmen seiner Bachelorarbeit [Bö13] entwickelt wurde. Dieses Werkzeug ermöglicht die Erkennung von Testklonen und arbeitet dabei in vier Stufen:

1. **Vorfilter**: Sortiert Klonkandidaten anhand einfacher Kriterien, beispielsweise der Zeilenzahl von Methoden, aus.
2. **Wandler**: Modifiziert den Quelltext nach festgelegten Regeln. Derzeit findet nur eine Umwandlung von testrahmenspezifischem Quelltext in eine neutrale Form statt.
3. **Metriken**: Vorvergleich, der Klonkandidaten frühzeitig verwerfen kann. Beispielsweise durch Vergleich der Anzahl an Methodenaufrufen.
4. **Finale Auswertung**: Eigentlicher Vergleichsschritt anhand der längsten gemeinsamen Teilsequenz.

Da zur Zeit die Klonerkennung von Jutci sehr stark an der Syntax hängt, versucht diese Arbeit die Klonerkennung so zu verändern, dass die Syntax des Quelltextes eine kleineren Einfluss auf die Erkennung von semantischen Klonen hat. (Was ein semantischer Klon ist wird in Abschnitt 2.3 beschrieben.)

2.2. Testrahmen

Ein Testrahmen ist eine Programmierbibliothek, die es ermöglicht automatische Modultests zu definieren und diese auszuführen. Die Konzepte dieser Arbeit sind grundsätzlich auf jegliche Art von Programmiersprachen und automatischen Modultests anwendbar, die Beispiele und Implementierungsdetails beziehen sich allerdings ausschließlich auf Java in Verbindung mit JUnit.

2.2.1. Automatische Modultests

Ein automatischer Modultest (engl. unit test) ist ein von einem Programm durchgeführter Test eines oder mehrerer Module eines Programms. Meist lassen sich, wie auch bei

JUnit, Gruppen von Testfällen zu Testpaketen (engl. test suite) zusammenfassen, die dann gemeinsam ausgeführt werden können. Jedes Testpaket kann dabei eine oder mehrere Initialisierungs- und Nachbearbeitungsfunktionen haben, die, je nach Konfiguration, einmalig oder für jeden Testfall getrennt ausgeführt werden. Die von einem Modultest getesteten Module sind ein vom Rest des Programms isolierter Teil des Gesamtprogramms. Um diese Isolation zu gewährleisten, können Teile des Programms durch eine Nachahmung ersetzt werden. Solch eine Nachahmung simuliert dann die Implementierung einer Schnittstelle, so dass der Test durchgeführt werden kann [MFC]. Anzumerken ist, dass mit einem Modultest nicht die innere Funktionsweise eines Moduls getestet wird, sondern lediglich die Schnittstelle – also ob die erhaltene Ausgabe mit der erwarteten Ausgabe übereinstimmt und es zu keinen unerwarteten Fehlern kommt.

2.3. Testklone

Ein Testklon ist ein Testfall, der einem anderen Testfall semantisch oder syntaktisch sehr ähnlich ist. Testklone entstehen oftmals durch Wiederverwendung von bestehendem Testquelltext. Es gibt eine Reihe von verschiedenen Arten von Klonen, die sich auf unterschiedliche Art von dem Originaltestfall unterscheiden. Um die semantischen Klone von den restlichen Klonen abzugrenzen, werden in Definition 2.3.1 vier verschiedene Typen von Klonen definiert. Diese Definitionen von Klonen stammen aus der Arbeit von Roy et al. [RCK09] und beschreiben jegliche Art von Quelltextklon.

Definition 2.3.1 (Klontypen von Roy et al. [RCK09])**.**

Typ-1 Klone haben identischen Quelltext zu ihrem Original, bis auf verhaltensinvariante Änderungen, wie Kommentare oder Layoutmodifikationen.

Typ-2 Klone sind identisch zu Klonen von Typ-1, können aber zusätzlich Veränderungen von Identifikatoren, Typen und Literalen enthalten.

Typ-3 Klone sind identisch zu Klonen von Typ-2, lassen aber zusätzlich auch das Hinzufügen oder teilweise Entfernen von Anweisungen zu.

Typ-4 Klone sind zwei Quelltextfragmente, die sich syntaktisch unterscheiden, aber dennoch identische Ergebnisse bei der Ausführung liefern.

Klone von Typ-1 bis Typ-3 zeichnen sich dadurch aus, dass sie über die Art der Modifikation des Originals definiert sind. Klone von Typ-4 hingegen sind nur über ihre semantische und nicht ihre syntaktische Ähnlichkeit definiert. Zur Veranschaulichung zeigt Abbildung 2.1 jeweils ein Beispiel der verschiedenen Klon-Typen zu dem Original-Quelltext aus Quelltext 2.1. Das Original dieses Beispiels berechnet für jeden Wert von 0 bis zu dem Eingabewert die Dreieckszahl und die Fakultät und ruft dann die Methode `foo` auf, deren Implementierungsdetails als irrelevant anzusehen sind. Die Charakteristika der Beispielklone zu diesem Original werden in Tabelle 2.1 beschrieben.

Bei den, in dieser Arbeit betrachteten, semantischen Klonen handelt es sich um Klone von Typ-4 (Klone von Typ-1 sind immer auch Klone von Typ-4).

2.4. Schneiden von Programmen

Beim *Schneiden von Programmen* (engl. slicing) wird versucht komplexe Programme in kleinere Teile zu unterteilen, meist um das Verständnis des Quelltextes zu verbessern. So kann ein *Programmschnitt* (im folgenden *Schnitt* (engl. slice)) beispielsweise die möglichen Auswirkungen von Veränderungen des Quelltextes darstellen.

```
1  public void triNumAndFact(int n) {
2    float sum = 0.f;
3    float prod = 1.f;
4    for(int i = 1; i <= n; i++) {
5      sum = sum + i;
6      prod = prod * i;
7      foo(sum, prod);
8    }
9  }
```

Quelltext 2.1: Original Quelltext

```
1  public void
       triNumAndFact(int n) {
2    float sum = 0.f;
3    float prod = 1.f;
4    //berechnet Dreieckszahl
       und Fak.
5    for(int i = 1; i <= n;
       i++) {
6      sum = sum + i;
7      prod = prod * i;
8      foo(sum, prod);
9    }
10 }
```

Quelltext 2.2: Typ-1 Klon

```
1  public void
       triNumAndFact(int n) {
2    int sum = 0;
3    float prod = 1.f;
4
5
6    for(int j = 1; j <= n;
       j++) {
7      sum = sum + j;
8      prod = prod * j;
9      foo((float) sum, prod);
10   }
11 }
```

Quelltext 2.3: Typ-2 Klon

```
1  public void
       triNumAndFact(int n) {
2    float sum = 0.f;
3    float prod = 1.f;
4    for(int i = 1; i <= n;
       i++) {
5      sum = sum + i;
6      //prod = prod * i;
         entfernt
7      foo(sum, prod);
8    }
9  }
```

Quelltext 2.4: Typ-3 Klon

```
1  public void
       triNumAndFact(int n) {
2    float prod = 1.f;
3    float sum = 0.f;
4    for(int i = 1; i <= n;
       i++) {
5
6      foo(sum += i, prod *=
         i);
7
8    }
9  }
```

Quelltext 2.5: Typ-4 Klon

Abbildung 2.1.: Beispiele der verschiedenen Klon-Typen (aus [RCK09])

Eingeführt wurde das Schneiden von Programmen erstmals 1981 durch Weiser [Wei81]. Ein Schnitt nach Weiser ist ein verhaltensäquivalenter Teil des Gesamtprogramms unter Berücksichtigung einer bestimmten Teilmenge der Programmergebnisse, den sogenannten *Programmschnittkriterien* (engl. slicing criterion). Zusätzlich zur Verbesserung des Verständnisses des Programms können Schnitte noch für eine Reihe anderer Dinge verwendet werden: so können sie miteinander verglichen werden, wobei ähnliche Schnitte in der Regel ähnliche Semantik nahelegen.

Die einfachste Form eines Schnittkriteriums ist die Betrachtung des Wertes einer bestimmten Variable an einem bestimmten Punkt des Programms. Ein solches Schnittkriterium wurde schon von Weiser eingeführt:

Klontyp	Veränderung	Verhalten
Typ-1	Ein Kommentar wurde hinzugefügt.	Verhaltensgleich zum Original.
Typ-2	Der Typ der Variable `sum` wurde auf `int` geändert.	Durch die Veränderung des Typs von `sum` verändert sich die Präzision und der Wertebereich von `sum`.
Typ-3	Die Anweisung `prod = prod * i;` wurde entfernt und `i` wurde in `j` umbenannt.	Der Wert für die Fakultät, der an `foo` übergeben wird, ist immer 1.
Typ-4	Die Variablendefinitionen wurden vertauscht, und die Zuweisungen und Operatoren zu kombinierten Zuweisungen vereinfacht und in den Methodenaufruf verschoben.	Verhaltensgleich zum Original.

Tabelle 2.1.: Kloncharakteristika

Definition 2.4.1 (Schnittkriterium nach Weiser)**.** Für ein gegebenes Programm mit n Zeilen ist ein Schnittkriterium ein 2-Tupel $< i, v >$, wobei $i \in (0, n]$ die zu betrachtende Zeile angibt und v die zu betrachtende Variable.

Für ein solches Schnittkriterium ist folglich jede Teilmenge des Gesamtprogramms ein gültiger Schnitt, wenn der Wert dieser Variable an der betrachteten Stelle durch die Wegnahme der ausgelassenen Anweisungen nicht verändert wird. Der Zweck, der mit dem Schneiden verfolgt wird, bestimmt das Schnittkriterium und das Vorgehen beim Schneiden. Daher gibt es kein allgemein gültiges Schnittkriterium, das für jede Art des Schneidens verwendet werden kann.

Unabhängig von der Form des Schnittkriteriums gilt: wenn ein Schnitt die kleinste mögliche Menge an Anweisungen enthält, die gerade noch ausreicht um alle Schnittkriterien zu erfüllen, dann handelt es sich um einen *minimalen Schnitt* für diese Menge an Schnittkriterien. Ein ausführbarer Schnitt hingegen ist jeder Schnitt, der auch ausführbar ist. (Bei ausführbaren Schnitten handelt es sich in der Regel nicht um minimale Schnitte.)

Das oben beschriebe Schnittkriterium wird für die grundlegende Art des Schneidens, das *Rückwärtsschneiden*, verwendet. In den folgenden Abschnitten werden *Rückwärtsschneiden* und *Vorwärtsschneiden* genauer erklärt. Es gibt allerdings noch eine Reihe anderer Schnittmethoden – eine Übersicht über diese Schnittmethoden findet sich beispielsweise in der Arbeit von Silva [Sil12] oder der Arbeit von Tip [Tip95]. Im Allgemeinen, sowie im Folgenden, bezieht sich Schneiden meist auf Rückwärtsschneiden (sofern nicht anderweitig angegeben).

2.4.1. Rückwärtsschneiden

Rückwärtsschneiden dient hauptsächlich der Beantwortung der Frage, welche Anweisungen des Gesamtprogramms den Wert einer bestimmten Variable an einem bestimmten Punkt des Programms beeinflussen können. Hierzu sucht man, von einer Anweisung ausgehend, rückwärts alle Anweisungen, die diese Anweisung beeinflussen könnten. Ein Rückwärtsschnitt zu dem Quelltext 2.1 und dem Schnittkriterium $< 7, sum >$ liefert beispielsweise die in Quelltext 2.6 abgebildete Teilmenge aller Anweisungen.

2.4.2. Vorwärtsschneiden

Vorwärtsschneiden hingegen versucht zu beantworten, welche Teile des Programms durch die Veränderung einer Anweisung betroffen wären [RB89]. Es werden also von dem Schnitt-

```
1  public void triNumAndFact(int n) {
2    float sum = 0.f;
3
4    for(int i = 1; i <= n; i++) {
5      sum = sum + i;
6
7      foo(sum, prod);
8    }
9  }
```

Quelltext 2.6: Rückwärtsschnitt mit $< 7, sum >$

kriterium aus vorwärts alle Anweisung gesucht, die durch direkte oder indirekte Veränderung des Schnittkriteriums beeinflusst werden könnten, wobei das Schnittkriterium analog zu dem aus Definition 2.4.1 definiert ist. Zusätzlich kann Vorwärtsschneiden aber auch dazu genutzt werden, die Ergebnisse anderer Schnittmethoden zu verbessern. Die Arbeit von Komondoor et al. [KH01] beispielsweise benutzt Vorwärtsschneiden, um den Einfluss von Schleifen auf den Kontrollfluss besser abzubilden. Ein Vorwärtsschnitt zu dem Quelltext 2.1 und dem Schnittkriterium $< 2, sum >$ liefert beispielsweise die in Quelltext 2.6 abgebildete Teilmenge aller Anweisungen.

```
1    float sum = 0.f;
2
3
4      sum = sum + i;
5
6      foo(sum, prod);
```

Quelltext 2.7: Vorwärtsschnitt mit $< 2, sum >$

2.5. Programmabhängigkeitsgraphen

Eine Möglichkeit Schnitte zu erhalten, ist es sie mithilfe eines sogenannten Programmabhängigkeitsgraphen (engl. program dependency graph, PDG) zu bestimmen. Beim Rückwärtsschneiden werden dazu rückwärts die eingehenden Kanten des Startknotens abgelaufen. Alle besuchten Knoten werden dann zum Schnitt hinzugefügt. Von diesen Knoten aus wird der Vorgang dann solange wiederholt, bis alle eingehenden Kanten des Schnitts besucht wurden. PDGs als Grundlage zum Schneiden wurde erstmals von Ottenstein und Ottenstein [OO84] eingeführt. Ein PDG ist im allgemeinen ein beschrifteter gerichteter Graph, so wie er in Definition 2.5.1 beschrieben wird.

Definition 2.5.1 (Beschrifteter gerichteter Graph)**.** Ein *beschrifteter gerichteter Graph* ist ein 4-Tupel $G = (V, E, \lambda_v, \lambda_e)$. Dabei ist V eine Menge von Knoten und $E \subseteq V \times V$ eine Menge von Kanten. Die Funktion $\lambda_v : V \to B_v$ ist eine Abbildung von einer Menge an Knoten V zu einer Menge an Beschriftungen B_v. Dazu analog ist die Funktion $\lambda_e : E \to B_e$ eine Abbildung von einer Menge an Kanten E zu einer Menge an Beschriftungen B_e.

