Institut für Softwaretechnologie
Abteilung Programmiersprachen
Universität Stuttgart
Universitätsstraße 38
D-70569 Stuttgart

Bachelorarbeit Nr. 262

Automatisierter Vergleich von Codeklonerkennungsergebnissen

Simon Gaiser

Studiengang:	Informatik
Prüfer:	Prof. Dr. Erhard Plödereder
Betreuer:	Torsten Görg
Beginn am:	15.10.2015
Beendet am:	15.04.2016
CR-Nummer:	D.2.7, D.3.4

Abstract

Als Codeklon bezeichnet man mehrere semantisch ähnliche Teile eines Programms. Diese können beispielsweise durch „Copy & Paste" entstehen und erschweren die Wartung von Software. Es gibt verschiedene Verfahren um Codeklone zu finden. Möchte man zwei Verfahren zur Erkennung von Codeklonen vergleichen, bietet es sich an, die bei gleicher Eingabe gefundenen Klonmengen zu vergleichen. Dabei ist man besonders an der Differenzmenge interessiert, also die Klone die nur durch eines der beiden Verfahren erkannt wurden. Da die Ergebnissmengen in der Regel groß sind, benötigt man eine automatische Vergleichsmöglichkeit. Diese Arbeit beschäftigt sich damit, wie man die Ausgabe verschiedener Klonerkenner automatisch vergleichen kann. Hierbei wird besonders darauf eingegangen, dass die Klone auf Text-, AST- oder PDG-Ebene dargestellt werden können und als Klonpaar oder Klongruppe vorliegen können. Des Weiteren werden die Klone so verglichen, dass auch ähnlich aber nicht identisch erkannte Klone gefunden und entsprechend behandelt werden. Außerdem beschäftigt sich diese Arbeit damit, wie Klone aus AST- und PDG-Ebene in Knotenmengen und diese wiederum in Quelltextfragmente umgewandelt werden können. Zum Schluss wird die erstellte Software noch zur Evaluation auf ein realitätsnahes Beispiel angewandt.

Inhaltsverzeichnis

Kapitel 1

Einführung

Als Codeklon bezeichnet man Quellcodestücke, die semantisch ähnlich sind. Codeklone können aus verschiedenen Gründen entstehen, beispielsweise durch das Kopieren von Programmstücken. Da Codeklone die Wartung von Software erschweren, ist es wünschenswert, Codeklone automatisiert zu erkennen. Es gibt verschiedene Verfahren zur Codeklonerkennung. Die Algorithmen können auf verschiedenen Darstellungsebenen des zu untersuchenden Programms arbeiten. Zum Beispiel direkt auf dem Quelltext, dem abstrakten Syntaxbaum (AST) oder dem Programmabhängigkeitsgraph (PDG).

Die Klone können je nach verwendetem Erkennungsverfahren in verschiedenen Darstellungsformen vorliegen. Zwar lassen sich die Darstellungen auf AST- oder PDG-Ebene auf Textstücke abbilden, allerdings gehen hierbei gegebenenfalls Informationen verloren. So könnten sich zum Beispiel zwei PDG-Klone darin unterscheiden, welche Datenabhängigkeiten Teil des Klones ist. Dies lässt sich nicht in einer quelltext-basierten Darstellung von Klonen abbilden. Von daher ist es wünschenswert, dass man zwei Verfahren, die auf der gleichen Darstellungsebene arbeiten, auch auf dieser vergleichen kann.

Des Weiteren ist zu beachten, dass zwei Klonerkennungsverfahren zwei sehr ähnliche, aber nicht identische Klone erkennen können. Beispielweise kann ein Verfahren eine kopierte Schleife als Klon erkennen und ein anderes Verfahren findet zusätzlich die weiter oben stehende Initialisierung der Schleifenvariable. Auch dies sollte bei dem Vergleich zweier Klonmengen erkannt werden.

Um verschiedene Verfahren zur Klonerkennung zu vergleichen, bietet es sich an, diese auf ein reales Softwareprojekt anzuwenden und dann die Ergebnisse zu vergleichen. Hierbei ist man besonders an der Klondifferenzmenge interessiert. Also an den Klonen, die nur eines der Verfahren erkannt hat. Da die zu vergleichende Menge in der Regel groß ist, ist dies in der Praxis fast nur automatisiert möglich. Diese Arbeit beschäftigt sich mit eben diesem automatischen Vergleich.

1.1 Aufgabenstellung

Ziel dieser Arbeit ist, es eine Software zu erstellen, die es ermöglicht, die Ergebnisse zweier Codeklonerkennungen automatisiert zu vergleichen. Hierbei sollen Klone auf verschiedenen Darstellungsebenen unterstützt werden: als Quelltextstück, als AST-Teilbaum und als PDG-Teilgraph. Außerdem sollen sowohl Klonpaare als auch Klongruppen verarbeitet werden können. Die Klone aus den beiden Eingabemengen liegen hierbei in der gleichen Darstellungsform vor. Dabei sollen auch ähnliche aber nicht identische Klone erkannt werden können.

Des Weiteren soll es möglich sein, Klone aus der AST-, und PDG-Darstellungen in Knotenmengen und diese in die Darstellung als Quelltextstück umzuwandeln.

Die Eingabe liegt dabei in Form der IML-Zwischendarstellung aus dem Programmanalyseframework Bauhaus vor.

1.2 Überblick

Nach der Einleitung wird zuerst auf einige Grundlagen eingegangen, auf welche dann im Folgenden aufgebaut wird. Dann wird die zu Beginn erarbeitete Spezifikation der Eingabedaten beschrieben und auf die Form der Ausgabe eingegangen. Das darauffolgende Kapitel stellt die erarbeiteten Verfahren zum Vergleich von Klonmengen vor. Darin wird auch das Verfahren zum Umwandeln zwischen den Klondarstellungen beschrieben. Im Anschluss wird die Gestaltung der Implementierung vorgestellt. Diese wird dann anhand einer Beispieleingabe evaluiert. Das letzte Kapitel fasst dann die Arbeit nochmal kurz zusammen und zeigt mögliche Verbesserungs- und Ergänzungsmöglichkeiten auf.

Kapitel 2

Grundlagen

2.1 Codeklone

Ein Codeklon (in dieser Arbeit in der Regel kurz als Klon bezeichnet) sind zwei oder mehrere Stücke eines Programms, welche semantisch ähnlich sind. Es gibt verschiedene Möglichkeiten, wie diese entstehen können. Die offensichtlichste Möglichkeit ist durch Kopieren und Einfügen („Copy & Paste"). Dabei kopiert der Entwickler ein Stück Quellcode, dass die gewünschte Funktionalität schon (teilweise) umsetzt und fügt es an einer anderen Stelle ein. Gegebenenfalls verändert er den kopierten Quellcode zusätzlich. Andere Möglichkeiten für die Entstehung von Codeklonen sind das mehrfache Schreiben gleicher, beziehungweise ähnlicher Funktionalitäten oder das Zusammenführen unterschiedlicher Quelltexte. (Siehe [1],[2])

Klone sind in der Regel unerwünscht, da sie die Wartung von Software erschweren können [1]. Findet man beispielsweise einen Fehler in einem Teil des Programms, wird er an der anderen Stelle nicht automatisch mit korrigiert. Da Klone in aller Regel aufgrund ihrer Entstehung nicht dokumentiert sind, heißt das, dass man den Fehler vermutlich mehrfach finden und beheben muss.

2.2 Klonerkennung

Aus den gerade beschriebenen Gründen, möchte man Klone automatisch finden können. Dazu gibt es verschiedene Ansätze. Diese unterscheiden sich sowohl im Vorgehen als auch darin, auf welche der Darstellungsebenen der Programmanalyse sie aufbauen.

Die Klonerkennung kann direkt auf Quelltextebene, beziehungsweise dem in Token umgewandelten Quelltext arbeiten. Das Verfahren nach Baker [4] beispielsweise wandelt jede Zeile auf Basis der darin vorkommenden Token in einen sogenannten prev-String um, der das Muster der Zeile unabhängig von Details wie Variablennamen abbilden soll. Mit Hilfe eines Suffix-Baumes werden dann gleiche Muster und somit ähnliche Zeilen gefunden und zu Klonen zusammengefasst.

Eine weitere Möglichkeit besteht darin, den bei der Programmanalyse oder Kompilierung erstellten abstrakten Syntaxbaum (abstract syntax tree; AST) zu Nutzen, um darin ähnliche Programmteile zu finden. Das Verfahren nach Baxter [5] beispielsweise sucht nach ähnlichen Teilbäumen in einem AST. Da ein paarweiser Vergleich aller Teilbäume sehr aufwändig wäre, wird eine Hashfunktion genutzt, die ähnliche Teilbäume auf den selben Wert abbildet. Damit müssen dann nur noch Teilbäume mit gleichem Hashwert verglichen werden.

