Institut für Softwaretechnologie

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit Nr. 310

Bauhaus-Analysis Driver

Andreas Bauer

Studiengang:	Informatik
Prüfer/in:	Prof. Dr. rer. nat./Harvard Univ. Erhard Plödereder
Betreuer/in:	Timm Felden
Beginn am:	22. Februar 2016
Beendet am:	23. August 2016
CR-Nummer:	D.m

Kurzfassung

Um die Programmanalyse mit Bauhaus zu vereinfachen, habe ich ein Steuerungs-Werkzeug für Bauhaus entwickelt. Dieses Werkzeug ermöglicht die automatisierte Ausführung von Bauhauswerkzeugen in Unkenntnis deren Abhängigkeiten. In dieser Ausarbeitung wird die Funktionsweise dieses Werkzeuges, sowie seine Realisierung beschrieben, sowie eine Analyse der für ein solches Werkzeug relevanten Eigenschaften von Bauhaus.

Inhaltsverzeichnis

1. Einleitung **7**
1.1. Ziel der Arbeit . . . 7
1.2. Verwandte Arbeiten . . . 7
1.3. Anforderungen . . . 8
1.4. Was ist Bauhaus? . . . 8
1.5. Alternativen . . . 8

2. Analyse **9**
2.1. Anwendungsfälle . . . 9
2.2. Abhängigkeiten . . . 10
2.3. Vorteile eines Steuerungs-Werkzeugs . . . 13
2.4. Annahmen . . . 14
2.5. Voraussetzungen für die Ausführung . . . 15

3. Methodik der Umsetzung **17**
3.1. Konfigurierbarkeit . . . 17
3.2. Automatisierte Analyse . . . 20
3.3. Wiederholbarkeit . . . 22
3.4. Präferenzmechanismus . . . 22

4. Realisierung **25**
4.1. Architektur . . . 25
4.2. Designentscheidungen . . . 31
4.3. Bedienung des Werkzeugs . . . 37
4.4. Tests . . . 39
4.5. Laufzeit . . . 47

5. Zusammenfassung und Ausblick **49**

Literaturverzeichnis **51**

A. Vollständige Konfiguration **55**

1. Einleitung

Bauhaus ist ein Projekt, welches von der Universität Stuttgart, in Kooperation mit der Universität Bremen und der Firma `Axivion` entwickelt wird. Bauhaus besteht momentan aus 128 Analysewerkzeugen, welche einzeln ausgeführt werden. Die hohe Anzahl an Werkzeugen, sowie die verzweigten Abhängigkeiten zwischen diesen, erschweren die Benutzung von Bauhaus. Um die Nutzung zu erleichtern, soll ein Steuerungs-Werkzeug entwickelt werden, welches die Ausführung der Werkzeuge automatisiert. In dieser Arbeit wird die Entwicklung eines solchen Werkzeuges beschrieben.

1.1. Ziel der Arbeit

Das Ziel der Arbeit ist die Entwicklung eines Werkzeuges, welches die Analyse mit Bauhaus-Werkzeugen steuern kann. Das Werkzeug soll in der Lage sein, die Abhängigkeiten zwischen Bauhaus-Werkzeugen aufzulösen und es somit einem Anwender ermöglichen, Analysen mit Bauhaus in Unkenntnis dieser Abhängigkeiten durchzuführen.

Es soll die Nutzung von Bauhaus erleichtern und den Aufwand, den ein Nutzer für eine Analyse aufwänden muss verringern.

Das Steuerungs-Werkzeug muss eine korrekte Ausführung ermöglichen, sowie einfach zu Bedienen sein. Zudem sollte es Funktionen bereitstellen, welche einem Anwender die korrekte Nutzung erleichtern.