Die Beschriftung der Knoten spezifiziert die Art des Knotens, wie Eingangsknoten oder Zuweisungsknoten. Die Beschriftung der Kanten gibt die Art des Verhältnisses zwischen den Knoten an, wie Kontrollabhängigkeit oder Wertabhängigkeit. Die genauen Details der Beschriftungen von Kanten und Knoten hängen stark von der jeweiligen Version des PDG ab und was mit diesem erreicht werden soll. Ein Beispiel für einen PDG ist in Abbildung 2.2 dargestellt. Dieser PDG wurde dem von Krinke in [Kri01] entworfenen PDG

```
1  static int foo(int a, int b, int c){
2    int x, y, z;
3    x = a * (y = b + c);
4    z = x + y;
5    return z;
6  }
```

Quelltext 2.8: PDG Beispielcode

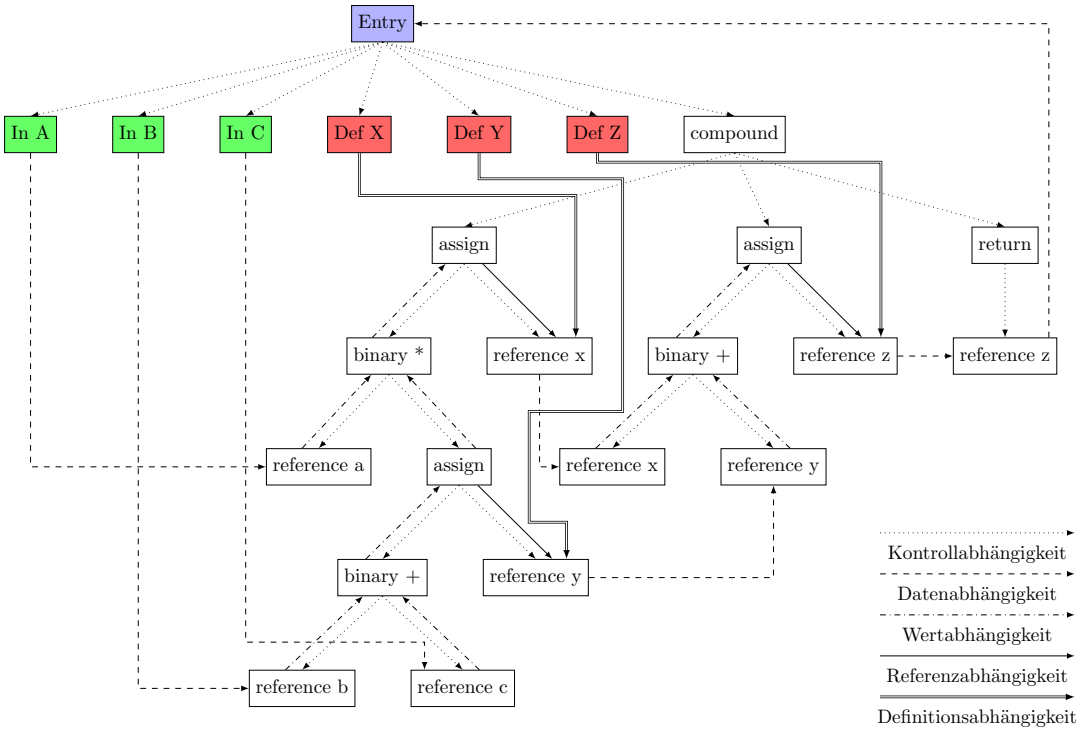

Abbildung 2.2.: PDG zu dem Beispiel 2.8

nachempfunden – der zugehörige Quelltext findet sich in Quelltext 2.8 (das Orginalbeispiel ist in C und wurde angepasst auf Java).

In diesem Graphen ist der lila Knoten der Einstiegsknoten, die grünen Knoten sind Eingangsknoten, die roten Knoten stellen Variablen-Definitionsknoten dar und die restlichen Knoten sind allgemeine Knoten. Die Kantentypen bedeuten folgendes:

- **Kontrollabhängigkeit**: Der Zielknoten wird für die Ausführung des Vaterknotens benötigt.
- **Datenabhängigkeit**: Der Zielknoten liest die im Ursprungsknoten geschriebene Variable.
- **Wertabhängigkeit**: Der Wert des Zielknotens ist abhängig vom Wert des Ursprungsknotens.
- **Referenzabhängigkeit**: Der Ursprungsknoten schreibt in den Zielknoten.
- **Definitionsabhängigkeit**: Der Zielknoten schreibt das erste Mal in die im Ursprungsknoten definierte Variable.

Sowohl die Kantentypen als auch die Knotentypen sind, wie schon erwähnt, implementierungsspezifisch und dienen hier nur der Veranschaulichung des Konzepts.

2.6. Graphenisomorphie und -ähnlichkeit

Eine weitere Möglichkeit Schnitte zu erstellen, ist die Bestimmung eines isomorphen Teilgraphen oder eines ähnlichen Teilgraphen zwischen den PDGs zweier Methoden. Isomorphie zwischen Graphen bedeutet dabei, dass es möglich ist, den einen Graph auf den anderen abzubilden:

Definition 2.6.1 (Graphenisomorphie [DM73])**.** Zwei Graphen $G_1 = (V_1, E_1)$ und $G_2 = (V_2, E_2)$ sind isomorph zueinander, wenn es eine bijektive Abbildung $f : V_1 \to V_2$, mit der Umkehrfunktion f^{-1} gibt, so dass gilt: Es gibt genau dann eine Kante von Knoten v_{2_a} nach v_{2_b} mit $v_{2_a}, v_{2_b} \in V_2$, wenn es eine Kante von $v_{1_a} = f^{-1}(v_{2_a})$ nach $v_{1_b} = f^{-1}(v_{2_b})$ mit $v_{1_a}, v_{1_b} \in V_1$ gibt. Solch eine äquivalente Kante muss für alle Kanten in E_1 und E_2 existieren. Im Falle von beschrifteten Graphen müssen zusätzlich alle Beschriftungen an äquivalenten Kanten und Knoten gleich sein.

Die durch die Isomorphiebestimmung erhaltenen Schnitte können beispielsweise dazu verwendet werden, semantische Klone zu finden. Zwar kann gleiche Semantik auch sehr unterschiedlich implementiert werden, aber isomorphe Teilgraphen sind dennoch eine gute Abstraktion von Verhaltensgleichheit. Das liegt daran, dass die Anweisungsordnung, sofern sie irrelevant für die Variablenwerte ist, keinen Einfluss auf die Form des PDGs hat.

Ist ein isomorpher Teilgraph gefunden, kann mit Hilfe der Abbildung der Knoten aus G_1 auf G_2 ein Ähnlichkeitsmaß für die Graphen bestimmt werden. Ein einfaches Ähnlichkeitsmaß ist beispielsweise das Verhältnis der äquivalenten Teile der Graphen zu den nicht äquivalenten Teilen. Abgesehen von der Bestimmung des Verhältnisses zwischen der isomorphen und nicht isomorphen Knotenzahl gibt es noch eine Reihe anderer Methoden zur Bestimmung eines Ähnlichkeitsmaßes, die beispielsweise in der Arbeit von Champin et al. [CS03] genauer behandelt werden.

3. Verwandte Arbeiten

In den letzten Jahren gab es eine Reihe neuer Entwicklungen in allen relevanten Bereichen dieser Arbeit. Die verwandten Arbeiten werden im Folgenden erläutert.

3.1. Semantische Klonerkennung

Die semantische Erkennung von Testklonen ist eng verwandt mit der allgemeinen semantischen Klonerkennung. Diese Arbeit legt zwar ihren Schwerpunkt auf die Klonerkennung durch Analyse von PDGs, dennoch gibt es eine Reihe anderer Ansätze, die ähnliche Ziele verfolgen.

3.1.1. PDG-basierte Verfahren

Ein PDG-basiertes Verfahren ist beispielsweise das von Krinke zur Identifikation von Klonen mit Hilfe von PDGs [Kri01]. Dafür führt er eine neue Art des PDGs ein: den sogenannten *fine grained PDG* der den abstrakten Syntaxbaum (AST) und den PDG einer Prozedur miteinander vereint. Die eigentliche Identifikation von Klonen basiert bei Krinke auf der Suche nach ähnlichen Teilgraphen. Dabei ist ein Teilgraph ähnlich zu einem anderen Teilgraph, wenn sich eine Abbildung finden lässt, so dass alle Knoten des Schnitts direkt oder indirekt mit dem Eingangsknoten verbunden sind. Für diese Abbildung müssen nur Kanten und Knotentypen übereinstimmen und jeder Knoten muss nur eine übereinstimmende Kante haben, es handelt sich hierbei also nicht um echte Isomorphie. Die in dieser Arbeit verwendeten PDGs sind konzeptionell sehr ähnlich zu denen von Krinke.

Eine weitere Methode zur Erkennung von klonierten Teilsequenzen mithilfe von PDGs ist die von Komondoor et al. [KH01]. Die Autoren beschrieben in ihrem Papier eine Methode, wie mit Hilfe des Schneidens von PDGs eine Klonerkennung implementiert werden kann. Die Klonfindung basiert auf der Bestimmung von isomorphen Teilgraphen. Die PDGs zweier Prozeduren werden dabei nicht direkt miteinander verglichen, sondern der Algorithmus vergleicht immer nur zwei Schnitte, die jeweils zwei übereinstimmende Knoten verbinden. Da nur diese Schnitte verglichen werden, eignet sich dieser Ansatz allerdings nur zur Erkennung von ähnlichen Teilsequenzen und nicht zur Erkennung ganzer geklonter Prozeduren. In ihrer Arbeit beschreiben die Autoren außerdem, welche Probleme durch Klone auftreten, die Quelltextstellen innerhalb und außerhalb von Schleifen haben. Die Autoren verwenden deshalb Vorwärtsschneiden, von den Schleifenknoten ausgehend, um eine bessere Repräsentation der Auswirkung von Schleifen zu erhalten.

3.1.2. Linguistische Verfahren

Das Verfahren von Kuhn et al. benutzt die sogenannte semantische Clusteranalyse [KDG07], um ähnliche Quelltextdateien zu finden. (Die Grundsatzarbeit zu diesem Thema stammt von Maletic und Marcus [MM00].) Bei der semantischen Clusteranalyse wird versucht, ähnliche Dateien über die verwendete Sprache zu identifizieren. Dazu haben die Autoren ein System entworfen, das alle Dokumente und ihren Inhalt als eine Matrix darstellt, die Dokumente auf Begriffe abbildet. Aus dieser Matrix kann dann mithilfe von *Latent Semantik Indexing* (siehe [DDL+90]) eine Korrelationsmatrix erstellt werden. Abschließend lassen sich die Dokumente anhand dieser Korrelationsmatrix in Cluster zusammenfügen, wobei jeder Cluster Dokumente mit ähnlichen Themen enthält. Die Cluster können dann unter anderem dazu verwendet werden, die Klonsuche einzuschränken, so dass nur Dateien, die sich im selben Cluster befinden und folglich das gleiche Thema haben, miteinander verglichen werden.

3.1.3. Metrikbasierte Verfahren

Die Arbeit von Kontogiannis et al. [KDM+96] befasst sich mit zwei verschiedenen Methoden der Klonerkennung. Die erste ist eine metrische Analyse, die verschiedene Metriken auswertet, um mögliche Klone zu finden (ähnlich wie Jutci, allerdings mit anderen Metriken).

Die andere von den Autoren vorgestellte Methode ist eine auf dynamischer Programmierung basierende Methode. Diese Methode findet Klone, indem sie Quelltextpaare bestimmt, bei denen sich beide Teile mit einer spezifischen Anzahl an Änderungen (Einfügungen, Löschungen und Ersetzungen) ineinander überführen lassen. Sowohl zusätzliche Metriken, als auch die auf der dynamischen Programmierung basierte Methode könnten möglicherweise die Klonerkennung von Jutci verbessern.

3.2. Schneiden

Die Originalform des Schneidens nach Weiser [Wei81] war nur für monolithische Prozeduren, also solche Prozeduren, die keine weiteren Prozeduren aufrufen, konzipiert. Um diese Beschränkungen zu beseitigen, wurden alternative Schnittverfahren entwickelt.

3.2.1. Interprozedurales Schneiden

Eine Verbesserung des klassischen Schneidens stammt aus der Arbeit [HRB90] von Horwitz et al., die ein Verfahren vorstellt, mit dem auch prozedurübergreifend korrekte Schnitte erstellt werden können. Als Ursache für inkorrekte interprozedurale Schnitte geben Horwitz et al. in ihrer Arbeit an, dass klassisches Schneiden den Kontext von Prozeduraufrufen nicht beachtet. Um den Kontext von Prozeduraufrufen richtig abzubilden, wird in dieser Arbeit eine neue Art von PDG eingeführt: der sogenannte Systemabhängigkeitsgraph (engl. system dependency graph, SDG). Ein SDG besteht dabei aus mehreren PDGs, einen für jede Prozedur des Gesamtprogramms, die über spezielle Kanten miteinander verbunden sind.

3.2.2. Schneiden von parallelen Programmen

Eine weitere Verbesserung des Schneidens ist die Berücksichtigung der Einflüsse von verschiedenen gleichzeitig ausgeführten Fäden auf geteilte Variablen. Eines der Verfahren zum Schneiden von parallelen Programmen stammt aus der Dissertation [Kri04] von Krinke, in der er beschreibt, wie Programme mit mehreren Fäden zu Schneiden sind. Die Programme müssen dafür bestimmte Voraussetzungen erfüllen: Jeder Faden benutzt seinen eigenen, von den anderen Fäden unabhängigen, Quelltext und die Kommunikation zwischen den Fäden erfolgt nur über globale Variablen in Verbindung mit atomaren Operationen.

4. Analyse und Entwurf

Das Ziel dieser Arbeit ist die Verbesserung der Klonerkennung durch stärkere Beachtung der Semantik. Dazu werden im Folgenden diese drei Fragestellungen erörtert:

- Was sind die Fälle, bei denen Jutci syntaktisch verschiedene aber semantisch ähnliche Testfälle nicht als Klonpaar erkennt, bzw. welche Fälle gibt es, in denen syntaktische Ähnlichkeit zur Fehlidentifikation als Klonpaar führt?
- Wie lassen sich die erkannten Probleme mit semantikbewussten Techniken lösen?
- Wie kann die Integration in Jutci am besten stattfinden?

4.1. Identifikation von Erkennungsschwächen

Bei der Klonerkennung von Jutci muss oftmals ein Kompromiss zwischen Ausbeute und Präzision eingegangen werden, um die gewünschte Ausbeute bzw. Präzision zu erreichen. Im Folgenden werden Fälle erörtert, in denen semantikbewusste Methoden zu einer erhöhten Ausbeute bei gleichbleibender Präzision bzw. erhöhter Präzision bei gleichbleibender Ausbeute führen könnten. Interessant sind dabei vor allem die Fälle, in denen die semantische Ähnlichkeit für einen menschlichen Klonsucher offensichtlich, die syntaktische Ähnlichkeit hingegen – und damit der von Jutci zugewiesene Ähnlichkeitswert – verhältnismäßig gering ist.