Darüber hinaus kann man sich noch weitere Ergebnisse von Programmanalysen zu Nutze machen. Dazu arbeiten Verfahren, wie beispielsweise das nach Krinke [6], auf dem Programmabhängigkeitsgraph (program dependency graph; PDG). In diesem sind zusätzlich Daten- und Kontrollabhän-

gigkeiten abgebildet. Das Verfahren nach Krinke sucht nach Klonen, indem es ähnliche Teilgraphen des PDG identifiziert.

Wenn im Folgenden von Klon die Rede ist, ist in aller Regel das Ergebnis eines Klonerkennungsverfahren gemeint. Also das, was ein Klonerkennungsverfahren als Klon eingestuft hat. Ob es sich dabei tatsächlich um zwei Klonfragmente handelt, die man als Klon einstufen sollte („true positive"), ist dabei nicht sicher. So erzeugen zum Beispiel die meisten Verfahren, wenn sie „zu fein" eingestellt sind Klone, die nur kleine gemeinsame Teilausdrücke darstellen, die man aber nicht als Klon einstufen würde („false positive").

2.3 Klondarstellung

So wie die Klonerkennungen auf verschiedenen Darstellungsformen des Programms arbeiten, kann man auch einen Klon verschieden darstellen.

Zum Einen stellt sich die Frage, wie man ein Stück eines Programms darstellt. Dies lässt sich als Stück des Quelltextes, als Teilbaum eines ASTs oder als Teilgraph eines PDGs darstellen. Hierfür wird in dieser Arbeit der Begriff Klonfragment genutzt. Üblich ist auch die Bezeichnung Codefragment, was aber zum Beispiel im Falle eines PDGs nur bedingt zutreffend ist, da dieser auch Eigenschaften wie Datenabhängigkeiten abbildet.

Des Weiteren stellt sich die Frage, wie man Klonfragmente zu einem Klon zusammenfasst. Dabei gibt es primär zwei Varianten: Klonpaare und Klongruppen (teilweise auch Klonklasse genannt). Andere Varianten, wie die von Stefan Bellon in [2] beschriebene Klonrelationenmenge, werden nicht betrachtet. Wenn nicht näher spezifiziert, sind beide Varianten gemeint. Wenn ein Paar von zwei Klonen gemeint ist wird in dieser Arbeit teilweise der Begriff Tupel verwendet um die Verwechslung mit Klonpaar zu vermeiden.

Bei einem Klonpaar werden zwei Klonfragmente zu einem Paar zusammengefasst. Bei einer Klongruppe hingegen werden eine ganze Menge von Klonfragmenten, die einander ähnlich sind, zusammengefasst. Ob die Klonbeziehung transitive Eigenschaften hat, hängt von den konkreten Klonerkennungsverfahren ab [2].

2.4 Vergleich von Klonerkennungsverfahren

Durch die verschiedenen Ansätze zum Finden von Klonen stellt sich natürlich die Frage, wie man diese vergleichen kann. Dabei gibt es sowohl den Fall, dass man zwei Varianten eines Algorithmus vergleichen will, um eventuell Unterscheide zu finden, als auch, dass man zwei deutlich verschiedene Verfahren vergleichen möchte.

Hierbei bietet es sich an, eine realistische Eingabe einer Klonerkennung zu nehmen, also ein nicht triviales Softwareprojekt und die zu vergleichenden Algorithmen darauf auszuführen [2]. Damit erhält man zwei Klonmengen. Diese müssen nun verglichen werden, um beispielsweise Klone zu finden, die nur von einem Verfahren gefunden werden. Da die erkannten Klonmengen in der Regel groß sind, sollte dieser Vergleich automatisch möglich sein.

Genau dieser automatisierte Vergleich zweier Klonmengen ist Gegenstand dieser Arbeit. Hierbei ist die Diplomarbeit von Stefan Bellon zu erwähnen [2]. In dieser vergleicht er verschiedene Werkzeuge zur Erkennung von Codeklonen. Dazu hat er auch eine Software erstellt, um Mengen von Textklonen automatisch zu vergleichen. Diese Arbeit unterscheidet sich von der Arbeit Bellons darin, dass die Darstellung von Textklonen erweitert wird, sodass auch nicht zusammenhängende Quelltextstücke ein Klonfragment bilden können. Vor allem werden zusätzlich noch die Darstellungsformen von Klonen auf AST- und PDG-Ebene betrachtet. Allerdings betrachtet diese Arbeit nur Verfahren zum Vergleich von Klonmengen und untersucht damit nicht konkrete Klonerkennungswerkzeuge.

2.5 Bauhaus

Bauhaus ist ein an der Universität Stuttgart entwickeltes Programmanalyse-Framework [3]. Für diese Arbeit ist vor allem die dazugehörige Zwischendarstellung IML relevant. IML ist ein Graph, welcher zum Einen das zu analysierende Programm in Form eines, mit einigen Zusatzinformationen (wie zum Beispiel Quelltextreferenzen), angereicherten ASTs und zum Anderen weitere Programmanalyseergebnisse enthält.

IML besteht aus einigen nativen Datentypen und mit Hilfe einer einfachen Beschreibungssprache definierten Knotentypen. Diese Beschreibungssprache ermöglicht es, die Knotentypen mittels von Klassenhierarchien und Interfaces objektorientiert zu gestalten. Bauhaus generiert auf Basis dieser Beschreibung automatisch entsprechenden Code, welcher die Datenstrukturen zur Repräsentation des Graphen enthält und Manipulationen des Graphen erlaubt. Des Weiteren stellt Bauhaus für den IML-Graphen De-/Serialisierungsfunktionen bereit. Dabei unterstützt Bauhaus verschiedene Programmiersprachen, wobei der Großteil von Bauhaus Ada nutzt.

Die einzelnen Analysewerkzeuge in Bauhaus arbeiten in der Regel so, dass sie die bisherigen Analyseergebnisse in Form eines IML-Graphen einlesen. Auf Basis des eingelesenen Graphen führen sie dann ihre Analyse durch und exportieren das Ergebnis erneut als IML-Graph, der nun mit den Analyseergebnissen angereichert ist. Eine Ausnahme hiervon bildet das Frontend, welches den Quelltext einliest und als angereicherten AST in Form eines IML-Graphen ausgibt.

Kapitel 3

Spezifikation

3.1 Eingabe

Die Eingabe besteht stets aus zwei Mengen von Klonen. Dieses Kapitel beschreibt die Darstellung der verschiedenen möglichen Eingabeklonmengen. Beide Mengen müssen Klone der selben Darstellungsform enthalten. Die Diagramme zeigen die Darstellung in der IML-Klassenhierarchie. Implementierungsdetails sind teilweise zum Zwecke der besseren Übersichtlichkeit ausgelassen.

Das Ergebnis einer Klonerkennung besteht stets aus einer Menge von Klonen. Die Klone sind entweder Klonpaare oder Klongruppen. In IML ist dies mit der Klasse `Abstract_Clone_Tool_Info` implementiert. Diese ist eine Unterklasse von `Analysis_Tool_Info`. `Analysis_Tool_Info` dient als Überklasse für Analyseergebnisse. Von `Abstract_Clone_Tool_Info` gibt es je eine konkrete Ausprägung für Mengen von Klonpaaren beziehungsweise Klongruppen. Siehe Abbildung 3.1.

3.1.1 Klonpaare

Ein Klonpaar (im Folgenden auch CP) besteht aus zwei Klonfragmenten ($CP.CF_1$ und $CP.CF_2$), die der Klonerkennungsalgorithmus als Klon voneinander eingestuft hat. Wie die beiden Klonfragmente dargestellt sind, hängt davon ab, auf welcher Darstellungsebene der Algorithmus arbeitet und wird in den nachfolgenden Abschnitten beschrieben. Welches Klonfragment als erstes beziehungsweise als zweites bezeichnet wird, ist willkürlich und hat somit keine Bedeutung.