1.2. Verwandte Arbeiten

Es existiert zur Zeit ein *make* basiertes Skript zur Automatisierten Analyse[IST16b]. Dieser Ansatz kann die Anforderungen an eine Automatisierte Analyse nicht vollständig abdecken. Beispielsweise ist die Ausführung von manchen Werkzeugen, wie dem Werkzeug `cobra` nicht möglich und die Parametrisierung der Werkzeuge kann nicht einfach verändert werden. Um eine Analyse beispielsweise mit veränderten Parametern zu wiederholen ist *make* also ungeeignet. Des weiteren muss der Analysepfad in den Dateinamen kodiert sein, damit die Analyse ausgeführt werden kann.

1.3. Anforderungen

Laut Aufgabenstellung muss das im Rahmen dieser Arbeit entwickelte Steuerungs-Werkzeug folgende Anforderungen erfüllen [16a]:

- **Konfigurierbarkeit** das Steuerungs-Werkzeug muss leicht konfiguriert und erweitert werden können. Diese Konfiguration muss es ermöglichen, die Abhängigkeiten zwischen Werkzeugen global zu konfigurieren.
- **Automatisierte Analyse** mit Hilfe des Steuerungs-Werkzeugs muss ein Anwender Analysen in Unkenntnis der Abhängigkeiten ausführen können.
- **Wiederholbarkeit** ausgeführte Analysen müssen vollständig oder Teilweise wiederholt werden können.
- **Präferenzmechanismus** es soll ein Präferenzmechanismus für Werkzeuge, die über mehrere Abhängigkeitspfade erreichbar sind geschaffen werden. Des weiteren soll es möglich sein, die Parametrisierung von Werkzeugen für unterschiedliche Pfade unterschiedlich zu wählen.

1.4. Was ist Bauhaus?

Wie man der Homepage des Bauhausprojekts [16c] entnehmen kann, handelt es sich bei Bauhaus um eine Software zur Programmanalyse. Wie bereits erwähnt, ist das Projekt eine Kooperation. Laut der Bauhaus Homepage ist die Verteilung der Aufgaben wie folgt:

> Während in Stuttgart vor allem Analysen des Programmverhaltens erforscht werden, widmet sich Bremen vorrangig den Architektur-bezogenen Themen. Axivion vermarktet die entwickelten Analysewerkzeuge, die für den kommerziellen Einsatz ausreichend ausgereift sind.

[16c]

Genauere Informationen über Bauhaus, wie beispielsweise über die Schwerpunktthemen oder über vorhandene Funktionalitäten, können der Bauhaus Homepage [16c], der Bauhaus Demo Website [16b] oder der Präsentation [**pr\IeC {\"a}sent**] entnommen werden.

1.5. Alternativen

Eine Alternative zu einem Steuerungs-Werkzeug ist der bereits erwähnte *make* basierte Ansatz. Diese Alternative kann jedoch nicht alle Funktionen bereitstellen, die für eine korrekte und einfache Steuerung von Bauhaus benötigt werden.

2. Analyse

2.1. Anwendungsfälle

Hauptsächlich lassen sich zwei Gruppen von Anwendern unterscheiden:

1. Entwickler
2. Reviewer
3. Bauhausentwickler
4. Bauhaus-Systemtest

Entwickler Ein Entwickler nutzt Bauhaus, um den von ihm geschriebenen Quellcode zu analysieren und anschließend zu optimieren. Die Analyse kann entweder sehr spezifisch geplant sein und darauf ausgerichtet, wenige Kriterien genau zu prüfen, oder sie wird eher breit angelegt, um möglichst viele Aspekte abzudecken.

Im ersten Fall wird der Entwickler eine bestimmte Analyse öfter wiederholen, und zwischen den Ausführungen seinen Code optimieren. Das bedeutet, er muss eine Reihe von Werkzeugen immer wieder in der richtigen Reihenfolge ausführen. Dafür muss der Anwender jedes Werkzeug einzeln aufrufen. Durch die Teilweise sehr hohe Analysedauer mancher Werkzeuge muss er oft lange warten, bevor er das nächste Werkzeug starten kann. Das bedeutet, er muss immer wieder seine Arbeit unterbrechen, um nachzusehen, ob das zuletzt gestartete Werkzeug bereits beendet ist, um danach das nächste Werkzeug zu starten, weshalb eine solche Analyse seine sonstige Arbeit blockieren und seinen Durchsatz deutlich verringern würde.