4.1.1. Methodik

Folgender Abschnitt beschreibt die Methodik die angewendet wurde, um die Fälle mit hoher Diskrepanz zwischen dem von Jutci bestimmten Ähnlichkeitswert und dem intuitiven semantischen Ähnlichkeitswert zu bestimmen. Eine solche hohe Diskrepanz liegt beispielsweise vor, wenn die Semantik zweier Testfälle vollkommen gleich ist, aber Jutci diesem Paar von Testfällen nur einen Ähnlichkeitswert von 60% zuweist. Zwar hat diese Form der Analyse ein erhebliches subjektives Element, dennoch sollte die hohe Differenz überzeugend sein. Um eine möglichst hohe Vergleichbarkeit der Evaluation dieser Arbeit und derer von Böhler zu gewährleisten, basieren die folgenden Ergebnisse auf der manuellen Identifikation von Testklonen von Böhler.

```
1     public void testSettingUsage() {
2         args = new String[]{"-wrong-usage"};
3         try {
4             parser.parseArgument(args);
5             fail("Doesnt detect wrong parameters.");
6         } catch (CmdLineException e) {
7             String expectedError = "\"-wrong-usage\" is not a
                  valid option";
8             String expectedUsage   = " -str VAL : set a string";
9             String errorMessage = e.getMessage();
10            String[] usageLines = getUsageMessage();
11            assertUsageLength(1);
12            assertTrue("Got wrong error message",
                  errorMessage.startsWith(expectedError));
13            assertEquals(expectedUsage, usageLines[0]);
14        }
15    }

17    public void testMissingParameter() {
18        args = new String[]{"-str"};
19        try {
20            parser.parseArgument(args);
21            fail("Should miss one parameter.");
22        } catch (CmdLineException e) {
23            String expectedError = "Option \"-str\" takes an
                  operand";
24            String expectedUsage   = " -str VAL : set a string";

26            String[] usageLines = getUsageMessage();
27            String errorMessage = e.getMessage();
28            assertUsageLength(1);
29            assertTrue("Got wrong error message: " + errorMessage,
                  errorMessage.startsWith(expectedError));
30            assertEquals("Got wrong usage message", expectedUsage,
                  usageLines[0]);
31        }
32    }
```

Quelltext 4.1: Auszug aus SetterTest.java (Args4j [Kaw13])

4.1.2. Semantikneutrale Umordnungen

Ein sehr offensichtliches Beispiel, bei dem die Jutci Auswertung stark von der Intuition eines Menschen abweicht, ist semantisch äquivalenter, aber umgeordneter Quelltext. Diese Formen von Klonen rufen selbst dann einen deutlich geringeren Ähnlichkeitswert als erwartet hervor, wenn nur zwei Anweisungen vertauscht wurden.

In dem Quelltext 4.1 aus Args4j findet sich ein offensichtlicher Klon, der dennoch erst ab einer Ähnlichkeitsschwelle unter 60% erkannt wird. Zu erwarten wäre aber eine Erkennung bei spätestens einer Ähnlichkeitsschwelle von 80%. (Ab diesem Wert werden die semantisch (und lexikalisch) praktisch gleichen Testfälle in Quelltext 4.2 als Klone erkannt.) Folglich reduziert das Vertauschen der beiden Anweisungen den Ähnlichkeitswert um fast $\frac{1}{4}$, obwohl die Semantik nicht verändert wurde.

4.1.3. Quelltextauslagerung

Ein weiteres Problem ist die Auslagerung von Testlogik in Hilfsmethoden. Dies kann zu mehreren Problemen führen.

```
1     public void testSettingUsage() {
2         args = new String[]{"-wrong-usage"};
3         try {
4             parser.parseArgument(args);
5             fail("Doesnt detect wrong parameters.");
6         } catch (CmdLineException e) {
7             String expectedError = "\"-wrong-usage\" is not a
                  valid option";
8             String expectedUsage   = " -str VAL : set a string";
9             String[] usageLines = getUsageMessage();
10            String errorMessage = e.getMessage();
11            assertTrue("Got wrong error message",
                  errorMessage.startsWith(expectedError));
12            assertUsageLength(1);
13            assertEquals("Got wrong usage message", expectedUsage,
                  usageLines[0]);
14        }
15    }

17    public void testMissingParameter() {
18        args = new String[]{"-str"};
19        try {
20            parser.parseArgument(args);
21            fail("Should miss one parameter.");
22        } catch (CmdLineException e) {
23            String expectedError = "Option \"-str\" takes an
                  operand";
24            String expectedUsage   = " -str VAL : set a string";
25            String[] usageLines = getUsageMessage();
26            String errorMessage = e.getMessage();
27            assertUsageLength(1);
28            assertTrue("Got wrong error message: " + errorMessage,
                  errorMessage.startsWith(expectedError));
29            assertEquals("Got wrong usage message", expectedUsage,
                  usageLines[0]);
30        }
31    }
```

Quelltext 4.2: Auszug aus SimpleStringTest.java (Args4j [Kaw13])

- Testfälle wirken dadurch teilweise deutlich kürzer, als sie tatsächlich sind. Das kann dazu führen, dass sie schon von dem Anweisungszahlvorfilter aussortiert werden, obwohl sie ohne diese Auslagerung lang genug gewesen wären. Außerdem enthält die Methode nach der Auslagerung weniger Anweisungen, weswegen metrikbasierte Vergleiche verzerrt werden.
- Das Ähnlichkeitsmaß von Jutci ist das Verhältnis von übereinstimmenden Anweisungen zur Gesamtmethodenlänge. Benutzen also zwei Testfälle die gleiche ausgelagerte Methode, so erscheinen die Methoden weit weniger ähnlich als wenn beide die Methode nicht ausgelagert hätten.

Im Beispiel in Quelltext 4.3 lassen sich beide Probleme leicht erkennen. Offensichtlich sind beide Testfälle sehr kurz, weswegen sie womöglich schon vor der eigentlichen Klonanalyse vom Anweisungszahlvorfilter aussortiert werden. Des Weiteren ist der Ähnlichkeitsgrad dieser beiden Testfälle nur 50%, da sie sich syntaktisch in einer Zeile unterscheiden: Wäre `executeIsEnabledTest` nicht ausgelagert worden, wäre der Ähnlichkeitsgrad viel höher.

An diesem Beispiel wird auch deutlich, wie Präzision und Ausbeute in Konkurrenz zueinander stehen: Bei abnehmender Ähnlichkeitsschwelle steigt die Ausbeute, während die Präzision sinkt. Für dieses Beispiel heißt das: Setzt man die Ähnlichkeitsschwelle von Jutci auf maximal 50%, ist die Ausbeute sehr gut, denn selbst dieses Klonpaar wird erkannt. Allerdings wird dann auch jeder andere zweizeilige Testfall als zusätzlicher Klon erkannt, wenn er mindestens einen Methodenaufruf mit genau einem Parameter enthält. So werden beispielsweise `testIsEnabledNamedLog` aus Quelltext 4.3 und `testPristineDecorated` aus Quelltext 4.4 als Klone erkannt, obwohl sie offensichtlich keine sind.

4.2. Mögliche Lösungsansätze

Folgender Abschnitt erörtert, welche semantischen Verfahren sich eignen könnten, die einzelnen Probleme zu beheben.

4.2.1. Schneiden an Zusicherungen

Unter *Schneiden an Zusicherungen* versteht man folgenden Ansatz: Mit Hilfe von Rückwärtsschneiden werden ausgehend von den JUnit-Zusicherungen die Anweisungen bestimmt, die die überprüften Werte beeinflussen. Anschließend werden einzelne Schnitte mittels den bestehenden Methoden miteinander verglichen. Dieser Ansatz beruht auf der Idee, dass Zusicherungen eine zentrale Rolle bei automatischen Modultests einnehmen und sie sich deshalb gut zum Schneiden eignen würden. Vorteil dieses Ansatzes ist, dass sich die schon bestehende Klonerkennung von Jutci weitgehend unverändert verwenden lassen würde und trotzdem könnte sich eine Verbesserung der Ergebnisse einstellen, da nur noch tatsächlich benötigte Anweisungen betrachtet werden. Andererseits wird nach genauerer Untersuchung offensichtlich, dass dieser Ansatz einige fundamentale Probleme hat:

- Viele Tests haben keine Zusicherungen (siehe z.b. Quelltext 4.3).
- Einige Testfälle haben weit mehr Zusicherungen als Anweisungen – die Schnitte wären also sehr klein.
- Irrelevante Anweisungen wie Protokollanweisungen werden in Praxistestfällen nur sehr selten verwendet.

Dieser Ansatz wird folglich nicht weiter beachtet.

```
1     public void testIsEnabledClassLog()
2     {
3         Log log = LogFactory.getLog(BasicOperationsTestCase.class);
4         executeIsEnabledTest(log);
5     }
6
7     public void testIsEnabledNamedLog()
8     {
9         Log log =
              LogFactory.getLog(BasicOperationsTestCase.class.getName());
10        executeIsEnabledTest(log);
11    }
12
13    public void executeIsEnabledTest(Log log)
14    {
15        try
16        {
17            log.isTraceEnabled();
18            log.isDebugEnabled();
19            log.isInfoEnabled();
20            log.isWarnEnabled();
21            log.isErrorEnabled();
22            log.isFatalEnabled();
23        }
24        catch (Throwable t)
25        {
26            t.printStackTrace();
27            fail("Exception thrown: " + t);
28        }
29    }
```

Quelltext 4.3: Auszug aus BasicOperationsTestCase.java (Apache Commons Logging [Fou13b])

```
1     public void testPristineDecorated() {
2
3         setUpDecorated("DecoratedLogger");
4         checkDecorated();
5
6     }
7     protected void checkDecorated() {
8
9         assertNotNull("Log exists", log);
10        assertEquals("Log class",
11                     "org.apache.commons.logging.simple.DecoratedSimpleLog",
12                     log.getClass().getName());
13
14        // Can we call level checkers with no exceptions?
15        assertTrue(!log.isDebugEnabled());
16        assertTrue(log.isErrorEnabled());
17        assertTrue(log.isFatalEnabled());
18        assertTrue(log.isInfoEnabled());
19        assertTrue(!log.isTraceEnabled());
20        assertTrue(log.isWarnEnabled());
21
22        // Can we retrieve the current log level?
23        assertEquals(SimpleLog.LOG_LEVEL_INFO, ((SimpleLog)
              log).getLevel());
24
25        // Can we validate the extra exposed properties?
26        assertEquals("yyyy/MM/dd HH:mm:ss:SSS zzz",
27                     ((DecoratedSimpleLog)
                        log).getDateTimeFormat());
28        assertEquals("DecoratedLogger",
29                     ((DecoratedSimpleLog) log).getLogName());
30        assertTrue(!((DecoratedSimpleLog) log).getShowDateTime());
31        assertTrue(((DecoratedSimpleLog) log).getShowShortName());
32
33    }
```

Quelltext 4.4: Auszug aus DefaultConfigTestCase.java (Apache Commons Logging [Fou13b])

4.2.2. Äquivalente Anweisungen

Es gibt eine Vielzahl an Anweisungen, die sich einfach in eine implementierungsneutrale Form verwandeln lassen. Beispielsweise lassen sich `for` Schleifen in verhaltensäquivalente `while` Schleifen umwandeln. Andere, einfach auf eine neutrale Form umwandelbaren Anweisungen sind: Prä-/Postinkrement sowie -dekrement, zusammengesetzte Zuweisungen wie `i += 1` und `switch` Anweisungen. Ein Beispiel für eine Reihe solcher Umwandlungen wird in Abbildung 4.1 gezeigt.

```
for(int i = 0; i < 20;
    i = ++i * 2, i -=
    1) {


  switch(i) {
  case 1:
    System.out.println
      ("i == 1");
  case 2:
  case 3:
    System.out.println
      ("i < 4");
    break;
  case 0:
    System.out.println
      ("s");
    break;
  case 7:
    System.out.println
      ("i == 7");
    break;
  default:
    System.out.println
      (i);
  }


}
```

Quelltext 4.5: Original

```
{
  int i = 0;
  while(i < 20) {
    if(i == 0) {
      System.out.println
        ("s");
    } else if(i == 2
      || i == 3) {
      System.out.println
        ("i < 4");
    } else if (i == 1)
       {
      System.out.println
        ("i == 1");
      System.out.println
        ("i < 4");
    } else if (i == 7)
       {
      System.out.println
        ("i == 7");
    } else {
      System.out.println
        (i);
    }
    i = i + 1;
    i = i * 2;
    i = i - 1;
  }
}
```

Quelltext 4.6: Neutrale Form

Abbildung 4.1.: Beispiele einfacher Semantik neutraler Umandlungen

Diese Umwandlungen könnten dann vor jeglicher Auswertung auf den Quelltext angewandt und danach Jutci wie bisher ausgeführt werden. Zwar könnten damit einige bestimmte Fälle von Klonen einfach erkannt werden, doch keines der oben aufgelisteten Probleme wäre behoben. Daher wird dieses Verfahren in dieser Arbeit nicht angewandt.

4.2.3. Strukturneutrale Klonerkennung mit PDGs

Semantikneutrale Restrukturierung kann, wie in den Abschnitten 4.1.2 und 4.1.3 gezeigt wurde, zu einer erheblichen Diskrepanz zwischen intuitiver Ähnlichkeit und dem von Jutci zugewiesenen Ähnlichkeitswert führen. Diese Probleme können behoben werden, indem anstatt des Quelltextes die PDGs der Methoden als Grundlage für die Analyse verwendet werden.

Die Reihenfolge von Anweisungen hat auf den PDG einer Methode in der Regel keinen Einfluss, so lange die Semantik sich nicht ändert. (Der PDG kann sich verändern, wenn

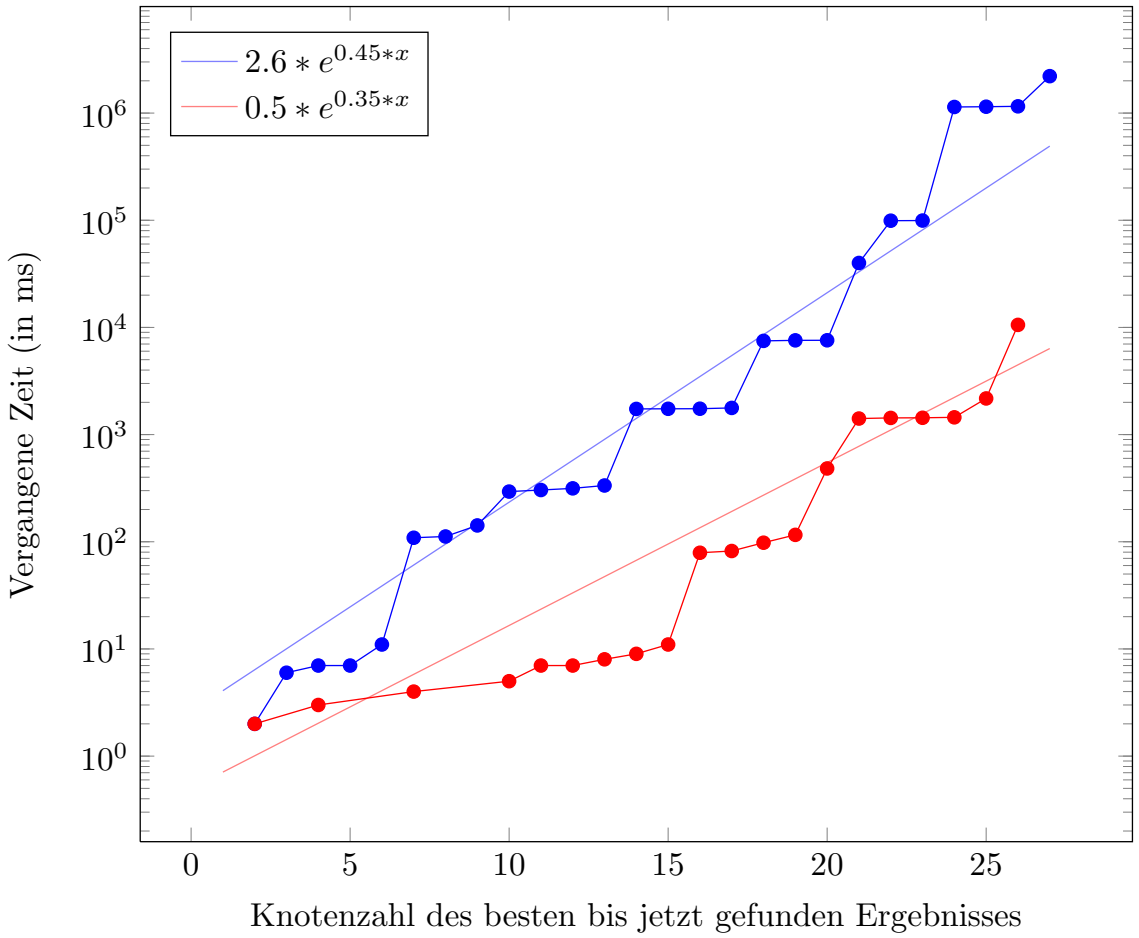

Abbildung 4.2.: Laufzeit der Testimplementierung

sich nicht erkennen lässt, dass eine Anweisung keinen Einfluss auf den Wert einer Variable hat, obwohl sie in dieser Anweisung verwendet wird.) Der Vergleich der PDGs von zwei Methoden anstelle des Quelltextes könnte also die eine Möglichkeit sein, Erkennungsprobleme, wie das aus Abschnitt 4.1.2, zu beheben.