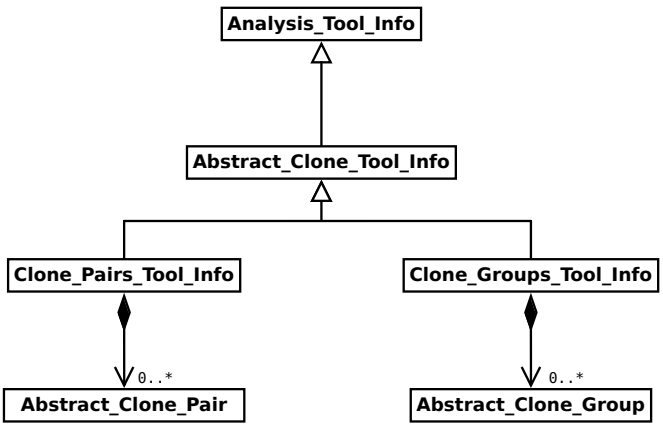

Abbildung 3.1: Klassendiagramm zur IML-Dartsellung von Klonerkennungsergebnissmengen

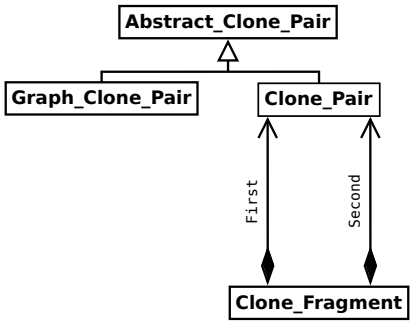

Abbildung 3.2: Klassendiagramm zur IML-Darstellung von Klonpaaren

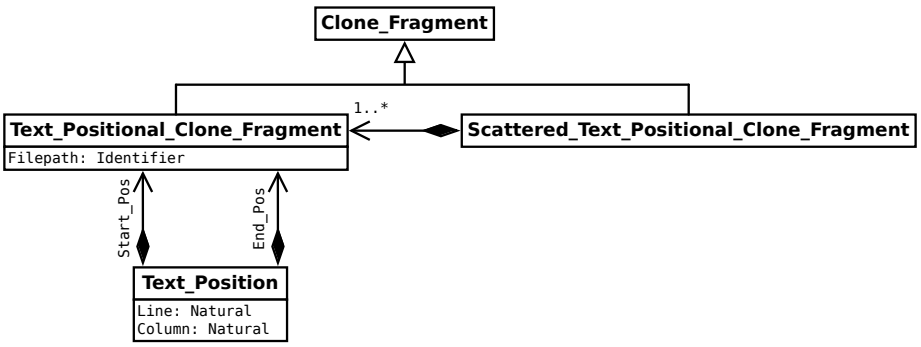

Abbildung 3.3: Klassendiagramm zur IML-Darstellung von Textklonfragmenten

In IML werden Klonpaare durch die Klasse `Clone_Pair` abgebildet. Die Darstellung von Graphklonen mit Zuordnung benötigt eine etwas andere Darstellung (siehe unten). Daher werden diese beiden Varianten von Klonpaaren als `Abstract_Clone_Pair` zusammengefasst. Siehe Abbildung 3.2.

3.1.1.1 Textklonpaare

Im Falle von Textklonpaaren besteht ein Klonfragment aus einem oder mehreren Quelltextstücken. Diese werden jeweils durch den Pfad der betreffenden Datei und der Anfangs- und Endposition darin beschrieben. Siehe Abbildung 3.3.

3.1.1.2 AST-Klonpaare

Bei AST-Klonpaaren besteht ein Klonfragment aus einem oder mehreren direkt aufeinanderfolgenden AST-Knoten. Mit einem AST-Knoten ist jeweils auch implizit der gesamte Teilbaum darunter gemeint.

Abbildung 3.4 zeigt die IML-Darstellung. `IML_Root` bezeichnet die Elternklasse für alle nicht nativen Knotentypen im IML-Graphen. In diesem Fall sollte es sich um einen Knoten des ASTs handeln. Die etwas inkonsistente Benennung mit `Clone_Statment_Fragment` beziehungsweise `Clone_Sequence_Fragment` rührt daher, dass dieser Teil bereits aus der `ccdiml`-Implementierung in Bauhaus vorhanden war.

3.1.1.3 Einfache Graphklonpaare

In der einfachen Graphklonpaardarstellung sind die beiden Klonfragmente eine Menge von Knoten. Der Knotentyp ist nicht näher spezifiziert, dabei ist aber primär an Knoten aus einem PDG gedacht. Die Implementierung trifft über die Art der Knoten aber keine weiteren Annahmen. Abgesehen

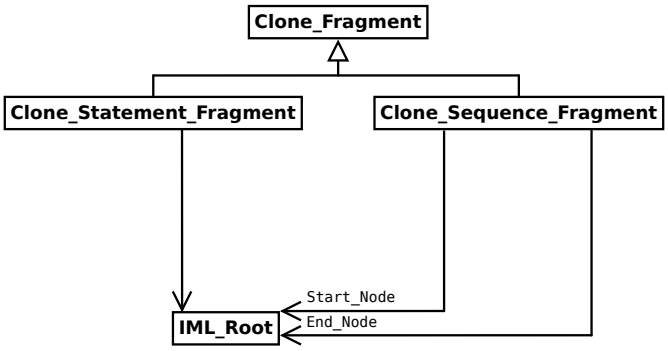

Abbildung 3.4: Klassendiagramm zur IML-Darstellung von AST-Klonfragmenten

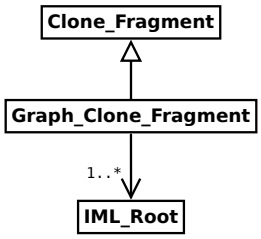

Abbildung 3.5: Klassendiagramm zur IML-Darstellung von einfachen Graphklonfragmenten

davon, dass in Abschnitt 4.2.3 diese einer Quelltextposition zuordenbar sein sollten. Abbildung 3.5 zeigt die Umsetzung in IML.

3.1.1.4 Graphklonpaare mit Zuordnung

Neben der gerade beschrieben einfachen Darstellung von Graphklonpaaren, wird auch noch eine zweite Form unterstützt. Hierbei wird zusätzlich eine bijektive Abbildung zwischen den Klonfragmenten angegeben $m : \mathrm{CF}_1 \rightarrow \mathrm{CF}_2$. Hiermit gibt der Erkennungsalgorithmus an, wie er die Knoten aus dem einen Fragment, den Knoten aus dem Anderen zuordnet.

Da IML keine Hashmaps oder Vergleichbares unterstützt, wird, wie Abbildung 3.6 zeigt, jeder Knoten aus dem ersten Fragment zusammen mit dem gemäß m dazugehörigen Knoten aus dem zweiten Fragment zu einem `Nodes_Match`-Objekt zusammengefasst. Die Menge dieser Knotenpaare ergibt das Klonpaar. In dieser Darstellung sind die beiden Fragmente implizit durch die Menge aller ersten beziehungsweise zweiten Elemente der `Nodes_Match`-Objekte gegeben.

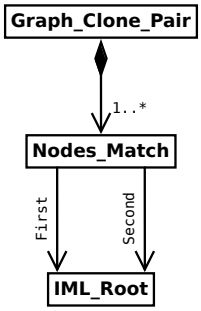

Abbildung 3.6: Klassendiagramm zur IML-Darstellung von Graphklonpaaren mit Zuordnung

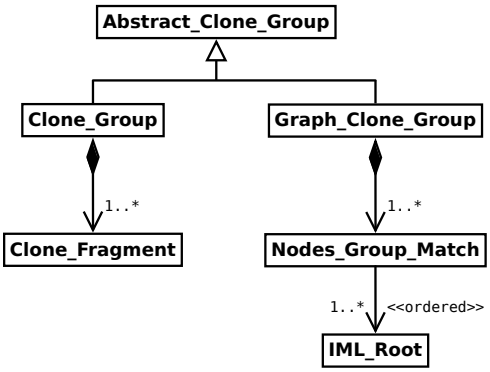

Abbildung 3.7: Klassendiagramm zur IML-Darstellung von Klongruppen

3.1.2 Klongruppen

Bei Klongruppen werden nicht ein Paar von zwei ähnlichen Klonfragmenten zu einem Klon zusammengefasst, sondern es wird gleich eine ganze Gruppen von identischen (beziehungsweise ähnlichen) Fragmenten zu einer Klongruppe zusammengefasst. Die einzelnen Klonfragmente werden wieder wie im Falle der Klonpaare dargestellt.

Abbildung 3.7 zeigt die Darstellung in IML. Für die IML-Darstellung von Graphklongruppen mit Zuordnung der einzelnen Knoten zu den entsprechenden Knoten in den anderen Fragmenten wird wieder der selbe „Trick" wie bei den Klonpaaren verwendet. Hier hingegen besteht ein `Nodes_Group_Match` aus mehreren Knoten. Die Anzahl der Elemente in der Liste `Nodes_Group_Match` muss für alle Elemente einer `Graph_Clone_Group` gleich sein. Diese Anzahl entspricht der Zahl an Fragmenten, aus welche diese Klongruppe besteht. Sollte es für eine Zuordnungsmenge (`Nodes_Group_Match`) in einem Fragment keinen passenden Knoten geben, ist der entsprechende Eintrag in der Liste dieses Fragmentes leer zu lassen.

3.2 Ausgabe

Die Ausgabe besteht aus einer Liste mit Einträgen, die jeweils eine der folgenden Formen haben:

1. Ein Klon, der nur in der ersten Eingabemenge vorkommt.
2. Je ein Klon aus den beiden Mengen und ein Wert, der die Ähnlichkeit beider Klone angibt.
3. Ein Klon, der nur in der zweiten Eingabemenge vorkommt.