PDGs eigenen sich auch zur Lösung der Missachtung ausgelagerter Quelltextfragmente durch Jutci. Da die Methodenaufrufe in PDGs durch Kanten abgebildet werden, können durch Traversieren dieser Kanten auch die Methodenrümpfe der ausgelagerten Methoden erfasst werden. Dabei dürfen nur Methoden untersucht werden, die tatsächlich ausgelagerte Methoden sind – Methodenaufrufe von getestetem Quelltext müssen explizit ignoriert werden.

Eine Möglichkeit, PDGs einzubinden, ist die oben beschriebene Isomorphiebestimmung. Diese anfangs sehr vielversprechend aussehende Methode, funktioniert allerdings nur für kleine Graphen in vertretbarer Zeit. Abbildung 4.2 zeigt, wie lange der Testalgorithmus gebraucht hat, um eine bessere Lösung als die bisher Beste zu finden.

Die abgebildeten gepunkteten Laufzeitverläufe das Algorithmus stammen von zwei Testläufen des Programms. Jeder der beiden Testgraphen hatte dabei 27 Knoten und wurde aus einem realen Testfall generiert. Für jeden der beiden PDGs suchte der Algorithmus einen isomorphen Teilgraphen zwischen dem PDG und einer Kopie dieses PDGs, so dass ein möglicher isomorpher Teilgraph immer der komplette Graph war. Anzumerken ist, dass es sich hier um eine logarithmische Skala handelt und der Unterschied zwischen dem roten und dem blauen Graphen mehr als Faktor 100 beträgt.

Die anderen Graphen zeigen die Approximation der jeweiligen Laufzeit durch eine Exponentialfunktion. Dabei stellt die Isomorphiebestimmung zwischen zwei identischen Graphen den schlechtesten Fall bezogen auf die Laufzeit des Algorithmus dar. Zwischen sehr verschiedenen Graphen kann der Algorithmus viele Möglichkeiten verwerfen, bei zwei gleichen Graphen hingegen kann dieser erst abbrechen, wenn eine Lösung gefunden wurde, die alle Knoten und Kanten enthält.

Die große Diskrepanz zwischen den beiden Graphen entsteht durch die unterschiedliche Struktur der PDGs. Der durch den roten Graphen präsentierte PDG ist sehr ähnlich zu dem dazugehörigen AST, das heißt jeder Knoten hat nur zu oder von den Knoten eine Kante, wenn es im AST mindestens eine Kante zwischen diesen beiden Knoten gibt. Der blaue Graph hingegen hat einige Kreuzverbindungen, wodurch der Algorithmus nicht so schnell ganze Zweige verwerfen kann und deutlich mehr Vergleiche durchführen muss.

Als Referenzwert ist anzumerken: Die meisten der in der Evaluation beachteten Testfälle haben zwischen 10 und 50 Knoten, der Mittelwert zwischen den beiden Approximationen würde also eine Laufzeit von 89 Tagen für den Vergleich von zwei identischen 50 Knoten großen PDGs vorhersagen. Selbst wenn die Laufzeit um einen Faktor 1000 reduziert werden könnte, also auf etwa 2 Stunden pro Vergleich, würde dies immer noch nicht ausreichen, um diese Methode in Betracht zu ziehen und zwar aus folgenden Gründen: Je nach Konfiguration der Vorfilter und Metriken werden etwa 8000 Durchläufe der finalen Auswertung von *Args4j* benötigt – ein Vergleich dürfte also höchstens 10 Sekunden dauern, damit der Algorithmus in unter 24 Stunden ausführbar wäre.

Obwohl eine isomorphiebasierte Klonbestimmung damit nicht praktikabel ist, können PDGs dennoch durch Analyse der Knoten und Kanten des Graphen zur Verbesserung der Klonerkennung verwendet werden. Wie PDGs eingesetzt werden können, um die oben (siehe 4.1.2 *ff*) beschriebenen Probleme zu beheben, wird im folgenden beschrieben.

4.3. Integration in Jutci

Wie in Abschnitt 2.1 erwähnt, arbeitet die Klonerkennung von Jutci in vier Stufen. Im Folgenden werden die Möglichkeiten erörtert, wie die Probleme der einzelnen Stufen durch Ersetzen oder Verändern der jeweiligen Stufe behoben werden können.

4.3.1. Vorfilter

Der Vorfilter betrachtet jede Methode einzeln ohne Vergleiche mit anderen Methoden anzustellen. Dabei sortiert der Vorfilter Methoden aus, die bestimmte Kriterien nicht erfüllen, der Anweisungszahlvorfilter beispielsweise verwirft Methoden die weniger als eine bestimmte Anzahl an Anweisungen haben. Hauptproblem dieses Vorfilters ist das in Abschnitt 4.1.3 beschriebene Problem des ausgelagerten Quelltextes. Durch diese Auslagerungen verändert sich die Methodenlänge, woraufhin der Vorfilter solche Methoden womöglich unerwünscht früh aussortiert. Dieser Vorfilter muss also so angepasst werden, dass er die Größe des PDG betrachtet, inklusive ausgelagerter Methoden. Alternativ hierzu könnten immer noch die Anweisungen gezählt und die Anweisungszahl ausgelagerter Methoden zu der Gesamtanweisungszahl hinzugefügt werden.

4.3.2. Wandler

Der Wandler modifiziert den Quelltext anhand vorher festgelegter Regeln. Zur Zeit wandelt der Wandler Testfälle nur auf eine Testrahmen-neutrale Form um. Da diese Stufe nicht direkt zur Auswertung beiträgt, hat sie auch keine Probleme hinsichtlich der Beachtung der Semantik. Der Wandler kann aber dazu verwendet werden semantisch äquivalente

Strukturen in eine einheitliche Form zu überführen (zusätzlich zu der derzeitig bestehenden Umwandlung auf eine Testrahmen-neutrale Form). Dabei können Umwandlungen, wie diese die in Abschnitt 4.2.2 beschrieben wurden, benutzt werden um den Einfluss einfacher syntaktischer Änderungen, die keine Bedeutung für die Semantik haben, zu verringern.

4.3.3. Metriken

In diesem Schritt werden Metriken verwendet die einfach vergleichbare Werte für einzelne Testmethoden erstellen und vergleichen. Die Metriken haben dabei das gleiche Problem wie der Vorfilter: Sie lassen sich leicht durch Quelltextauslagerungen beeinflussen. Zur Zeit gibt es zwei Metriken in Jutci die mithilfe von PDGs überarbeitet werden können:

Die Zusicherungsmetrik bestimmt, ob beide Methoden die gleichen Zusicherungen enthalten. Oftmals finden sich in den betrachteten Bibliotheken Methoden, die eine Art benutzerdefinierte Zusicherung darstellen. Diese benutzerdefinierten Zusicherungen bestehen oftmals aus einer Reihe „echter" Zusicherungen. Die Methode `checkDecorated` aus Quelltextauszug 4.4 stellt beispielsweise eine solche benutzerdefinierte Zusicherung dar. Durch Betrachtung der ausgelagerten Methoden können diese benutzerdefinierten Zusicherungen auf eine ähnliche Weise von der Zusicherungsmetrik erfasst werden wie „echte" Zusicherungen.

Eine andere Metrik, die durch PDGs verbessert werden kann, ist die Anweisungszahlmetrik. Diese bestimmt, ob Schleifenzahl, bedingte Anweisungszahl und Gesamtanweisungszahl zweier Methoden über einem bestimmten Schwellenwert liegen – dabei werden unterschiedliche Schleifentypen getrennt betrachtet. Auch hier kann eine Berücksichtigung ausgelagerter Methoden die Metrik verbessern, vor allem wenn ein Großteil der Testlogik sich in diesen ausgelagerten Methoden befindet.

Zusätzlich zu diesen, jetzt schon bestehenden Metriken können mit Hilfe von PDGs neue Metriken eingeführt werden:

Eine Knotenzahlmetrik, die die Anzahl jedes Knotentyps in einem Testfall bestimmt und den daraus resultierenden Vektor mit dem Vektor eines anderen Testfalls vergleicht. Obwohl die Anweisungszahlmetrik stark der Knotenzahlmetrik ähnelt, kann es vorteilhaft sein, diese Metriken getrennt zu halten, da dadurch kann ein höherer Ähnlichkeitswert für die Schleifen- und bedingte Anweisungszahl festgelegt werden kann. Schleifen und bedingte Anweisungen beeinflussen das Verhalten des Testfalls meist stärker – pro Anweisung – auf das Verhalten eines Testfalls haben als der Rest der Anweisungen.

Zusätzlich kann noch eine Kantenzahlmetrik eingeführt werden die ähnliche Vektoren wie die Knotenzahlmetrik bildet. Durch den Vergleich der Kantenzahlen können Testklonkandidaten aussortiert werden, die Variablen oder Anweisungen unterschiedlich behandeln. So kann mit dieser Metrik beispielsweise bestimmt werden, ob ein Testfall eine vergleichbare Anzahl an Variablen liest und schreibt.

4.3.4. Finale Auswertung

Die derzeitige verwendete Methode für die finale Auswertung ist die Bestimmung der längsten gemeinsamen Teilsequenz. Die längste gemeinsame Teilsequenz ist dabei eine Abbildung von einem Testfall auf einen anderen, so dass folgende Regeln zutreffen:

- Eine Anweisung wird immer auf $0 - 1$ äquivalente Anweisungen abgebildet.
- Zwei Anweisungen sind äquivalent wenn ihre Typen übereinstimmen – Identifikatoren sind nicht relevant.
- Die Reihenfolge der Anweisungen muss sowohl im Original als auch in der Abbildung gleich sein (keine „verkreuzte" Abbildung von Anweisungen).

- Es handelt sich um die längste aller möglichen Abbildungen.

Diese Methodik eignet sich vor allem für „vollständige" Klone; d.h. wenn es sich um kopierten Quelltext handelt bei dem nur die Identifikatoren geändert wurden. Veränderungen wirken sich unterschiedlich auf diese Methode aus:

Veränderung	**Auswirkung**	**Problematik**
Löschungen und Hinzufügungen	Die Veränderung des Ähnlichkeitswerts ist relativ zur Methodenlänge	gering
Änderungen der Anweisungsreihenfolge	Senkt den Ähnlichkeitswert überproportional stark (siehe Abschnitt 4.1.2)	hoch
Auslagerung von Quelltext	Ähnlichkeitswert beachtet nur noch nicht ausgelagerten Quelltext (siehe Abschnitt 4.1.3)	hoch

Die Anfälligkeit auf Auslagerungen ließe sich relativ leicht lösen, wenn man ausgelagerte Methoden in die Erkennung mit einbeziehen würde. Das Problem des Einflusses der Anweisungsreihenfolge auf den Ähnlichkeitswert ist allerdings ein fundamentales Problem der längsten gemeinsamen Teilsequenz und lässt sich nur durch vollständiges Ersetzen dieser Stufe lösen. Eine gute Möglichkeit wäre die Bestimmung von isomorphen Teilgraphen zwischen zwei PDGs. Da aber die Isomorphiefindung wie oben beschrieben (siehe Abschnitt 4.2.3) sich als nicht praktikabel herausgestellt hat, wird sich die Veränderung der finalen Auswertung auf die Einbeziehung ausgelagerter Methoden beschränken.

5. Implementierung

Die Implementierung der in Analyse und Entwurf vorgestellten Konzepte hat zwei Komponenten: erstens die Implementierung der PDG-Erzeugung und zweitens die eigentliche Integration dieses neu entworfenen Programms in Jutci.

Die Implementierung der PDG Erzeugung wurde in dieser Arbeit getrennt von Jutci angefertigt, so dass die PDG-Erzeugung auch getrennt von Jutci verwendet werden kann. Die Programmbibliothek, die im Rahmen dieser Arbeit entwickelt wurde, heißt JSlice bzw. Java Slicing Library.

5.1. JSlice

JSlice ist ein Werkzeug das aus Java-Quelltext einen großen PDG erstellen und diesen manipulieren kann. Dieser große PDG besteht aus vielen einzelnen miteinander verbundenen PDGs und wird deshalb im Folgenden vernetzter PDG (engl. interconnected program dependency graph, IPDG) genannt.

JSlice liefert eine Reihe von Schnittstellen, um Java Quelltext in einem IPDG und damit in eine Menge darin enthaltener PDGs abzubilden. Die Algorithmen zur Manipulation und Auswertung von IPDGs und PDGs können mit jeglichem IPDG bzw. PDG verwendet werden, der diese Schnittstellen benutzt. Im Zuge dieser Arbeit wurde ein Algorithmus implementiert, der einen solchen PDG erzeugen kann. Der Rest dieser Arbeit bezieht sich immer auf JSlice in Verbindung mit diesem Algorithmus.

Die genaue Implementierung von JSlice wird in den nächsten Abschnitten beschrieben.

5.1.1. PDG-Erzeugung

Die PDG-Erzeugung von JSlice arbeitet auf Basis von ASTs, die von der *JDT/Core* Bibliothek [AJA+13] erzeugt werden. Diese eignen sich besonders zur Konstruktion von PDGs, da sie so konfiguriert werden, dass sie versuchen die *Bindings* aller Variablen, Methoden und Klassen aufzulösen.

Die *Bindings* sind eindeutige Identifikatoren, die für zwei unterschiedliche Knoten dann den gleichen Wert haben, wenn der Knoten sich auf die gleiche Variable, Methode oder Klasse bezieht. Dabei ist die Bibliothek auch in der Lage, den Kontext von Variablen zu beachten, so dass zum Beispiel auch lokale Variablen mit gleichem Namen aber anderer Methodenzugehörigkeit unterschieden werden können. Diese Bindings werden benötigt

um korrekt bestimmen zu können an welchen Stellen Variablen gelesen oder geschrieben werden.

Die von *JSlice* erzeugten Graphen verwenden intern die Programmbibliothek *Java Universal Network/Graph Framework* (JUNG) [OFN+13]. Mit Hilfe von JUNG können eine Vielzahl von Graphenarten modelliert und visualisiert werden. Außerdem liefert JUNG einige Algorithmen um diese Graphen zu manipulieren.

Um einen AST zu traversieren, stellt die *JDT/Core* Bibliothek die Besucherklasse `ASTParser` zur Verfügung, die für jeden Knotentyp eine überladene Version der Methoden `visit` und `endvisit` hat. Mit Hilfe der `accept` Methode kann ein `ASTParser` an einen Knoten übergeben werden. Die Standardimplementierung von `ASTParser` führt keine Operationen durch. Ihr Verhalten kann durch Überschreiben der `visit` und `endvisit` Methoden bestimmt werden. Wie die eben beschrieben Methoden im Zusammenhang stehen, wird in Algorithmus 1 beschrieben.

Eingabe : Knoten, Besucher
besuche_Kindknoten := Besucher.visit(Knoten);
wenn *besuche_Kindknoten* **dann**
  für alle *Knoten.Kindknoten* **tue**
    Kindknoten.accept(Besucher);
  Ende
Ende
Besucher.endVisit(Knoten);

Algorithmus 1: Besucher

JSlice verwendet je nach Kontext verschiedene Unterklassen von `ASTParser`. Dies ist notwendig, da zwei Knoten des AST vom gleichen Typ nicht immer durch zwei Knoten mit den selben Typen abgebildet werden. Dies ist beispielsweise bei Variablendeklarationsknoten (Klasse `SingleVariableDeclaration`) der Fall: Sobald JSlice einen Methodenknoten findet, werden die Parameterknoten mit dem Parameterbesucher (Klasse `ParameterVisitor`) abgehandelt, wohingegen alle Knoten, die sich im Rumpf der Methode befinden, mit dem allgemeinen Besucher (Klasse `GeneralVisitor`) abgehandelt werden. Der Parameterbesucher erstellt für Variablendeklarationsknoten Knoten mit dem Typ `INPUT`, der allgemeine Besucher hingegen Knoten mit dem Typ `DECLARE`. Eine Liste der Besucher und deren Aufgabe ist Abschnitt A zu entnehmen.

Zusätzlich zur Erstellung eines Knotens müssen bei manchen Knotentypen weitere Schritte durchgeführt werden. Für Zuweisungsknoten (Klasse `Assignment`) im AST muss beispielsweise bestimmt werden, ob der Variablenknoten zusätzlich zu der Schreibkante auch noch eine Lesekante benötigt. (Eine Lesekante ist immer dann erforderlich, wenn die Variable nicht nur geschrieben sondern auch gelesen wird. Dies ist immer dann der Fall, wenn es sich um eine zusammengesetzte Zuweisung (`+=`, `-=`, `*=` usw.) handelt.)

Sobald der IPDG erstellt ist können aus diesem für alle Klassen und Methoden einzelne PDGs extrahiert werden, dabei sind Übergänge in andere PDGs mit speziellen Kanten und Knoten repräsentiert.

Das UML Diagramm in Abbildung 5.1 zeigt, wie die einzelnen Komponenten von JSlice zusammenhängen.

5.1.2. IPDG und PGD Details

Bei dem von JSlice erzeugten IPDG handelt es sich um keinen echten SDG, das heißt es gibt zwar Kanten, die Methodenaufrufe repräsentieren, aber das genaue Verhalten der

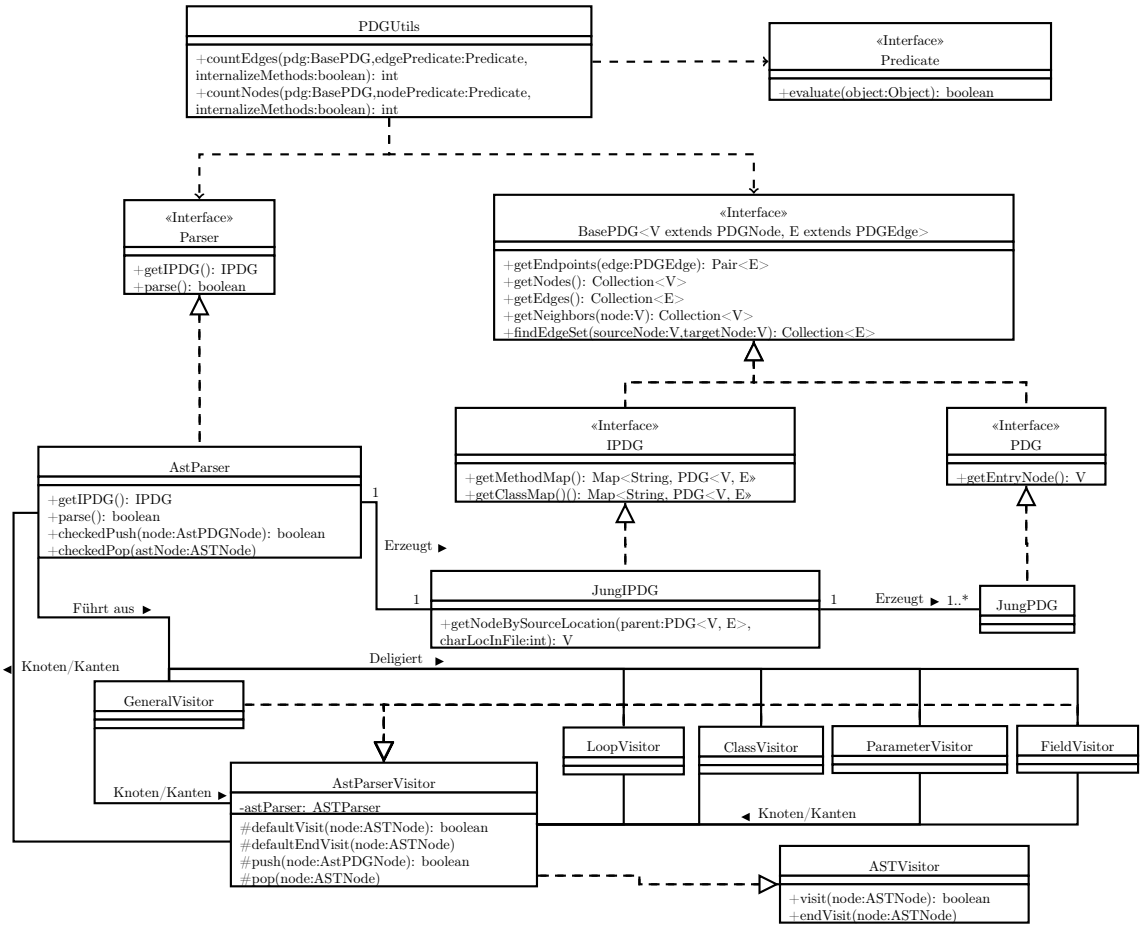

Abbildung 5.1.: IPDG Erzeugung und Auswertung

Kantentyp	Bedeutung
Wertkante	Quellknoten und Zielknoten referenzieren die gleiche Variable. Die Kante signifiziert, dass der im Quellknoten geschriebene Wert im Zielknoten gelesen wird.
Schreibkante	Der Quellknoten überschreibt die Variable des Zielknotens.
Lesekante	Der Wert des Quellknotens wird im Zielknoten verwendet.

Tabelle 5.1.: Variablenkanten

Methode wird nicht beachtet. Wie in Abschnitt 3.1.1 erwähnt, orientieren sich Struktur, sowie Kanten- und Knotentypen des IPDG an der Arbeit von Krinke [Kri01].

Da die PDGs dem zugehörigen AST ähnlich erscheinen sollen, hat jeder Knoten eine direkte Kontrollkante von seinem Vaterknoten. Direkte Kontrollkanten signalisieren, dass der Zielknoten nur ausgeführt wird, wenn auch der Quellknoten ausgeführt wird. Um die Kontrollkanten möglichst nahe an dem AST zu halten, sind Rücksprünge durch Spezialkanten abgebildet.

Die Einflüsse des Schreibens und Lesens werden durch verschiedene Kanten abgebildet, die Namen dieser Kanten und deren Bedeutung ist in Tabelle 5.1 aufgelistet.

Zusätzlich zu diesen Kantentypen gibt es noch die Aufrufskanten, die von dem aufgerufenen PDG auf den aufrufenden Knoten verweisen. Diese Kanten unterscheiden sich von allen anderen Kanten dadurch, dass sie zusätzliche Informationen über Eingabewerte der aufgerufenen Methode tragen.

Von den Knotentypen gibt es einen für jede Art von Anweisung in Java – die genaue Liste findet sich in der Implementierung. Es gibt nur einen Spezialknotentyp: den IPDG-Knoten der immer da eingesetzt wird wo eine Kanten in einen anderen PDG übergeht, er ersetzt also den jeweiligen Ziel- bzw. Quellknoten und hält eine Referenz zu dem jeweiligen Ziel-PDG.

Der PDG in Abbildung 5.2 zeigt den von JSlice erzeugten IPDG zu der Klasse in Quelltext 5.1.

```
1 public class M {
2   public int mal2(int i) {
3     return i * 2;
4   }
5 }
```

Quelltext 5.1: Die Beispielklasse M

5.2. Integration in Jutci

Dieser Abschnitt beschäftigt sich mit der Integration von JSlice in Jutci, analog zu Abschnitt 4.3.

5.2.1. Anweisungsbaum & AST

Jutci benutzt sowohl den AST der *JDT/Core* Bibliothek, als auch einen mit dessen Hilfe aufgebauten Anweisungsbaum. Der AST wird hauptsächlich für die Metriken eingesetzt, wohingegen der Anweisungsbaum für die Vorfilter und die finale Auswertung verwendet wird. Mit Hilfe von JSlice werden jetzt bei der Erstellung des Anweisungsbaums die ausgelagerten Methoden internalisiert.

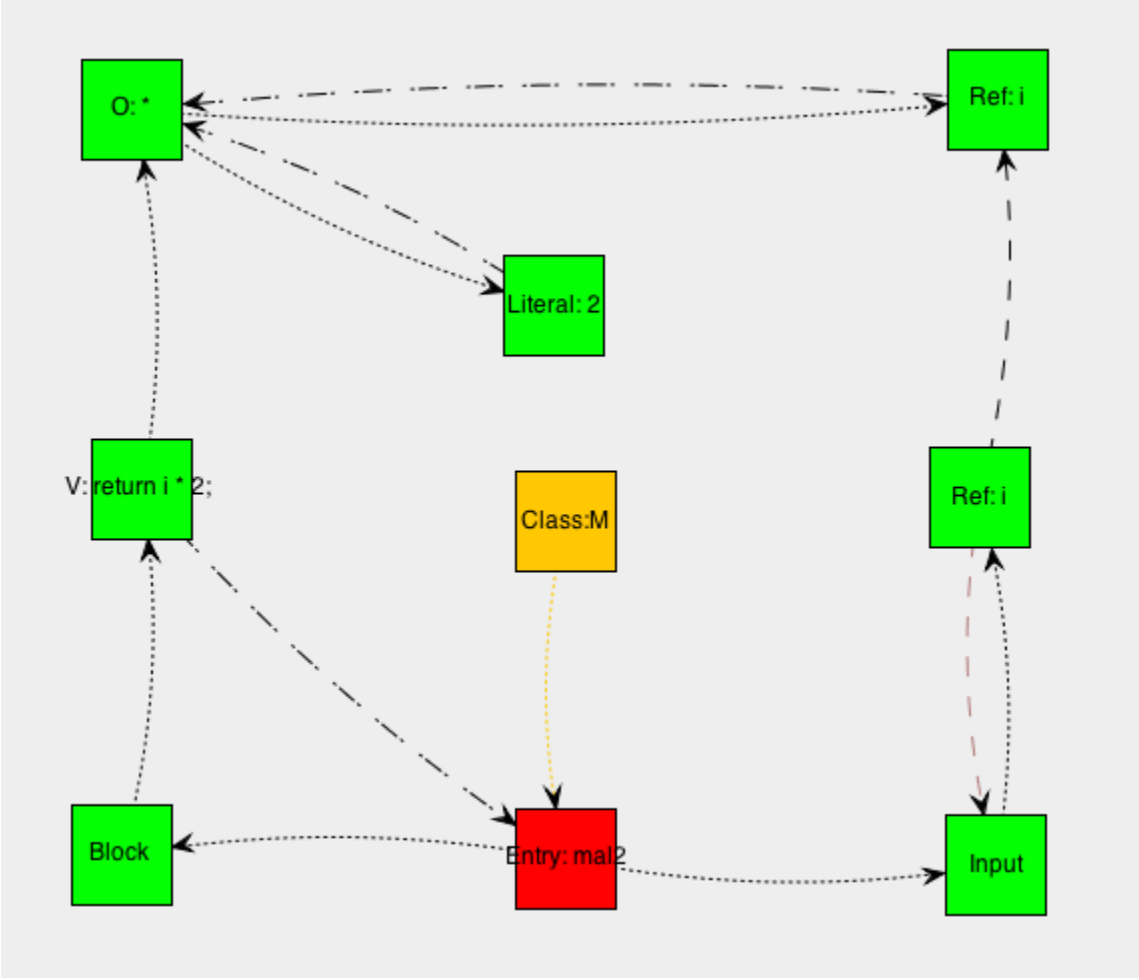

Abbildung 5.2.: IPDG der Klasse M

Diese Internalisierung funktioniert, indem der modifizierte Algorithmus für jeden gefundenen Methodenaufruf überprüft, ob es von diesem Knoten aus einen IPDG-Knoten gibt. Existiert solch ein Knoten wird eine Pseudo-Internalisierung durchgeführt, indem der Methodenrumpf komplett in die aufrufende Methoden kopiert wird. Dabei wird nur die erste Ebene des ausgelagerten Quelltextes betrachtet, ausgelagerte Methoden, die von ausgelagerten Methoden aufgerufen werden, werden somit nicht internalisiert. Da rekursive Methodenaufrufe keine IPDG-Knoten (es handelt sich ja immer noch um den selben PDG) erzeugen, werden diese nicht beachtet. Zu beachten ist allerdings, dass die bei der Internalisierung entstehenden Methoden in der Regel nicht ausführbar sind.

Abgesehen von der Anweisungsbaumkonstruktion wurden alle Algorithmen in denen der AST verwendet wird so umgestellt, dass sie jetzt die IPDGs und PDGs als Datenquelle verwenden.

5.2.2. Vorfilter

Die Vorfilter wurden nicht direkt verändert. Die oben beschrieben Veränderungen des Anweisungsbaums führen allerdings dazu, dass die Vorfilter jetzt auch ausgelagerte Methoden betrachten – dies betrifft derzeit allerdings nur den Anweisungszahlvorfilter. Mit Hilfe dieser Änderung sollten Testfälle mit ausgelagerten Quelltext nicht mehr frühzeitig aussortiert werden.

5.2.3. Metriken

Die Anweisungszahlmetrik wurde komplett überarbeitet und verwendet jetzt anstelle des AST einer Methode den PDG dieser Methode als Datenquelle. Dazu benutzt die Anweisungszahlmetrik nun die in Abbildung 5.1 dargestellte Methode `countNodes(...)`, die alle Knoten zählt, die sich in diesem PDG befinden und optional auch die aller aufgerufener Methoden. Außerdem ist es möglich, `countNodes(...)` einen Prädikatfilter mitzugeben der dafür sorgt, dass Knoten nur gezählt werden, wenn sie durch den Prädikatfilter nicht aussortiert wurden.

Zusätzlich zu der Zählmetrik wurden zwei weitere Metriken implementiert: eine Knotenzahlmetrik und eine Kantenzahlmetrik. Die Knotenzahlmetrik betrachtet im Gegensatz zur Anweisungszahlmetrik alle Knotentypen und nicht nur Schleifenknoten. Die alte Anweisungszahlmetrik wurde beibehalten da sie nur Schleifen zählt und dadurch ein höheres Gewicht auf die Gleichheit der Schleifenzahl gelegt werden kann.