Das Ähnlichkeitsmaß im zweiten Fall ist eine Zahl zwischen 0 und 1 (1 entspricht „identisch"), welche die Ähnlichkeit der beiden Klone beschreiben soll. Siehe Abschnitt 4.1.2 für Details. Ein Klon kann in mehreren Einträgen der zweiten Form auftauchen, da er zu verschiedenen Klonen aus der anderen Menge ähnlich sein kann.

Ab welchem Ähnlichkeitsmaß zwei Klone als zusammenpassend eingestuft werden, und damit einen Eintrag von Typ 2 bilden, lässt sich einstellen.

Diese Ergebnismenge lässt sich nach den Einträgen filtern, an denen man interessiert ist. Vergleicht man beispielsweise zwei verschiedene Verfahren, ist man daran interessiert, welche Klone nur von einem der beiden Verfahren erkannt wurden. Hierzu betrachtet man die Einträge von Typ 1 beziehungsweise Typ 3. Interessant sind auch Klone, die ähnlich aber nicht identisch sind. Diese kann man näher betrachten, um zu untersuchen, warum die zwei Verfahren sich hier unterscheiden. Dazu filtert man die Einträge von Typ 2 nach ihrem Ähnlichkeitsmaß.

Kapitel 4

Lösungsansatz

4.1 Vergleich

Dieser Abschnitt beschreibt das Verfahren zum Vergleich zweier Klonerkennungsergebnisse. Dieses Verfahren ist zweigeteilt. Der eine Teil, welcher im ersten Abschnitt beschrieben ist, beschäftigt sich mit der Strategie welche Klone paarweise verglichen werden müssen. Der paarweise Vergleich, welcher im zweiten Abschnitt beschrieben wird, ordnet den zwei gegebenen Klonen ein Ähnlichkeitsmaß zu. Entsprechend der Ergebnisse der einzelnen Vergleiche, werden die Klone in die oben beschriebene Ergebnismenge einsortiert.

4.1.1 Vergleich von Klonmengen

Dieser Abschnitt beschreibt, wie zwei Mengen von Klonen (d.h. Klonpaare oder Klongruppen) verglichen werden. Dazu benötigt man eine Strategie, welche Paare von Klonen man auf Ähnlichkeit testen möchte. Natürlich muss dabei sichergestellt sein, dass keine Klone, die ähnlich sind, ausgelassen werden.

4.1.1.1 Naiver Ansatz

Der naive Ansatz ist, jeden Klon aus der ersten Menge mit jedem Klon der zweiten Menge zu vergleichen. Dieser Ansatz hat den Vorteil, dass er offensichtlich korrekt ist. Außerdem trifft er keine weitere Annahmen über die Eigenschaften der Klone. Allerdings werden quadratisch viele Vergleiche benötigt.

4.1.1.2 Hashbasierter Vergleich

Beim naiven Ansatz werden viele Klone verglichen, die offensichtlicher Weise nicht ähnlich sein können. Betrachtet man beispielsweise Textklone, ist es klar, dass zwei Klone deren Fragmente nur verschiedene Dateien betreffen, nicht zueinander ähnlich sein können. Daraus ist die Idee entstanden, dass man Mengen für jeden Dateipfad bildet und anschließend nur Klone, die dieselbe Datei betreffen, vergleichen muss.

Die Idee wurde dahingehend erweitert, dass nicht nur der Dateipfad verwendet werden kann, sondern, dass ein Klon auf eine Menge von Hashwerten abgebildet wird. In dem gerade beschriebenen Beispiel wäre das die Menge aller Hashwerte, der von einem Klon abgedeckten Dateien.

Im Allgemeinen muss diese Funktion die Eigenschaft haben, dass der Schnitt der Hashmengen zweier Klone genau dann leer ist, wenn die zwei Klone nicht ähnlich sein können.

```
// build hashmap for the given set of clones
function to_hashmap(clones)
  map = new Hashmap Hashvalue => (Set of Clones);
  for clone in clones loop
    for i in list_of_hashes(clone) loop
      map.put(i, clone);
    end;
  end;
  return map;
end;

// compare two sets of clones
function compare(a, b)
  compared := new Set of (Clone, Clone);
  map_a = to_hashmap(a);
  map_b = to_hashmap(b);

  for (k, clones) in map_a loop
    for i in clones loop
      for j in map_b.get(k) loop
        if not compared.contains((i, j)) then
          // compare i and j ...
          compared.put((i, j));
        end;
      end;
    end;
  end;

  // ...
end;
```

Abbildung 4.1

Um damit Klonmengen effizienter zu vergleichen, werden mit Hilfe dieser Funktion die Klone der beiden Eingabemengen in jeweils eine Hashmap sortiert. Eine Hashmap enthält für jeden Hashwert die Menge der Klone, die darauf abgebildet werden.

Nun müssen die für einen Hashwert in der ersten Hashmap enthaltenen Klone, mit denen für den selben Wert aus der zweiten Hashmap verglichen werden. Dabei können schon verglichene Paare von Klonen übersprungen werden.

Dieses Verfahren wird als Pseudocode in Abbildung 4.1 nochmals verdeutlicht. `list_of_hashes` bezeichnet dabei die Funktion, die dem Klon, wie gerade beschrieben, eine Menge an Hashwerten zuordnet. Der paarweise Vergleich ist im Pseudocode nur mit einem Kommentar angedeutet.

Der Effizienzgewinn hängt natürlich sehr von der gewählten Funktion und der Gestalt der Eingabedaten ab. Im Worst-Case werden weiterhin quadratisch viele Vergleiche benötigt. Allerdings sollte dies für typische Eingaben die Zahl der Vergleiche deutlich reduzieren.

In dieser Arbeit wird ausschließlich die Verwendung von zu Klonen gehörenden Dateipfaden betrachtet. Nachteil dieser Methode ist, dass man zwar Textklonen und AST-Klonen direkt Dateipfaden zuordnen kann, dies für Graphknoten im Allgemeinen aber nicht gilt.

4.1.2 Vergleich einzelner Klone

Nun betrachten wir, wie sich ein Paar von Klonen vergleichen lässt. Konkret möchten wir das Paar auf ein Ähnlichkeitsmaß abbilden. Das Ähnlichkeitsmaß ist ein Wert zwischen 0 und 1. Wobei es 0 sein soll, wenn die Klone völlig verschieden sind und 1, falls diese identisch sind. Es werden keine weiteren Einschränkungen für das Ähnlichkeitsmaß gefordert.

4.1.2.1 Textklonpaare

Für Textklone wird die von Stefan Bellon in [2] als good bezeichnete Funktion gewählt. Diese betrachtet das Verhältnis der sich überlappenden Zeilen der Klonfragmente zu der Gesamtzahl der Zeilen.

$$\text{overlap}(\text{CF}_1, \text{CF}_2) = \frac{|\text{lines}(\text{CF}_1) \cap \text{lines}(\text{CF}_2)|}{|\text{lines}(\text{CF}_1) \cup \text{lines}(\text{CF}_2)|}$$

Wobei lines(CF) die Menge der von dem Klonfragment CF überdeckten Zeilen angibt.

Um nun zwei Klonpaare miteinander zu vergleichen, betrachtet man die Überlappung der jeweiligen Klonfragmente. Dabei ist zu beachten, dass die Zuordnung vom ersten und zweiten Klonfragment keine Bedeutung hat und man daher beachten muss, dass das erste Klonfragment des ersten Klonpaares zu dem zweiten Klonfragment des zweiten Klonpaares passen kann.