Die Kantenzahlmetrik hingegen vergleicht die Kantenzahl der beiden verglichenen Methoden ohne direkte Abhängigkeitskanten zu beachten.

5.2.4. Finale Auswertung

Wie die Vorfilter wurde auch die finale Auswertung nicht direkt verändert. Trotzdem wurde das Verhalten der Auswertung indirekt modifiziert, durch Internalisierung der Methodenaufrufe wird die längste gemeinsame Teilsequenz nun anhand dieser neuen Gesamtmethode erstellt. Demzufolge ist bei Methoden, die ausgelagerten Quelltext enthalten, eine Abweichung der Klonmenge zu erwarten.

6. Evaluation

Um die Wirksamkeit der implementierten Methoden zu überprüfen, wird in diesem Kapitel eine Evaluation der modifizierten Klonfindung von Jutci vorgestellt.

Ziel dieser Evaluation ist es zu bestimmen, ob die neuen Klonerkennungsmethoden, die im Rahmen dieser Arbeit in Jutci implementiert wurden, zu einer Verbesserung der Klonerkennung geführt haben.

Die Daten der verschiedenen Versionen von Jutci werden hier mit dem F_β-Maß [ZZ11] verglichen. Das F_β-Maß bewertet die Güte eines Entscheiders mit einem Wert zwischen 0 und 1, wobei 1 der beste Wert ist. Dieser Entscheider bestimmt über die Zugehörigkeit zu einer Menge, was im Falle von Jutci die Identifikation als Klonpaar bedeutet. Zur Berechnung des F_β-Maßes werden folgende Kennzahlen benötigt, wobei die Nomenklatur der von Böhler entspricht [Bö13]:

$$F_\beta = (1 + \beta^2)\frac{p * a}{\beta^2 * p + a}$$

wobei

Präzision:	$p = \frac{r_+}{r_+ + f_+}$
Ausbeute:	$a = \frac{r_+}{r_+ + f_-}$
Zahl der richtig positiven Identifikationen:	r_+
Zahl der falsch negativen Identifikationen:	f_-
Zahl der falsch positiven Identifikationen:	f_+
Gewichtungswert:	β

Mit Hilfe des Gewichtungswertes β kann die Gewichtung von Präzision oder die Ausbeute festgelegt werden. Dabei gilt:

$\beta \in (0, 1)$	$\Rightarrow$	Präzision ist höher gewichtet als die Ausbeute
$\beta = 1$	$\Rightarrow$	Präzision und Ausbeute sind gleich gewichtet
$\beta \in (1, \infty)$	$\Rightarrow$	Präzision ist geringer gewichtet als die Ausbeute

Ein β von $0,5$ bedeutet beispielsweise die doppelte Gewichtung der Ausbeute und ein β von 2 die doppelte Gewichtung der Präzision. In dieser Evaluation sind Kennzahlen, die sich auf die modifizierte Klonerkennung beziehen, mit einem Stern ($\star$) gekennzeichnet.

Komponente	Einstellung	Wert
Testprofilfilter	Unerkannte Profile erlauben	Ja
	Ignorierte Tests einschließen	Nein
	Regulärer Ausdruck für Methodennamen	test.*
Zählmetrik	Minimale Knotenzahl	5
	Minimaler Ähnlichkeitswert	100%
Kanten- & Knotenzahlmetrik	Minimaler Ähnlichkeitswert	95%
	Minimale Kanten-/Knotenzahl	5
	Minimales Kanten-/Knotenzahlverhältnis	75%
Quelltextnormalisierung	Zusicherungen unterschiedlicher Rahmenwerke normalisieren	Ja

Tabelle 6.1.: Jutci Einstellungen

6.1. Vorgehensweise

Zur Erstellung dieser Evaluation mussten Einstellungen gefunden werden, die sich für alle Testrahmen verwenden lassen. In der Tabelle 6.1 sind die für die Evaluation verwendeten Einstellungen von Jutci abgebildet. Dabei sind deaktivierte Komponenten oder Einstellungen, die nicht relevant für das Ergebnis sind, nicht aufgelistet. Diese Einstellungen wurden empirisch ermittelt und sind so gewählt, dass ein gutes Verhältnis zwischen Präzision und Ausbeute erreicht wird, wobei die Präzision als der wichtigere Wert angesehen wurde.

Im Detail bedeutet dies:

- Der Testprofilfilter ist so eingestellt, dass er alle Testfälle zulässt – auch wenn Jutci den Testrahmen nicht bestimmen kann. Es wird allerdings mithilfe des regulären Ausdrucks verhindert, dass Methoden, die keine Testfälle sind, mit Testfällen verglichen werden, da dies zu Fehlidentifikationen führt – durch die Betrachtung von ausgelagerten Methoden bei der Klonfindung werden diese und der sie aufrufende Testfall oftmals als Klon erkannt.
- Die Zählmetrik wurde auf einen Ähnlichkeitswert von 100% gesetzt, um nur Testfälle mit der gleichen Zahl an Schleifen und bedingten Anweisungen zu finden. Dies ist darin begründet, dass diese Anweisungen in der Regel von enormer Wichtigkeit für das Verhalten des Testfalls sind, da ein Abweichen eine unterschiedliche Semantik nahelegt.
- Der Ähnlichkeitswert von 95% für die Kanten und Knoten wurde so festgelegt, dass bei den einzelnen Projekten mindestens 50% der manuell bestimmten Klone gefunden werden und die Präzision möglichst hoch ist. Das Kanten- und Knotenverhältnis wurde so gewählt, dass auch Klone zwischen sehr kleinen und leicht größeren Testfällen erkannt werden können.

Als Grundlage für die Evaluation dient die manuelle Identifikation von Testklonen, die Böhler in seiner Arbeit angefertigt hat. Da Böhler ausgelagerte Methoden und Methoden mit stark abweichender Syntax – zB. unterschiedlicher Anweisungszahl – nicht betrachtet hat, wurden die manuell bestimmten Testklonpaare im Rahmen dieser Evaluation von Hand in drei Schritten überarbeitet:

1. Entfernung von Testklonpaaren aus der Menge der von Hand bestimmten Klone, wenn es sich nicht um Klone unter Betrachtung ausgelagerter Methoden handelt.
2. Bestimmung der Testklone mit den oben angegeben Einstellungen von Jutci.

3. Analyse der vermeintlich falsch positiven Ergebnisse von Hand und Hinzufügen richtig positiver Ergebnisse zu der Menge manuell bestimmter Testklonpaare.

   Für diese Analyse wurden die Daten des jeweiligen Durchlaufs von Jutci mit einer Ähnlichkeitsschwelle von 30% verwendet. (Es wurden die Ergebnisse der alten und der modifizierten Klonerkennung betrachtet.)

Das Vorgehen bei der manuellen Bestimmung von Testklonen bedeutet, dass Präzision und Ausbeute tendenziell überbewertet werden. Der Verlust an Genauigkeit ist aber hinzunehmen, da die Bestimmung der Klone mit abnehmender Ähnlichkeit zunehmend subjektiv wird. Die Klonpaare, die bei einer minimalen Ähnlichkeit von 30% zusätzlich erkannt werden – im Vergleich zu denen bei einer minimalen Ähnlichkeit von 50% – können großteils von Hand nur schwer in richtig und falsch positive Klonpaare unterteilt werden.

Abschließend werden die Daten mit der Evaluation der ersten Version von Jutci verglichen. Um die Vergleichbarkeit zu gewährleisten, wurde die Auswertung der von Böhler erstellten Version von Jutci mit den neuen manuellen Daten wiederholt.

Die Evaluation wurde für jedes Projekt mit verschiedenen Mindest-Ähnlichkeitswerten durchgeführt. Alle anderen Einstellungen wurden unverändert beibehalten, da der minimale Ähnlichkeitswert der einzige Vergleichsparameter ist, der bei beiden Versionen von Jutci von äquivalenter Bedeutung ist. Es ist im Allgemeinen davon auszugehen, dass sich mit der individuellen Anpassung der Metrikeinstellungen für jedes Projekt bessere Ergebnisse erzielen lassen. (Dies gilt für beide Versionen von Jutci, aber besonders für die neue Version, da es mehr Metriken gibt und somit die Ergebnisse feiner gesteuert werden können.)

6.2. Auswertungsergebnisse

In diesem Abschnitt werden die Ergebnisse der einzelnen Projekte vorgestellt und analysiert. Zusätzlich sind die überarbeiteten manuellen Auswertungsdaten und die Größe jedes Projekts angegeben. Es wurden dafür die Kennzahlen von Böhler verwendet [Bö13]:

Gesamtzahl der Testfälle:	T_g
Zahl der manuell identifizieten Testklone:	K_m
Testfälle mit mindestens einem Klon:	K_i
Klonquote:	$K_q = \frac{K_i}{T_g}$

Die besten Werte jeder Spalte sind in den folgenden Tabellen fett markiert.

6.2.1. Args4j

T_g	95
K_m	709
K_i	87
K_q	91,58%

Args4j [Kaw13] hat eine ungewöhnlich hohe Klonquote und mit sehr kurzen und stilistisch ähnlichen Testfällen. Diese Ähnlichkeit sorgt für eine gute Ausbeute beider Versionen von Jutci. Allerdings zeigen die F_β-Werte in Tabelle 6.2, dass die modifizierte Klonerkennung einen besseren Kompromiss zwischen Ausbeute und Präzision liefert – vor allem bei einem niedrigen minimalen Ähnlichkeitswert. Anzumerken ist dennoch, dass das in Abschnitt 4.1.2 beschriebene Problem nicht direkt gelöst werden konnte – die neuen Metriken erlauben es aber, die minimale Ähnlichkeitsschwelle weit genug herunterzusetzen, um diese Fälle mit guter Präzision zu erkennen. (Das Beispiel

wird dennoch früher gefunden als zuvor, weil es zusätzlich zu der Restrukturierung eine identische ausgelagerte Methode enthält.)

min. Ähnlichkeit	$F^{\star}_{0,5}$	$F_{0,5}$	$F^{\star}_{1}$	F_1	$F^{\star}_{2}$	F_2
30%	**0,99**	0,87	**0,97**	**0,91**	**0,96**	**0,96**
40%	0,94	0,85	0,86	0,81	0,80	0,79
50%	0,93	0,89	0,84	0,82	0,77	0,77
60%	0,93	**0,92**	0,83	0,83	0,76	0,76
70%	0,62	0,61	0,39	0,39	0,29	0,28
80%	0,61	0,60	0,38	0,38	0,28	0,28
90%	0,57	0,57	0,35	0,35	0,25	0,25
95%	0,57	0,57	0,35	0,35	0,25	0,25
100%	0,55	0,55	0,33	0,33	0,23	0,24

Tabelle 6.2.: F_β-Maße von Args4j (Details siehe: Tabelle B.1)

6.2.2. Log4j

T_g	108
K_m	69
K_i	64
K_q	59, 26%

Wie die Tabelle 6.3 zeigt, sind die Ergebnisse der modifizierten Klonerkennung in ihren besten Werten nur marginal besser als die der Originalversion. An dem in Tabelle B.2 abgebildeten starken Präzisionsverlust, der bei der alten Klonerkennung bei abnehmendem minimalem Ähnlichkeitswert auftritt, wird der Einfluss der neuen Metriken ersichtlich: Da Log4j fast keine ausgelagerten Methoden in den Testfällen besitzt, verhalten sich die beiden finalen Auswertungs-Versionen praktisch gleich, die neuen Metriken verhindern allerdings einen starken Abfall der Präzision mit abnehmender Ähnlichkeitsschwelle.

min. Ähnlichkeit	$F^{\star}_{0,5}$	$F_{0,5}$	$F^{\star}_{1}$	F_1	$F^{\star}_{2}$	F_2
30%	0,76	0,33	0,81	0,44	0,87	0,66
40%	0,84	0,52	0,86	0,64	0,89	0,81
50%	0,91	0,74	**0,91**	0,81	**0,90**	**0,90**
60%	**0,94**	0,88	0,89	**0,86**	0,85	0,84
70%	0,92	**0,90**	0,85	0,81	0,78	0,73
80%	0,88	0,88	0,74	0,74	0,64	0,64
90%	0,82	0,82	0,64	0,64	0,53	0,53
95%	0,76	0,76	0,56	0,56	0,44	0,44
100%	0,72	0,74	0,51	0,53	0,39	0,41

Tabelle 6.3.: F_β-Maße von Log4j (Details siehe: Tabelle B.2)

6.2.3. Apache Commons Logging

T_g	77
K_m	37
K_i	49
K_q	63, 64%

Die manuelle Auswertung der Apache Commons Logging Bibliothek [Fou13b] musste relativ stark korrigiert werden, da sehr viele vermeintliche Testklonpaare in der manuellen Identifikation nach Betrachtung der ausgelagerten Methoden nicht mehr als Testklonpaar eingestuft werden konnten. Viele dieser Testfälle mit ausgelagerten Methoden bestehen nur aus dem Aufruf dieser Methoden und deren Parametern und wurden deshalb von der alten Jutci-Version fälschlicherweise als Klone erkannt. Dadurch ist die Präzision der alten Klonerkennung sehr viel schlechter als die der neuen, die

bei diesem Projekt die meisten Klone mit Quelltextauslagerungen richtig erkennt, was sich auch in den F_β-Werten aus Tabelle 6.4 widerspiegelt.

min. Ähnlichkeit	$F^\star_{0,5}$	$F_{0,5}$	$F^\star_1$	F_1	$F^\star_2$	F_2
30%	0,74	0,25	0,79	0,35	0,85	0,57
40%	0,78	0,30	0,82	0,41	0,86	0,64
50%	0,89	0,31	**0,88**	0,42	**0,87**	0,65
60%	0,90	0,51	0,88	0,59	0,86	0,71
70%	0,90	0,55	0,86	0,62	0,82	0,71
80%	**0,91**	**0,57**	0,84	**0,64**	0,77	**0,72**
90%	0,88	0,54	0,78	0,59	0,70	0,66
95%	0,85	0,54	0,72	0,58	0,63	0,64
100%	0,74	0,43	0,54	0,44	0,42	0,46

Tabelle 6.4.: F_β-Maße von Apache Commons Logging (Details siehe: Tabelle B.3)

6.2.4. Apache Commons Email

T_g	118
K_m	156
K_i	73
K_q	61,86%

Die modifizierte Klonerkennung hat für das Apache Commons Email [Fou13a] Projekt in dieser Evaluation eine vergleichsweise schlechte Ausbeute. Das liegt hauptsächlich an Klonpaaren, in denen der eine Testfall ein Array an das getestete Objekt und ein zweiter Testfall die Elemente des Arrays in einer Schleife einzeln übergibt. Diese Klonpaarkandidaten werden durch jede der drei in Tabelle 6.1 aufgelisteten Metriken verworfen und würden erst durch niedrigere minimale Ähnlichkeitswerte erkannt werden. Trotz dieser Probleme kann die modifizierte Klonerkennung bessere F_β-Werte (siehe Tabelle 6.5) erzielen als die alte – solange die Ausbeute nicht höher gewichtet wird als die Präzision.