Stefan Bellon behandelt dieses Problem, indem er die Klonfragmente anhand der Quelltextpositionen ordnet. Daher muss er nur das erste Fragment des einen Klonpaares mit dem ersten Fragment des anderen Klonpaares vergleichen.

$$\begin{aligned}\text{similarity}(\text{CP}_1, \text{CP}_2) = \min(&\text{overlap}(\text{CP}_1.\text{CF}_1, \text{CP}_2.\text{CF}_1),\\ &\text{overlap}(\text{CP}_1.\text{CF}_2, \text{CP}_2.\text{CF}_2))\end{aligned}$$

Dies ist in unserem Fall nicht möglich, da ein Klonfragment nicht nur aus einem einzelnen zusammenhängenden Quelltextstück bestehen kann, sondern auch aus mehreren verteilten Quelltextstücken. Daher lassen sich die Klonfragmente nicht einfach ordnen. Von daher müssen wir beide Kombinationen vergleichen:

$$\begin{aligned}\text{similarity}(\text{CP}_1, \text{CP}_2) = \max(&\min(\text{overlap}(\text{CP}_1.\text{CF}_1, \text{CP}_2.\text{CF}_1),\\ &\quad\text{overlap}(\text{CP}_1.\text{CF}_2, \text{CP}_2.\text{CF}_2))\\ &\min(\text{overlap}(\text{CP}_1.\text{CF}_1, \text{CP}_2.\text{CF}_2),\\ &\quad\text{overlap}(\text{CP}_1.\text{CF}_2, \text{CP}_2.\text{CF}_1)))\end{aligned}$$

Um overlap zu berechnen, wird nicht wirklich die Mengen der von einem Klonfragment abgedeckten Zeilen gebildet und anschließend Schnitt beziehungsweise Vereinigung erzeugt. Dazu werden die in einem Fragment enthaltenen Textstücke bereits beim Einlesen nach ihrer Position sortiert. Dann lässt sich overlap in einem Durchlauf über die Liste der Textstücke der Fragmente berechnen, wie der Pseudocode in Abbildung 4.2 zeigt. Der Übersichtlichkeit halber, geht der Pseudocode davon aus, dass alle Textstücke zur selben Datei gehören.

```
function text_fragment_overlap(a, b)
  return max(0,
             min(a.end_pos.line, b.end_pos.line) -
             max(a.start_pos.line, b.start_pos.line) + 1);
end;

// calculate overlap of two code fragments (each a list of text fragments)
function overlap(cf_a, cf_b)
  cut = 0;
  union = 0;

  i = 0;
  j = 0;

  a = cf_a[i];
  b = cf_a[j];
  union += a.end_pos.line - a.start_pos.line + 1;
  union += b.end_pos.line - b.start_pos.line + 1;

  loop
    cut += text_fragment_overlap(a, b);

    if a.end_pos.line < b.end_pos.line
      i += 1;
      break if i >= length(cf_a);
      a := cf_a[i];
      union += a.end_pos.line - a.start_pos.line + 1;
    else
      j += 1;
      break if j >= length(cf_b);
      b = cf_a[j];
      union += b.end_pos.line - b.start_pos.line + 1;
    end;
  end;

  while i < length(cf_a) - 1 do
    i += 1;
    a := cf_a[i];
    union += a.end_pos.line - a.start_pos.line + 1;
  end;

  while j < length(cf_b) - 1 do
    j += 1;
    b := cf_b[i];
    union += b.end_pos.line - b.start_pos.line + 1;
  end;

  union -= cut;

  return cut / union;
end;
```

Abbildung 4.2

4.1.2.2 AST-Klonpaare

Für AST-Klonpaare lässt sich ein sehr ähnliche Funktion nutzen. overlap wird wie folgt ersetzt:

$$\text{overlap}(\text{CF}_1, \text{CF}_2) = \frac{|\text{subtree}(\text{CF}_1) \cap \text{subtree}(\text{CF}_2)|}{|\text{subtree}(\text{CF}_1) \cup \text{subtree}(\text{CF}_2)|}$$

Wobei subtree(CF) die Menge der AST-Knoten in dem durch CF gebildeten Teilbaum sind.

Wie im Falle der Textklone, kann auch hier overlap effizienter berechnet werden, als tatsächlich die Schnitt- beziehungsweise Vereinigungsmengen zu bilden. Dazu wird entweder beim ersten Besuch eines Knotens oder bereits im Voraus die Größe des darunterliegenden Teilbaumes berechtnet. Um den Schnitt und die Vereinigung zweier Knoten zu ermitteln, wird die Größe der jeweiligen Teilbäume verglichen. Ist diese identisch, können sie sich nur überlappen, wenn es sich um den selben Knoten handelt. Falls sie unterschiedliche Größen haben, wird von dem kleineren Knoten ausgehend in Richtung Wurzel gegangen und getestet ob der anderen Knoten getroffen wird. Ist dies der Fall, kann anhand der bekannten Teilbaumgrößen overlap berechnet werden, ansonsten überlappen sich die Knoten nicht.

4.1.2.3 Einfache Graphklonpaare

Auch hier wird die Ähnlichkeitsfunktion für Textklonpaare verwendet. In diesem Fall ist overlap direkt die Überdeckung der beiden Klonfragmente:

$$\text{overlap}(\text{CF}_1, \text{CF}_2) = \frac{|\text{CF}_1 \cap \text{CF}_2|}{|\text{CF}_1 \cup \text{CF}_2|}$$

4.1.2.4 Graphklonpaare mit Zuordnung

Für Graphklonpaare mit Zuordnung ist ein leicht anderer Ansatz nötig, da hier noch die Abbildung zwischen den beiden Knotenmengen zu betrachten ist. Seien m_1 und m_2 die beiden Zuordnungen:

$$m_1 : \text{CP}_1.\text{CF}_1 \rightarrow \text{CP}_1.\text{CF}_2$$
$$m_2 : \text{CP}_2.\text{CF}_1 \rightarrow \text{CP}_2.\text{CF}_2$$

Nun werden die dazugehörigen Relationen M_1 und M_2 betrachtet, wobei M_2' die Relation zur Umkehrabbildung von m_2 sei.

$$M_1 \subset (\text{CP}_1.\text{CF}_1 \times \text{CP}_1.\text{CF}_2)$$
$$M_2 \subset (\text{CP}_2.\text{CF}_1 \times \text{CP}_2.\text{CF}_2)$$
$$M_2' \subset (\text{CP}_2.\text{CF}_2 \times \text{CP}_2.\text{CF}_1)$$

Als Ähnlichkeitsmaß wird das Verhältnis, der identisch zugeordneten Knotenpaare zu der Gesamtzahl, gewählt. Dabei muss wieder beachtet werden, dass erstes und zweites Klonfragment vertauscht sein können.

$$\text{similarity}(\text{CP}_1, \text{CP}_2) = max\left(\frac{|M_1 \cap M_2|}{|M_1 \cup M_2|}, \frac{|M_1 \cap M_2'|}{|M_1 \cup M_2'|}\right)$$

4.1.2.5 Klongruppen

Im Falle von Klongruppen handelt es sich jeweils um eine Menge von Klonfragmenten CG_1 und CG_2. Im Folgenden gehen wir davon aus, dass die erste Klongruppe die Kleinere ist $|\mathrm{CG}_1| \leq |\mathrm{CG}_2|$ (ansonsten tauschen).

Für das Ähnlichkeitsmaß ordnen wir jedem Klonfragment aus der ersten Klongruppe mit Hilfe der oben definierten overlap-Funktion das am besten passende Fragment aus der zweiten Klongruppe zu. Die overlap werden aufsummiert und durch die Anzahl an Klonfragmenten in der größeren Klongruppe geteilt. Damit wird im Prinzip der durchschnittliche overlap berechnet.

$$\mathrm{similarity}(\mathrm{CG}_1, \mathrm{CG}_2) = \frac{1}{|\mathrm{CG}_2|} \sum_{i \in \mathrm{CG}_1} \max_{j \in \mathrm{CG}_2} \mathrm{overlap}(i, j)$$

Hierbei wird jeweils der beste overlap-Wert genommen. Daher kann es sein, dass für mehrere Klonfragmente das selbe Klonfragment aus der zweiten Klongruppe „zugeordnet" wird. Somit wird in diesem Fall die Ähnlichkeit zu gut abgeschätzt. Dies liese sich beheben, indem anstatt des Maximums, die größte gewichtete Paarung berechnet wird. Dieses Vorgehen ist allerdings aufwändiger [7] und wird daher nicht verwendet.

Diese Methode kann für Text-, AST- und einfache Graphklone genutzt werden. Die overlap-Funktion ist, jeweils wie für entsprechende Klonpaare beschrieben, definiert.

4.1.2.6 Graphklongruppen mit Zuordnung

Für Graphklongruppen mit Zuordnung ergibt sich das Problem, dass das Verfahren für Klonpaare nicht übertragbar ist, da es zu viele Möglichkeiten gibt, wie das Zuordnungs-Tupel sortiert sein kann. Daher wird eine Kombination aus zwei Ähnlichkeitsmaßen genutzt.

Zum Einen wird das oben beschriebene Ähnlichkeitsmaß für einfache Klongruppen verwendet. Dadurch wird der Fall abgedeckt, dass sich die Knotenmengen der Klonfragmente in den beiden Klongruppen unterscheiden. Allerdings wird die Zuordnung der Knoten zueinander ignoriert. Dieser Wert wird im folgenden mit sim_1 bezeichnet.

Zum Anderen wird dazu analog ein Ähnlichkeitsmaß für die Zuordnung bestimmt. Eine Zuordnung ist gegeben durch die Menge M aller n-Tupel. n ist hierbei die Anzahl an Klonfragmenten in der Klongruppe. Als M' bezeichnen wir die Menge, in der die Tupel durch eine entsprechende Menge ersetzt wurden. Damit wird zwar die Zuordnung zu den einzelnen Klonfragmenten verloren, dafür lassen sich diese Mengen einfach vergleichen. Seien M'_1 und M'_2 die Zuordnungen der beiden gegebenen Klongruppen und sei $|M'_1| \leq |M'_2|$. Analog zu oben erhalten wir schließlich sim_2.

$$\mathrm{sim}_2 = \frac{1}{|M'_2|} \sum_{i \in M'_1} \max_{j \in M'_2} \frac{|i \cap j|}{|i \cup j|}$$

Dieses Ähnlichkeitsmaß wird schlechter, falls sich die Zuordnungen der beiden Klongruppen unterscheiden. Allerdings wird die Zuordnung der Knoten zu den Klongruppen ignoriert. Daher werden beide Maße kombiniert.

$$\mathrm{similarity}(\mathrm{CG}_1, \mathrm{CG}_2) = \min(\mathrm{sim}_1, \mathrm{sim}_2)$$

4.2 Konvertierung der Klondarstellungen

In diesem Abschnitt wird beschrieben, wie sich AST-und Graphklone mit Zuordnung in einfache Graphklone umwandeln lassen. Diese wiederum können auf Textklone abgebildet werden. Dabei

werden effektiv nur die Klonfragmente umgewandelt. Ob es sich sich um ein Klonpaar oder eine Klongruppe handelt, bleibt unverändert.

4.2.1 Umwandlung von AST-Klonen zu einfachen Graphklonen

Um einen AST-Klon in einen einfachen Graphklon umzuwandeln, werden die zu den AST-Knoten der Klonfragmente gehörenden Teilbäume gebildet. Die in diesen Teilbäumen enthaltenen Knoten bilden die Menge, die das jeweilige Klonfragment des einfachen Graphklon bilden.

4.2.2 Umwandlung von Graphklonen mit Zuordnung zu einfachen Graphklonen

Um ein Graphklon mit Zuordnung in einen einfachen Graphklon umzuwandeln, wird die Menge, die das jeweilige Klonfragment bildet genommen und als Klonfragment für den einfachen Graphklon verwendet. Dabei geht die Zuordnung der Knoten der Klonfragmente verloren.

4.2.3 Einfache Graphklone zu Textklone

Um nun ein Klonfragment eines einfachen Graphklons, also eine Menge an Knoten, in ein Textklonfragment umzuwandeln, werden zuerst den einzelnen Knoten die jeweilige Quelltextposition zugeordnet. Dazu wird eine bereits vorhandene Funktion genutzt, die zu einem Knoten Anfangs- und Endposition angibt. Dies ist allerdings nur für Knoten möglich, die direkt einem syntaktischen Element des Quelltextes zuzuordnen sind, wie beispielsweise einer Zuweisung. Knoten bei denen dies nicht möglich ist, werden im Moment ignoriert. In einigen Fällen ließe sich das noch verbessern. Allerdings gibt es auch Knotentypen, bei denen es nicht klar ist, welche Quelltextposition zugeordnet werden soll.

Nun müssen diese Quelltextstücke zu einem Textklonfragment zusammengefasst werden. Dazu wurden zwei Verfahren verwendet.

4.2.3.1 Zusammenfassen zu mehrere Textstücke

Bei diesem Verfahren werden die zu den einzelnen Knoten gehörenden Textstücke nur zusammengefasst, wenn diese direkt aneinander liegen. So kann das resultierende Klonfragment aus mehreren unzusammenhängenden Textstücken bestehen. Dies ist besonders vorteilhaft, wenn das Graphklonfragment nichtzusammenhängende Teile des Quelltextes abdeckt. Dies ist beispielsweise bei PDG-basierten Verfahren üblich.

Der Nachteil ist, dass dieses Verfahren nur zeilenweise arbeiten kann. Dies liegt daran, dass es auf Spaltenebene nicht einfach festzustellen ist, ob zwei Textstücke effektiv aneinander liegen. Dies ist der Fall, da beispielsweise in dem Codestück `b + c`, den Zwischenräumen keine AST-Knoten zuordenbar sind. Würde also die Spalteninformation beim Zusammenfassen der Textstücke betrachtet werden, wären dies drei getrennte Textstücke. Zwei Textstücke, welche die Variablen abdecken und ein Textstück, das den Plusoperator abdeckt. Diese müssten also intelligent zusammengefasst werden, was aufwändig ist.

4.2.3.2 Zusammenfassen zu einem einzelnen Textstück

Entstand das einfache Graphklonfragment beispielsweise durch das Umwandeln von einem AST-Klonfragment, weiß man, dass diese Knoten sinnvoll zu einem Textstück zusammenfassbar sind. Daher wird im zweiten Verfahren (pro Datei) ein Textstück anhand der kleinsten Start- beziehungsweise der größten Endposition gebildet. Dies hat den Vorteil, dass hierbei auch die Spalteninformationen erhalten bleiben können.

Kapitel 5

Entwurf

Dieses Kapitel soll eine Übersicht über die Gestaltung der Implementierung geben. Die Software wurde in Ada implementiert. Zum Entwurf wurden hauptsächlich das Paradigma der Objektorientierung genutzt. Im Allgemeinen wurde darauf geachtet, dass sich einzelne Teile gut tauschen lassen, so dass es zum Beispiel möglich ist, ein weiteres Eingabeformat oder eine andere Vergleichsstrategie zu unterstützen.

Das Diagramm in Abbildung 5.1 gibt einen Überblick über den Datenfluss bei der Durchführung eines Vergleiches. Die einzelnen Bezeichner sind hierbei aus der Implementierung übernommen.

`InputList` und `Result` dienen hierbei zur Repräsentation der Eingabe beziehungsweise Ausgabemengen. Die Klone sind dabei in etwa so dargestellt wie in Abschnitt 3.1 vorgestellt. Der signifikanteste Unterschied ist, dass die Graphklonpaare mit Zuordnung als Hashmap abgebildet werden.

Bei den anderen Kästchen im Diagramm handelt es sich um abstrakte Klassen, von denen es jeweils verschiedene Implementierungen gibt, um die verschiedenen Darstellungen, Vergleichsoperationen, etc. zu unterstützen. Diese werden im Folgenden auch als Module bezeichnet.

Ein `Input`-Modul liest eine Menge aus Klonen aus einer Datei mit dem von ihm unterstützten Dateiformat. Daraus erzeugt es eine `InputList`, welche die Menge der eingelesene Klone darstellt. Für jede unterstützte Kombination aus Klondarstellung und Eingabeformat gibt es eine entsprechende Implementierung. Beispielsweise liest `Text_CSV_Input` Textklonpaare aus einer CSV-Datei und `Tree_IML_Input` liest AST-Klonpaare aus einer IML-Datei, die wie in Abschnitt 3.1.1 gestaltet ist.

Die beiden eingelesenen Mengen werden dann mit einem `Diff`-Modul verglichen. Siehe Abschnitt 4.1.1 für die beiden in dieser Arbeit vorgestellten Vergleichsstrategien.

Zwei Klone vergleicht das `Diff`-Modul mit Hilfe eines `Comparator`-Moduls, welches deren Ähnlichkeit, wie in Abschnitt 4.1.2 beschrieben ermittelt.

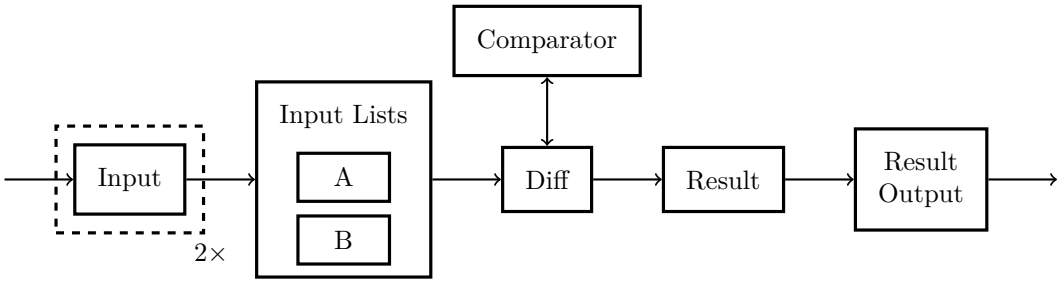

Abbildung 5.1: Übersicht über den Datenfluss beim Vergleich zweier Klonmengen

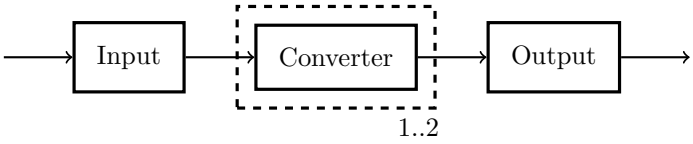

Abbildung 5.2: Übersicht über den Datenfluss beim Konvertieren zwischen verschiedenen Klondarstellungsformen

Das Ergebnis wird anschließend von einem `ResultOutput` Modul wieder in eine Datei geschrieben.