min. Ähnlichkeit	$F^\star_{0,5}$	$F_{0,5}$	$F^\star_1$	F_1	$F^\star_2$	F_2
30%	0,87	0,47	0,82	0,59	0,78	0,78
40%	**0,90**	0,67	**0,83**	0,76	**0,78**	**0,89**
50%	0,88	0,72	0,75	**0,77**	0,65	0,81
60%	0,86	0,80	0,71	0,75	0,61	0,71
70%	0,85	**0,83**	0,69	0,74	0,59	0,67
80%	0,80	0,79	0,62	0,59	0,50	0,48
90%	0,75	0,75	0,55	0,54	0,43	0,43
95%	0,72	0,72	0,51	0,51	0,39	0,39
100%	0,19	0,19	0,09	0,09	0,06	0,06

Tabelle 6.5.: F_β-Maße von Apache Commons Mail (Details siehe: Tabelle B.4)

6.2.5. JBoss Web

T_g	129
K_m	1002
K_i	110
K_q	85,27%

Die Evaluation des JBoss Web [JBo12] Projekts zeigt zwar eine leichte Verbesserung der Ergebnisse, allerdings schneiden beide Jutci-Versionen hier so gut ab, dass diese Verbesserung sich kaum in den in der Tabelle 6.6 abgebildeten F_β-Werten widerspiegelt. Anzumerken ist allerdings, dass das F_β-Maß die Anzahl der falsch positiven bzw. negativen Klonpaare verschleiert und so selbst ein kleinster Unterschied in den F_β-Maßen eine große absolute Änderung der Ergebnisse bedeutet.

min. Ähnlichkeit	$F^\star_{0,5}$	$F_{0,5}$	$F^\star_1$	F_1	$F^\star_2$	F_2
30%	0,98	0,94	0,99	0,95	0,99	0,97
40%	0,99	0,96	0,99	0,97	1,00	0,98
50%	0,99	0,98	1,00	0,98	**1,00**	0,98
60%	**1,00**	**0,99**	**1,00**	**0,99**	1,00	**0,98**
70%	0,98	0,98	0,96	0,95	0,94	0,92
80%	0,98	0,98	0,96	0,95	0,94	0,92
90%	0,98	0,98	0,96	0,95	0,94	0,92
95%	0,98	0,98	0,96	0,95	0,94	0,92
100%	0,43	0,98	0,23	0,94	0,16	0,91

Tabelle 6.6.: F_β-Maße von JBoss Web (Details siehe: Tabelle B.5)

6.2.6. Postgresql

T_g	439
K_m	269
K_i	132
K_q	30,07%

Bei PostgreSQL [HCJJ12] zeigen beide Versionen von Jutci mit abnehmendem minimalen Ähnlichkeitswert einen – verglichen mit den anderen Projekten – starken Abfall der Präzision. Dies ist hauptsächlich darin begründet, dass es sich bei dieser Programmierbibliothek um einen Datenbankschnittstelle handelt, weswegen große Teile der Testlogik durch Zeichenketten oder andere Methodenparameter bestimmt werden. Da Jutci immer nur den Typ und nicht den Inhalt einer Variable betrachtet, kommt es zu diesem Präzisionsverlust. Dieser kann in der modifizierten Klonerkennung allerdings teilweise abgefedert werden, da die Kanten der PDGs Aufschluss über die Verwendung von Variablen geben und so zumindest Variablen unterschieden werden können, wenn sie unterschiedlich oft geschrieben bzw. gelesen werden.

min. Ähnlichkeit	$F^\star_{0,5}$	$F_{0,5}$	$F^\star_1$	F_1	$F^\star_2$	F_2
30%	0,32	0,09	0,43	0,13	0,65	0,27
40%	0,41	0,18	0,52	0,26	0,69	0,45
50%	0,56	0,33	0,63	0,42	**0,73**	0,59
60%	0,68	0,54	0,69	0,59	0,70	0,64
70%	**0,79**	**0,72**	**0,73**	**0,68**	0,67	**0,65**
80%	0,73	0,68	0,57	0,54	0,47	0,45
90%	0,72	0,65	0,52	0,48	0,41	0,39
95%	0,62	0,54	0,39	0,36	0,29	0,27
100%	0,35	0,40	0,18	0,25	0,12	0,18

Tabelle 6.7.: F_β-Maße von PostgreSQL (Details siehe: Tabelle B.6)

6.2.7. Guava und Derby

Die manuelle Auswertung von Guava [Goo13] und Derby [Fou12] ist wegen der Größe der Projekte praktisch unmöglich, was sich auch in der Anzahl der gefundenen möglichen Klonpaare (siehe Tabelle 6.8) widerspiegelt. Eine stichprobenartige Analyse der Ergebnisse suggeriert bei beiden Projekten zwar eine verhältnismäßig gute Präzision, allerdings kann keine definitive Aussage über die Güte der Ergebnisse getroffen werden. Die Evaluation dieser Projekte hat allerdings gezeigt, dass die Laufzeit der PDG-Konstruktion bei großen Projekten überproportional ansteigt. So dauerten das Importierten des Guava Projekts in Jutci und die anschließende Analyse über 8 Stunden, wohingegen für die selben Arbeitsschritte mit der alten Version von Jutci auf der gleichen Maschine unter 15 Minuten benötigt wurden.

minimaler Ähnlichkeitswert	Guava*	Derby*	Guava	Derby
30%	54044	18771	134581	76959
40%	48700	15036	84717	32228
50%	44821	12698	63495	19112
60%	13887	10707	15879	10638
70%	11850	7697	12451	8404
80%	10968	6271	11641	7714
90%	10102	5024	10602	7014
95%	9951	4555	10525	6895
100%	8324	2156	9088	6105

Tabelle 6.8.: Anzahl von Jutci erkannter potenzieller Klone

6.3. Fazit

In der Abbildung 6.1 und den folgenden Abbildungen sind die oben beschrieben Ergebnisse graphisch dargestellt (die Legende befindet sich in Abbildung 6.7). Wie deutlich zu erkennen ist, ist die modifizierte Klonerkennung meist mindestens genauso gut wie die alte. Die Ausbeute liegt zwar in der Regel unter der vorherigen, doch kann dadurch die Präzision gesteigert werden. Dies bedeutet, dass bei einem Ausbeutungswert, der von beiden Versionen von Jutci erreicht werden kann, die Präzision der modifizierten Klonerkennung mindestens genauso gut, meist jedoch besser ist. Damit konnte das Ziel der Verbesserung der Präzision bei gleichbleibender Ausbeute, das in Abschnitt 4.1 beschrieben wurde, erreicht werden, und die modifizierte Klonerkennung kann die alte ablösen, sofern eine Echtzeitauswertung nicht benötigt wird. Es ist allerdings davon auszugehen, dass diese modifizierte, wie auch die alte Klonerkennung für große Projekte schlechter funktioniert als für kleine. Dies ist darin begründet, dass je größer das Projekt wird, desto wahrscheinlicher ist es, dass die Metriken gleiche Werte für semantisch vollkommen unterschiedliche Testfälle berechnen.

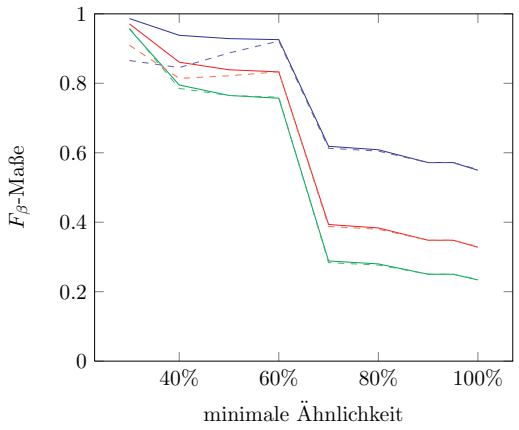

Abbildung 6.1.: Args4j

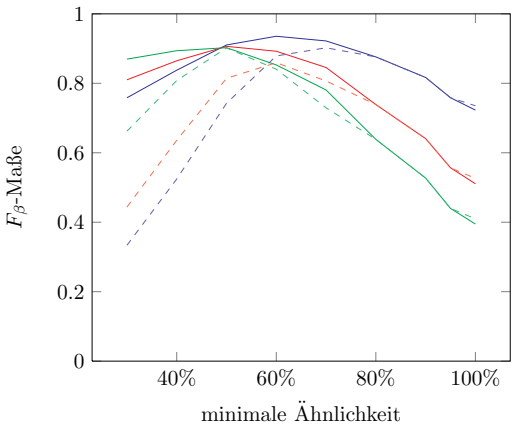

Abbildung 6.2.: Log4j

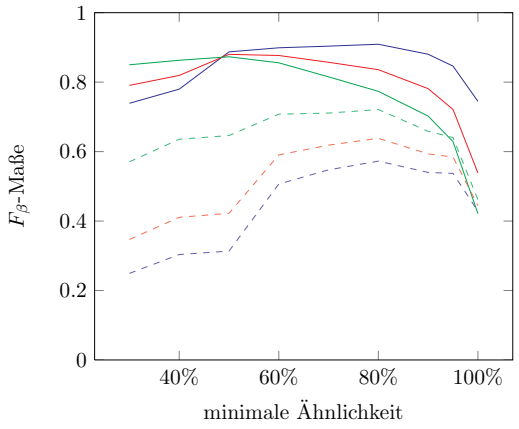

Abbildung 6.3.: Commons Logging

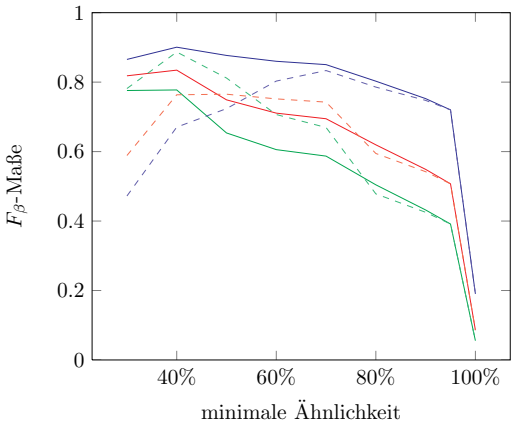

Abbildung 6.4.: Commons Mail

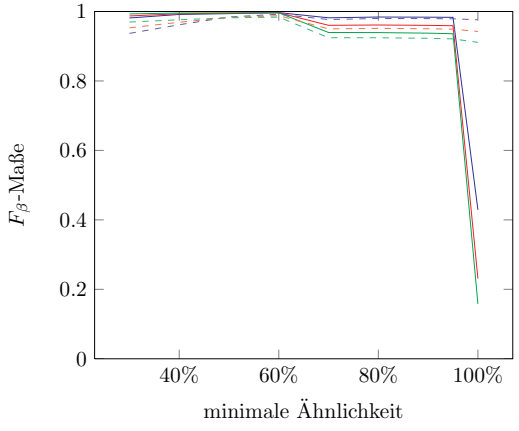

Abbildung 6.5.: JBoss Web

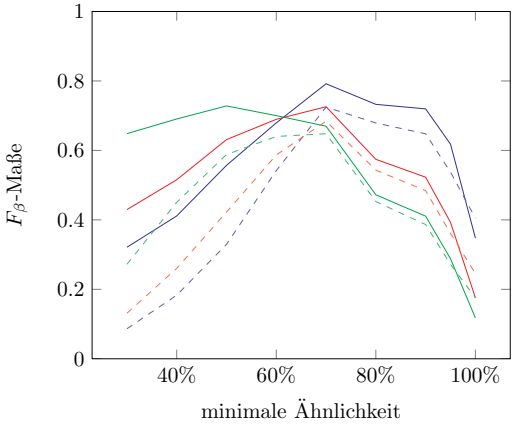

Abbildung 6.6.: PostgreSQL

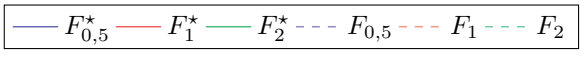

Abbildung 6.7.: Legende

7. Zusammenfassung und Ausblick

Im Rahmen dieser Arbeit wurden die bestehenden Probleme der Klonerkennung von Jutci analysiert. Dabei wurden Quelltextauslagerungen und Restrukturierungen des Quelltextes als zwei Hauptprobleme identifiziert, die zu Erkennungsfehlern durch Jutci führen und somit die Präzision bzw. Ausbeute der Klonerkennung negativ beeinflussen.

Um die Präzision zu steigern, wurden eine Reihe von Methoden vorstellt, mit deren Hilfe aus der Syntax des Quelltextes Rückschlüsse auf die Semantik getroffen werden können. Aus diesen Methoden wurde eine PDG basierte Klonerkennung, bestehend aus Metriken und Quelltextinternalisierung, abgeleitet. Die PDG-Erzeugung wurde in einer eigenständigen Programmbibliothek implementiert, die von Jutci zur Auswertung verwendet wird. Die mit Hilfe der PDGs erstellten Metriken versuchen die Semantik des Testfalls in Vektoren zu abstrahieren, die Kanten und Knoten zählen. Anhand dieser Vektoren können Klonkandidaten mit unterschiedlichem Datenfluss oder abweichender Knotenzusammensetzung aussortiert werden.

In der Evaluation wurde gezeigt, dass die Güte der Ergebnisse mit den neu implementierten Methoden verbessert werden kann. Meist bedeutet dies, dass die Präzision bei gleichbleibender Ausbeute gesteigert wird. Dazu wird das Problem der Quelltextauslagerungen, also die Extraktion mehrmals verwendeter Testlogik in neue Methoden, durch Reinternalisierung derselben gelöst.

Bei der Restrukturierungsproblematik handelt es sich um Erkennungsschwierigkeiten der semantischen Analyse aufgrund von Restrukturierungen des Quelltextes, wie beispielsweise Änderungen der Anweisungsreihenfolge. Diese konnte nur indirekt dadurch gelöst werden, dass die Klonerkennung auch bei niedrigen minimalen Ähnlichkeitswerten noch eine gute Präzision erreicht.

Außerdem hat die Evaluation gezeigt, dass die modifizierte Klonerkennung ein deutlich schlechteres Laufzeitverhalten für große Projekte aufweist, weswegen diese dort nicht für die Echtzeitanalyse verwendet werden kann.

Um die Auswertung zu beschleunigen, muss die PDG-Konstruktion verbessert oder die Größe der Eingabe reduziert werden, da die PDG-Konstruktion mehr als lineare Zeit braucht. Die Eingabegröße der IPDG-Konstruktion könnte kleingehalten werden, indem die Gesamtmenge der Eingabedateien in Teilmengen unterteilt wird, bei denen es am wahrscheinlichsten ist, Klone innerhalb dieser Teilmenge zu finden. Solche Teilmengen könnten mithilfe von latenter semantischer Analyse [MM00] durch die Gruppierung von Dateien mit ähnlichem Thema bestimmt werden.

Abgesehen von der Laufzeitverbesserung könnten weiter Fortschritte in der Klonanalyse erfolgen. Vor allem die finale Auswertung könnte durch eine andere Methode ersetzt werden, die die Semantik besser beachtet. Um beispielsweise das Restrukturierungsproblem direkt zu beheben, könnte die Klonerkennung auf der Basis von Teilgraphenisomorphie implementiert werden. Zwar ist die vollständige Isomorphiefindung, wie in Abschnitt 4.2.3 gezeigt wurde, zumindest nicht auf triviale Weise in polynomialer Zeit zu lösen, es kann dennoch andere Möglichkeiten geben, die PDGs in die finale Auswertung mit einzubeziehen. Entweder muss dazu ein hinreichend schneller und skalierender Algorithmus gefunden werden, der isomorphe Teilgraphen zwischen zwei beschrifteten Graphen bestimmen kann, oder ein Ergebnis muss heuristisch hergeleitet werden. Dazu kann ein ähnlicher Ansatz wie der von Gabel et al. [GJS08] verwendet werden, um das Problem der Isomorphiefindung auf ein einfacheres Problem zu überführen. Alternativ dazu können die PDGs unabhängig von der Isomorphiefindung mit in die Klonerkennung einfließen, beispielsweise durch weitere Metriken.