Die Umwandlung verschiedener Klondarstellungen verläuft wie in Abbildung 5.2 verdeutlicht. Hierbei bezeichnet `Converter` das Modul, welches eine der drei in Abschnitt 4.2 beschriebenen Umwandlungen durchführt. Das Ergebnis ist wieder eine Klonmenge. Diese wird von einem `Output`-Modul in eine Datei geschrieben.

In der aktuellen Version wird für die Ein-/Ausgabe der Klonmengen primär IML verwendet. Für Textklone wird aktuell noch CSV als Format unterstützt. Die Ausgabe des Vergleichsergebnisses erfolgt im Moment in einem CSV-Format.

Kapitel 6

Evaluierung

Zur Evaluierung des erstellten Werkzeuges, wurde es beispielhaft angewandt. Als Klonerkennung wurde zum Einem `ccdiml` und zum Anderen ein im Moment von Torsten Görg in Entwicklung befindender PDG-basierter Klonerkenner verwendet. `ccdiml` ist ein von Stefan Bellon entwickelter Klonerkenner, der das AST-basierte Verfahren nach Baxter [5] implementiert und ist Bestandteil von Bauhaus. Für Bauhaus existiert ein Satz von bereits zur Analyse vorbereiteten Open-Source-Projekten. Beide Werkzeuge bauen auf Bauhaus auf, daher wurde eines dieser Open-Source-Projekte gewählt und mit beiden Klonerkennern analysiert.

Bei dem analysierten Softwareprojekt handelt es sich um gnuplot, ein unter einer freien Lizenz verfügbares Werkzeug zum Plotten von Daten und Funktionen. Gnuplot ist in C geschrieben. Es wurde Version 4.0.0 von gnuplot verwendet.

Dazu wurde zuerst das Bauhaus-Frontend aufgeführt, um aus dem Quellcode die IML-Darstellung zu erzeugen. Daraufhin wurden, die vom PDG-basierten Klonerkenner benötigten Analyseergebnisse (Zeigeranalyse, Kontrollflussanalyse, SSA, etc.), mit den entsprechenden Bauhauswerkzeugen erzeugt. Die resultierenden IML-Graphen konnten nun von beiden Klonerkennungswerkzeugen analysiert werden.

Der PDG-Klonerkenner war so konfiguriert, dass er nur Klonfragmente, die mindestens 10 Knoten enthalten, ausgibt. Mit dieser Einstellung fand er 7082 Klone.

Da der PDG-Klonerkenner viele kleine Klone erkannt hatte wurde `ccdiml` auch so konfiguriert, dass es kleine Klone erkennt. Konkret wurde `ccdiml` im „feinen"-Modus (das heißt, dass es auch kleine Statements als Klone betrachtet) mit einer Mindestlänge von drei Zeilen betrieben.

Um die beiden Ausgaben vegleichen zu können, wurden sie wie in Abschnitt 4.2 beschrieben in Textklone umgewandelt.

Die erstellte Software fand 7017 Klone, die nur der PDG-Erkenner gefunden hat, 183 Klone die nur `ccdiml` gefunden hat und 72 Tupel von Klonen, die einander ähnlich sind. Die Summe ist großer als die Gesamtzahl der Klone, da ein Klon zu mehreren verschiedenen Klonen ähnlich sein kann.

Von den nur durch eines der Werkzeuge erkannten Klonen, wurden jeweils fünf zufällig ausgewählt und von Hand betrachtet.

In den Abbildungen ist jeweils das erste Klonfragment rot und das zweite grün markiert. Zeigt die Abbildung zwei Klonpaare, zeigt die linken Markierungen das Klonpaar des PDG-Klonerkenners und die rechten Markierungen das Klonpaar, dass `ccdiml` gefunden hat.

```
src/gplt_x11.c:
[...]
3444  #define KNOWN_KEYSYMS(gp_keysym)                                        \
3445                  if (plot == current_plot) {                             \
3446                      gp_exec_event(GE_keypress,                          \
3447                          (int)RevX(event->xkey.x), (int)RevY(event->xkey.y),  \
3448                          gp_keysym, 0);                                  \
3449                  }                                                       \
3450                  return;
[...]
3486          case XK_Prior:          /* XXX */
3487              KNOWN_KEYSYMS(GP_PageUp);
3488          case XK_Next:           /* XXX */
[...]
3536  #ifdef XK_KP_End
3537          case XK_KP_End:
3538              KNOWN_KEYSYMS(GP_KP_End);
3539  #endif
[...]
```

Abbildung 6.1

6.1 Nur vom PDG-Klonerkenner gefundene Klone

Von den fünf gewählten Klonen, die nur der PDG-Klonerkenner gefunden hatte, handelt es sich bei vieren um Klone, die eine einzelne Quelltextzeile umfassen und in der Tat sehr ähnlich sind. Bei drei dieser Klone handelt es sich, wie in Abbildung 6.1 gezeigt, um einen Zweig eines `switch`-Statements, welches Tastendrücke verarbeitet.

Diese Klone sind unterhalb des Schwellwerts von drei Zeilen, der bei `ccdiml` eingestellt war und konnten daher von diesem nicht erkannt werden. Dies ist mutmaßlich auch einer der Gründe, warum der PDG-Klonerkenner sehr viel mehr Klone gefunden hat als `ccdiml`.

Der eine verbleibende Klon (siehe Abbildung 6.2) ist ein Reihe von über viele Zeilen zerstreuten Anweisungen, die auf die selben Variablen zugreifen. Diese weisen zwar Ähnlichkeiten auf, allerdings sind sie nicht als „echter" Klon zu betrachten.

6.2 Nur von `ccdiml` gefundene Klone

Bei den fünf betrachteten Klonen die nur von `ccdiml` erkannt wurden handelt es sich bei allen um Anweisungsfolgen die zwar etwas ähnlich sind, allerdings keinen „echten" Klon bilden wie das Beispiel in Abbildung 6.3 zeigt.

6.3 Von beiden Werkzeugen gefundene Klone

Von den Klonen, die von beiden Werkzeugen gefunden wurden, wurden die fünf Tupel betrachtet, die sich, nach dem verwendeten Ähnlichkeitsmaß, am ähnlichsten sind.

Bei zwei dieser Klone handelt es sich um „richtige" Klone. Bei diesen Klonen würde es sich anbieten, die Funktionalität in eine Funktion auszulagern und so die Doppelung zu reduzieren. Wie man an dem Beispiel in Abbildung 6.4 sieht, erkennen hier beide Werkzeuge effektiv den selben Klon.

Bei dem in Abbildung 6.5 gezeigten Tupel erkannten auch beide Werkzeuge den selben Klon:

Hier enthält der durch dem PDG-Klonerkenner erkannte Klon zusätzlich die Referenz auf die verwendeten Variablen. `ccdiml` erkennt dagegen, dass die nachfolgende Zeile auch noch zu dem Klon gehört.

```
src/gplt_x11.c:
[...]
3908      XrmInitialize();
3909      XrmParseCommand(&dbCmd, options, Nopt, Name, &Argc, Argv);
3910      if (Argc > 1) {
[...]
3923      }
3924      if (pr_GetR(dbCmd, ".display"))
3925          ldisplay = (char *) value.addr;
[...]
4013
4014      XrmMergeDatabases(dbCmd, &db);
4015
[...]
4019      /* a specific visual can be forced by the X resource visual */
4020      db_string = pr_GetR(db, ".visual") ? (char *) value.addr : (char *) 0;
4021      if (db_string) {
[...]
4026          for (i = 0; *ptr; i++, ptr++) {
4027              if (!strcmp(db_string, *ptr)) {
4028  #if 0
[...]
4071      /* check database for maxcolors */
4072      db_string = pr_GetR(db, ".maxcolors") ? (char *) value.addr : (char *) 0;
4073      if (db_string) {
4074          int itmp;
4075          if (sscanf(db_string, "%d", &itmp)) {
4076              if (itmp <= 0) {
[...]
```

Abbildung 6.2

```
src/gplt_x11.c:
[...]
2387  #ifdef EXPORT_SELECTION
2388      export_graph(plot);
2389  #endif
2390
2391      UpdateWindow(plot);
2392  #ifdef USE_MOUSE
[...]
2436  #ifdef USE_MOUSE
2437      EventuallyDrawMouseAddOns(plot);
2438
2439      XFlush(dpy);
2440
[...]
```

Abbildung 6.3

```
src/gpexecute.c:
[...]
 154  {
 155      gpe_fifo_t *base = malloc(sizeof(gpe_fifo_t));
 156      /* fprintf(stderr, "(gpe_init) \n"); */
 157      assert(base);
 158      base->next = (gpe_fifo_t *) 0;
[...]
 167      if ((*base)->prev) {
 168          gpe_fifo_t *new = malloc(sizeof(gpe_fifo_t));
 169          /* fprintf(stderr, "(gpe_push) \n"); */
 170          assert(new);
 171          (*base)->prev->next = new;
[...]
```

Abbildung 6.4

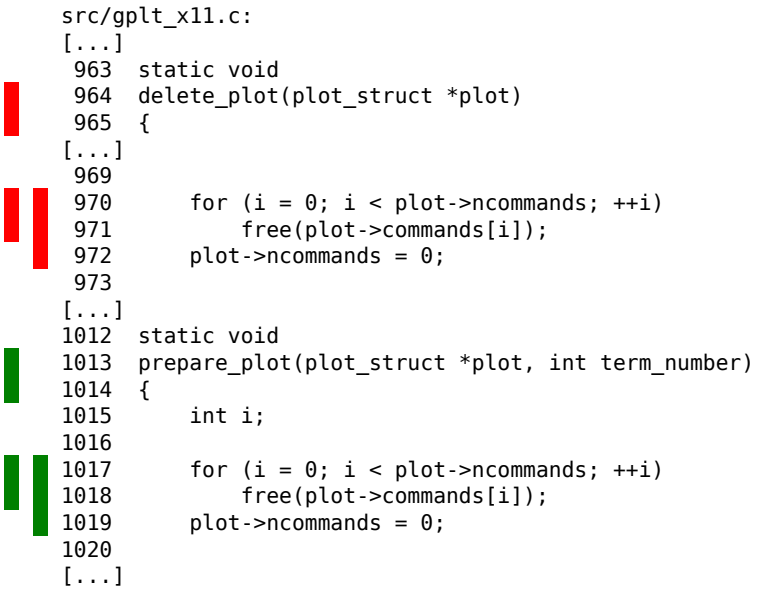

Abbildung 6.5

Die beiden restlichen betrachteten Tupel betreffen einen von `ccdiml` erkannten Klon, der zu zwei verschiedenen von dem PDG-Klonerkenner erkannten Klonen ähnlich ist. Exemplarisch ist eines der Tupel in Abbildung 6.6 gezeigt. Hierbei handelt es sich um einen kleinen Klon, wobei die von dem PDG-Klonerkenner zusätzlich gewählte Zeile willkürlich erscheint.

6.4 Zusammenfassung

Die Ergebnisse des Vergleiches scheinen plausibel. Die Menge der nur von einem der beiden Werkzeuge erkannten Klonmengen ist in diesem Fall eher nicht hilfreich, da beide Werkzeuge „zu empfindlich" eingestellt sind und viele Klone erkennen die keine „echten" sind. Aus dieser Menge wurden erfolgreich die von beiden erkannten Klone herausgefiltert. An dem oben gezeigten Beispiel wird ersichtlich, wie sich die beiden erkannten Klone unterscheiden. Für den Vergleich benötigte die Software auf einem „AMD Opteron 6174" ca. 9,0 Sekunden.

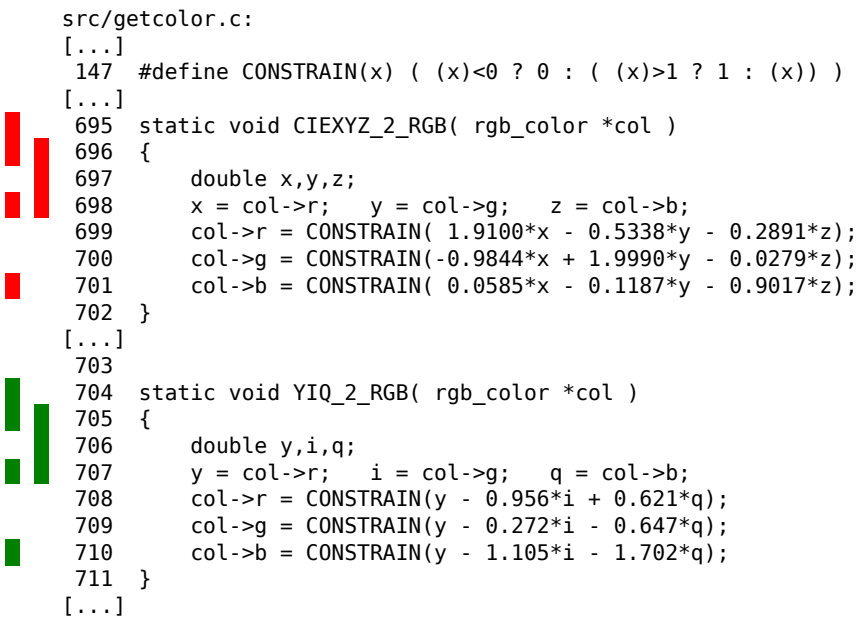

Abbildung 6.6

Kapitel 7

Fazit

In dieser Arbeit wurde ein Verfahren zum Vergleich von Klonmengen erarbeitet. Hierbei wurde besonders darauf eingegangen, dass die Klone in verschiedenen Darstellungsformen vorliegen können. Des Weiteren wurde ein Verfahren zum Umwandeln von AST- und PDG-Klonen in Textklone erarbeitet.

Diese Verfahren wurden in Form eines Softwarewerkzeuges implementiert. Besonders darauf geachtet wurde, die Software modular zu gestalten, sodass sich einzelne Teil leicht tauschen, beziehungsweise ergänzen lassen. So lässt sich zum Beispiel ein weiteres Eingabeformat oder ein anderes Vergleichsverfahren für Klonpaare einfach hinzuzufügen. Der von Bauhaus unabhängige Teil der Software ist unter der MIT-Lizenz verfügbar [8].

7.1 Ausblick

Abschließend sollen noch ein paar Verbesserungs- und Ergänzungsmöglichkeiten aufgezeigt werden.

- Für den Spezialfall von Textklonpaaren, die aus einem zusammenhängenden Textfragment bestehen, lässt die Anzahl der benötigten paarweisen Vergleiche noch verringern, indem die Klonpaare nach der Textposition ihrer Fragmente sortiert werden.
- Für den Vergleich zweier Klongruppen lässt sich eventuell ein effizienteres Ähnlichkeitsmaß finden.
- Für die manuelle Betrachtung der Ergebnismenge wäre eine Visualisierung der Klone sehr hilfreich. Für Textklone sollte das nicht schwer umzusetzen sein. Die kompakte Visualisierung von Graphklonen ist hingegen eine schwierige Aufgabe.
- Die Umwandlung von Graphklonfragmente in Quelltextfragmente lässt sich verbessern, indem die Behandlung von Knoten, die nicht direkt einem syntaktischen Element zuzuordnen sind, verbessert wird.
- Die Effizienz der Implementierung lässt sich an einigen Stellen noch verbessern.
- Es gibt noch einige Aspekte der Implementierung, die evaluiert werden können.

Kapitel 8

Quellen und Referenzen

[1] Chanchal Kumar Roy und James R. Cordy, *„A Survey on Software Clone Detection Research“*, Technical Report No. 2007-541, School of Computing, Queen's University at Kingston, 2007.
`http://research.cs.queensu.ca/TechReports/Reports/2007-541.pdf`

[2] Stefan Bellon, *„Vergleich von Techniken zur Erkennung duplizierten Quellcodes“*, Diplomarbeit Nr. 1998, Institut für Softwaretechnologie, Universität Stuttgart, 2002.
`http://www.bauhaus-stuttgart.de/bauhaus/papers/DIP-1998.pdf`

[3] Aoun Raza, Gunther Vogel und Erhard Plödereder, *„Bauhaus - A Tool Suite for Profram Analysis and Reverse Engineering“*, In *Reliable Software Technologies, Ada-Europe 2006, LNCS(4006)* Seiten 71–82, 2006.
`http://www.bauhaus-stuttgart.de/bauhaus/papers/bauhaus.pdf`

[4] Brenda S. Baker, *„On Finding Duplication and Near-Duplication in Large Software Systems“*, In *Proceedings of 2nd Working Conference on Reverse Engineering, IEEE* Seiten 86–95, 1995.

[5] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna und Lorraine Bier, *„Clone Detection Using Abstract Syntax Trees“*, In *Proceedings of the International Conference on Software Maintenance* Seiten 368–377, 1998.

[6] Jens Krinke, *„Identifying Similar Code with Program Dependence Graphs“*, In *Proceedings of the Eighth Working Conference On Reverse Engineering (WCRE'01)*, 2001.

[7] James Munkres *„Algorithms for the Assignment and Transportation Problems“*, In *Journal of the Society of Industrial and Applied Mathematics* Vol. 5, Nr. 1, Seite 32–38, 1957.

[8] `https://ccdiff.ipsumj.de/`

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben.

Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet.

Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens.

Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht.

Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Stuttgart, 15.04.2016, Simon Gaiser