Des Weiteren sind viele Testfälle durch die Werte der in ihnen verwendeten Variablen bestimmt. Ohne die Analyse dieser Parameter kommt es hier gehäuft zu Fehlidentifikationen, vor allem bei sehr kurzen Testfällen. Da sich diese oftmals nur durch ihre Variablenwerte unterscheiden, muss womöglich der Datenfluss dieser Variablen innerhalb der getesteten Objekte betrachtet werden, um zu entscheiden, ob die Semantik des Testfalles ähnlich ist. Dies würde allerdings eine große Umstellung der Funktionsweise von Jutci bedeuten, da der getestete Quelltext zur Zeit nicht mit in die Analyse einfließt.

Literaturverzeichnis

[AJA+13] Arthanareeswaran, Jayaprakash ; Jain, Ayushman ; Audel, David ; Fusier, Frederic ; Arthanareeswaran, Jayaprakash ; Lanneluc, Jerome ; Keller, Markus ; Thomann, Olivier ; Mulet, Philippe ; Herrmann, Stephan ; Sankaran, Srikanth ; Harley, Walter ; Moller, Jesper ; Palat, Manoj: *JDT Core*. http://projects.eclipse.org/projects/eclipse.jdt.core. Version: 08 2013

[Bö13] Böhler, Johann: *Identifikation von Test-Klonen*, Karlsruher Institut für Technologie (KIT) – IPD Tichy, Bachelor's Thesis, März 2013. https://svn.ipd.kit.edu/trac/protest/wiki/Theses/boehler_ba

[CS03] Champin, Pierre-Antoine ; Solnon, Christine: Measuring the similarity of labeled graphs. In: *Case-Based Reasoning Research and Development*. Springer, 2003, S. 80–95

[DDL+90] Deerwester, Scott C. ; Dumais, Susan T. ; Landauer, Thomas K. ; Furnas, George W. ; Harshman, Richard A.: Indexing by latent semantic analysis. In: *JASIS* 41 (1990), Nr. 6, S. 391–407

[DM73] Dörfler, Willibald ; Mühlbacher, Jörg: *Graphentheorie für Informatiker*. Berlin [u.a.] : de Gruyter, 1973 (Sammlung Göschen ; 6016). http://digitool.hbz-nrw.de:1801/webclient/DeliveryManager?pid=2006116&custom_att_2=simple_viewer. – ISBN 3–11–003946–X

[Fou12] Foundation, Apache S.: *Derby*. http://db.apache.org/derby/. Version: 11 2012. – Version: 10.8.3.0

[Fou13a] Foundation, Apache S.: *Commons Email*. http://commons.apache.org/proper/commons-email/. Version: 01 2013. – Version 1.3

[Fou13b] Foundation, Apache S.: *Commons Logging*. http://commons.apache.org/proper/commons-logging/. Version: 03 2013. – Version 1.1.2

[GJS08] Gabel, Mark ; Jiang, Lingxiao ; Su, Zhendong: Scalable detection of semantic clones. In: *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on* IEEE, 2008, S. 321–330

[Goo13] Google: *Guava*. http://code.google.com/p/guava-libraries/. Version: 01 2013. – Version: 14.0-SNAPSHOT

[HCJJ12] Hall, Adrian ; Cramer, Dave ; Jurka, Kris ; Jowett, Oliver: *PostgreSQL*. http://jdbc.postgresql.org/. Version: 11 2012. – Version: 9.2-1002

[HRB90] Horwitz, Susan ; Reps, Thomas ; Binkley, David: Interprocedural slicing using dependence graphs. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12 (1990), Nr. 1, S. 26–60

[JBo12] JBoss: *JBoss Web*. http://www.jboss.org/jbossweb. Version: 07 2012. – Version: 7.7.0-SNAPSHOT

[Kaw13] Kawaguchi, Kohsuke: *Args4J*. http://args4j.kohsuke.org/. Version: 01 2013. – Version: 2.0.23-SNAPSHOT

[KDG07] Kuhn, Adrian ; Ducasse, Stéphane ; Gírba, Tudor: Semantic clustering: Identifying topics in source code. In: *Information and Software Technology* 49 (2007), Nr. 3, S. 230–243

[KDM+96] Kontogiannis, KA ; DeMori, Renator ; Merlo, Ettore ; Galler, M ; Bernstein, Morris: Pattern matching for clone and concept detection. In: *Automated Software Engineering* 3 (1996), Nr. 1-2, S. 77–108

[KH01] Komondoor, Raghavan ; Horwitz, Susan: Using slicing to identify duplication in source code. In: *Static Analysis*. Springer, 2001, S. 40–56

[Kri01] Krinke, Jens: Identifying similar code with program dependence graphs. In: *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on* IEEE, 2001, S. 301–309

[Kri04] Krinke, Jens: Advanced slicing of sequential and concurrent programs. In: *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on* IEEE, 2004, S. 464–468

[LT12] Landhausser, M ; Tichy, Walter F.: Automated test-case generation by cloning. In: *Automation of Software Test (AST), 2012 7th International Workshop on* IEEE, 2012, S. 83–88

[MFC] Mackinnon, Tim ; Freeman, Steve ; Craig, Philip: Endo-testing: unit testing with mock objects.

[MM00] Maletic, Jonathan I. ; Marcus, Andrian: Using latent semantic analysis to identify similarities in source code to support program understanding. In: *Tools with Artificial Intelligence, 2000. ICTAI 2000. Proceedings. 12th IEEE International Conference on* IEEE, 2000, S. 46–53

[OFN+13] O'Madadhain, Joshua ; Fisher, Danyel ; Nelson, Tom ; White, Scott ; Boey, Yan-Biao: *Java Universal Network/Graph Framework*. http://jung.sourceforge.net/. Version: 09 2013

[OO84] Ottenstein, Karl J. ; Ottenstein, Linda M.: The program dependence graph in a software development environment. In: *ACM Sigplan Notices* Bd. 19 ACM, 1984, S. 177–184

[RB89] Reps, Thomas ; Bricker, Thomas: *Illustrating interference in interfering versions of programs*. ACM, 1989

[RCK09] Roy, Chanchal K. ; Cordy, James R. ; Koschke, Rainer: Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. In: *Science of Computer Programming* 74 (2009), Nr. 7, S. 470–495

[Sil12] Silva, Josep: A vocabulary of program slicing-based techniques. In: *ACM Computing Surveys (CSUR)* 44 (2012), Nr. 3, S. 12

[Tip95] Tip, Frank: A survey of program slicing techniques. In: *Journal of programming languages* 3 (1995), Nr. 3, S. 121–189

[Wei81] Weiser, Mark: Program slicing. In: *Proceedings of the 5th international conference on Software engineering* IEEE Press, 1981, S. 439–449

[ZZ11] Zhang, Ethan ; Zhang, Yi: *F-Measure*. http://www.springerreference.com/docs/html/chapterdbid/63594.html. Version: 1 2011

Anhang

A. ASTVisitor-Implementierungen

Klasse	Aufgabe
ASTParserVisitor	• Basisklasse für alle anderen Besucher. • Stellt Methoden zur Standardbehandlung von AST-Knoten zur Verfügung. • Liefert die Schnittstelle zum Einfügen von Knoten und Kanten in den IPDG.
ClassVisitor	• Behandelt Enum- und Klassenknoten. (`EnumDeclaration`, `TypeDeclaration` und `AnonymousClassDeclaration`.)
FieldVisitor	• Behandelt Felddeklarationen. (`SingleVariableDeclaration` und `VariableDeclarationFragment`, wenn sie Kindknoten von `FieldDeclaration` sind.)
ParameterVisitor	• Wird von `ClassVisitor` für jeden Parameterknoten aufgerufen. (`SingleVariableDeclaration` und `VariableDeclarationFragment`.)
LoopVisitor	• Behandelt jegliche Art von Schleifenknoten und ordnet den Namen einer Schleife – sofern vorhanden – dem richtigen Knoten zu. (`EnhancedForStatement`, `LabeledStatement`, `WhileStatement` und `DoStatement`.)
GeneralVisitor	• Wird immer der `accept` Methode des Ursprungsknotes eines AST übergeben. • Alle anderen Besucher rufen diesen Besucher auf, wenn sie auf einen Knoten treffen für den sie keine `visit` Methode haben. • Behandelt alle anderen Knoten und delegiert bei Bedarf weiter. • Zusätzlich zu den oben nicht genanten Knoten werden die Knoten `SingleVariableDeclaration` und `VariableDeclarationFragment` behandelt.

B. Detailauswertungen

Die in den folgenden Tabellen verwendeten Kennzahlen werden in Kapitel 6 erläutert.

min. Ä.	$r_+^\star$	$f_+^\star$	$f_-^\star$	r_+	f_+	f_-	$p^\star$	$a^\star$	p	a
30%	**638**	2	**36**	**670**	129	**4**	99,69%	**94,66%**	83,85%	**99,41%**
40%	510	1	164	517	79	157	99,80%	75,67%	86,74%	76,71%
50%	487	0	187	493	33	181	**100,00%**	72,26%	93,73%	73,15%
60%	481	0	193	484	4	190	100,00%	71,36%	99,18%	71,81%
70%	168	0	518	165	0	521	100,00%	24,49%	**100,00%**	24,05%
80%	163	0	524	161	0	526	100,00%	23,73%	100,00%	23,44%
90%	145	0	543	145	0	543	100,00%	21,08%	100,00%	21,08%
95%	145	0	543	145	0	543	100,00%	21,08%	100,00%	21,08%
100%	135	**0**	553	136	**0**	552	100,00%	19,62%	100,00%	19,77%

Tabelle B.1.: Evaluation von Args4j

min. Ä.	$r_+^\star$	$f_+^\star$	$f_-^\star$	r_+	f_+	f_-	$p^\star$	$a^\star$	p	a
30%	**64**	24	6	**69**	172	1	72,73%	**91,43%**	28,63%	**98,57%**
40%	64	14	**6**	69	78	**1**	82,05%	91,43%	46,94%	98,57%
50%	63	6	7	68	29	2	91,30%	90,00%	70,10%	97,14%
60%	58	2	12	58	7	12	96,67%	82,86%	89,23%	82,86%
70%	52	1	18	48	1	22	98,11%	74,29%	97,96%	68,57%
80%	41	0	29	41	0	29	**100,00%**	58,57%	**100,00%**	58,57%
90%	33	0	37	33	0	37	100,00%	47,14%	100,00%	47,14%
95%	27	0	43	27	0	43	100,00%	38,57%	100,00%	38,57%
100%	24	**0**	46	25	**0**	45	100,00%	34,29%	100,00%	35,71%

Tabelle B.2.: Evaluation von Log4j

min. Ä.	$r_+^\star$	$f_+^\star$	$f_-^\star$	r_+	f_+	f_-	$p^\star$	$a^\star$	p	a
30%	**34**	14	4	**38**	143	0	70,83%	**89,47%**	20,99%	**100,00%**
40%	34	11	**4**	38	109	0	75,56%	89,47%	25,85%	100,00%
50%	33	4	5	38	104	**0**	89,19%	86,84%	26,76%	100,00%
60%	32	3	6	31	36	7	91,43%	84,21%	46,27%	81,58%
70%	30	2	8	30	29	8	93,75%	78,95%	50,85%	78,95%
80%	28	1	10	30	26	8	96,55%	73,68%	**53,57%**	78,95%
90%	25	1	13	27	26	11	96,15%	65,79%	50,94%	71,05%
95%	22	1	16	26	25	12	95,65%	57,89%	50,98%	68,42%
100%	14	**0**	24	18	**25**	20	**100,00%**	36,84%	41,86%	47,37%

Tabelle B.3.: Evaluation von Apache Commons Logging

min. Ä.	$r_+^\star$	$f_+^\star$	$f_-^\star$	r_+	f_+	f_-	$p^\star$	$a^\star$	p	a
30%	**117**	13	**39**	**156**	218	**0**	90,00%	**75,00%**	41,71%	**100,00%**
40%	116	6	40	155	95	1	95,08%	74,36%	62,00%	99,36%
50%	94	1	62	132	57	24	98,95%	60,26%	69,84%	84,62%
60%	86	0	70	106	20	50	**100,00%**	55,13%	84,13%	67,95%
70%	83	0	73	98	10	58	100,00%	53,21%	90,74%	62,82%
80%	70	0	86	66	0	90	100,00%	44,87%	**100,00%**	42,31%
90%	59	0	97	58	0	98	100,00%	37,82%	100,00%	37,18%
95%	53	0	103	53	0	103	100,00%	33,97%	100,00%	33,97%
100%	7	**0**	149	7	**0**	149	100,00%	4,49%	100,00%	4,49%

Tabelle B.4.: Evaluation von Apache Commons Mail

min. Ä.	$r_+^\star$	$f_+^\star$	$f_-^\star$	r_+	f_+	f_-	$p^\star$	$a^\star$	p	a
30%	**1000**	23	3	**984**	78	19	97,75%	**99,70%**	92,66%	**98,11%**
40%	1000	10	3	984	44	19	99,01%	99,70%	95,72%	98,11%
50%	1000	7	**3**	984	15	19	99,30%	99,70%	98,50%	98,11%
60%	999	3	4	984	5	**19**	99,70%	99,60%	99,49%	98,11%
70%	929	3	74	911	4	92	99,68%	92,62%	99,56%	90,83%
80%	928	0	75	910	0	93	**100,00%**	92,52%	**100,00%**	90,73%
90%	926	0	77	908	0	95	100,00%	92,32%	100,00%	90,53%
95%	924	0	79	906	0	97	100,00%	92,12%	100,00%	90,33%
100%	131	**0**	872	894	**0**	109	100,00%	13,06%	100,00%	89,13%

Tabelle B.5.: Evaluation von JBoss Web

min. Ä.	$r_+^\star$	$f_+^\star$	$f_-^\star$	r_+	f_+	f_-	$p^\star$	$a^\star$	p	a
30%	**265**	699	**5**	**258**	3402	**12**	27,49%	**98,15%**	7,05%	**95,56%**
40%	241	424	29	237	1313	33	36,24%	89,26%	15,29%	87,78%
50%	219	205	51	215	533	55	51,65%	81,11%	28,74%	79,63%
60%	191	93	79	184	173	86	67,25%	70,74%	51,54%	68,15%
70%	172	32	98	169	55	101	84,31%	63,70%	75,45%	62,59%
80%	114	13	156	110	25	160	89,76%	42,22%	81,48%	40,74%
90%	97	4	173	92	18	178	96,04%	35,93%	**83,64%**	34,07%
95%	66	0	204	63	16	207	**100,00%**	24,44%	79,75%	23,33%
100%	26	**0**	244	40	**16**	230	100,00%	9,63%	71,43%	14,81%

Tabelle B.6.: Evaluation von PostgreSQL