Im zweiten Fall wird der Anwender viele verschiedene Werkzeuge ausführen. Da es momentan über 100 Werkzeuge gibt, die häufig über verschiedene Abhängigkeitspfade erreichbar sind, kann man nicht davon ausgehen, dass der Anwender alle Abhängigkeiten zwischen Werkzeugen kennt, weshalb dieser während dieses Vorgangs des öfteren die Dokumentation danach durchsuchen muss. Des weiteren haben die Bauhaus Werkzeuge keine einheitliche Parametrisierung, sodass ein Anwender nicht nur die Abhängigkeiten zwischen den Werkzeugen, sondern auch die für den Aufruf erforderlichen Parameter recherchieren muss.

Reviewer Ein Reviewer hat das Ziel, ihm vorgelegten Code zu Analysieren, um zu entscheiden, ob beispielsweise eine Software eingekauft werden soll, oder ein Softwareprodukt so freigegeben werden kann. Dieser Anwendungsfall ist mit dem zweiten, im Abschnitt 2.1 beschriebenen Fall vergleichbar. Der Reviewer würde ebenfalls eine breit angelegte Analyse starten, bei der er viele verschiedene Werkzeuge ausführen muss, wobei er stets deren Abhängigkeiten und erforderlichen Parameter ermitteln muss.

Bauhaus-Entwickler Ein Bauhausentwickler, der ein neues Werkzeug entwickelt, möchte dieses im Zusammenhang mit den bereits existierenden Werkzeugen testen. Um die Funktionalität seines Werkzeugs generell zu testen, muss er geeignete Eingabedateien erzeugen, welche sein Werkzeug zur Analyse benötigt. Zusätzlich muss er sicherstellen, dass alle möglichen Abhängigkeitspfade, über die sein Werkzeug erreichbar sein soll, auch zu einem Sinnvollen Ergebnis führen.

Hierzu muss der Bauhaus-Entwickler zunächst sämtliche Abhängigkeiten zu seinem Werkzeug auflösen und danach jedes einzelne Werkzeug auf diesen Abhängigkeitspfaden ausführen.

Bauhaus-Systemtest Beim Bauhaus-Systemtest werden alle Werkzeuge getestet. Hierzu müssen sämtliche Abhängigkeiten aufgelöst und Pfade gefunden werden, mit denen alle Werkzeuge abgedeckt werden können.

2.2. Abhängigkeiten

Die einzelnen Bauhaus Werkzeuge haben untereinander Abhängigkeiten, insofern, dass ein Werkzeug mit den Ergebnissen eines anderen Werkzeuges arbeitet, wodurch letzteres zwangsläufig zuerst ausgeführt werden muss. Betrachtet man diese Abhängigkeiten global, so lässt sich daraus ein Abhängigkeitsgraph konstruieren. Der so generierte Graph enthält nur direkte Abhängigkeiten. Für eine Analyse ist es jedoch häufig sinnvoll, nicht nur die minimalen Berechnungen durchzuführen, sondern weiterführende Analyseschritte dazwischen einzuschieben, weshalb es ratsam ist, einen Abhängigkeitsgraphen zu betrachten, der neben direkten auch indirekte Abhängigkeiten enthält. Haben zwei Werkzeuge a und b die gleiche direkte Abhängigkeit, können also beide die gleiche Eingabe verarbeiten, Werkzeug a kann jedoch auch das Ergebnis, welches von b direkt, oder über beliebig viele andere Werkzeuge erzeugt wurde verarbeiten, ist a indirekt von b abhängig, genauso wie von den anderen gegebenenfalls verwendeten Werkzeugen.

Ein beispielhafter Abhängigkeitsgraph wurde mir in Form einer `.dot` Datei bereitgestellt. Siehe Abbildung 2.1. Dieser Graph ist eine grobe Veranschaulichung der Abhängigkeiten in Bauhaus. Die vollständigen Abhängigkeiten sind der Dokumentation zu entnehmen [IST16a].

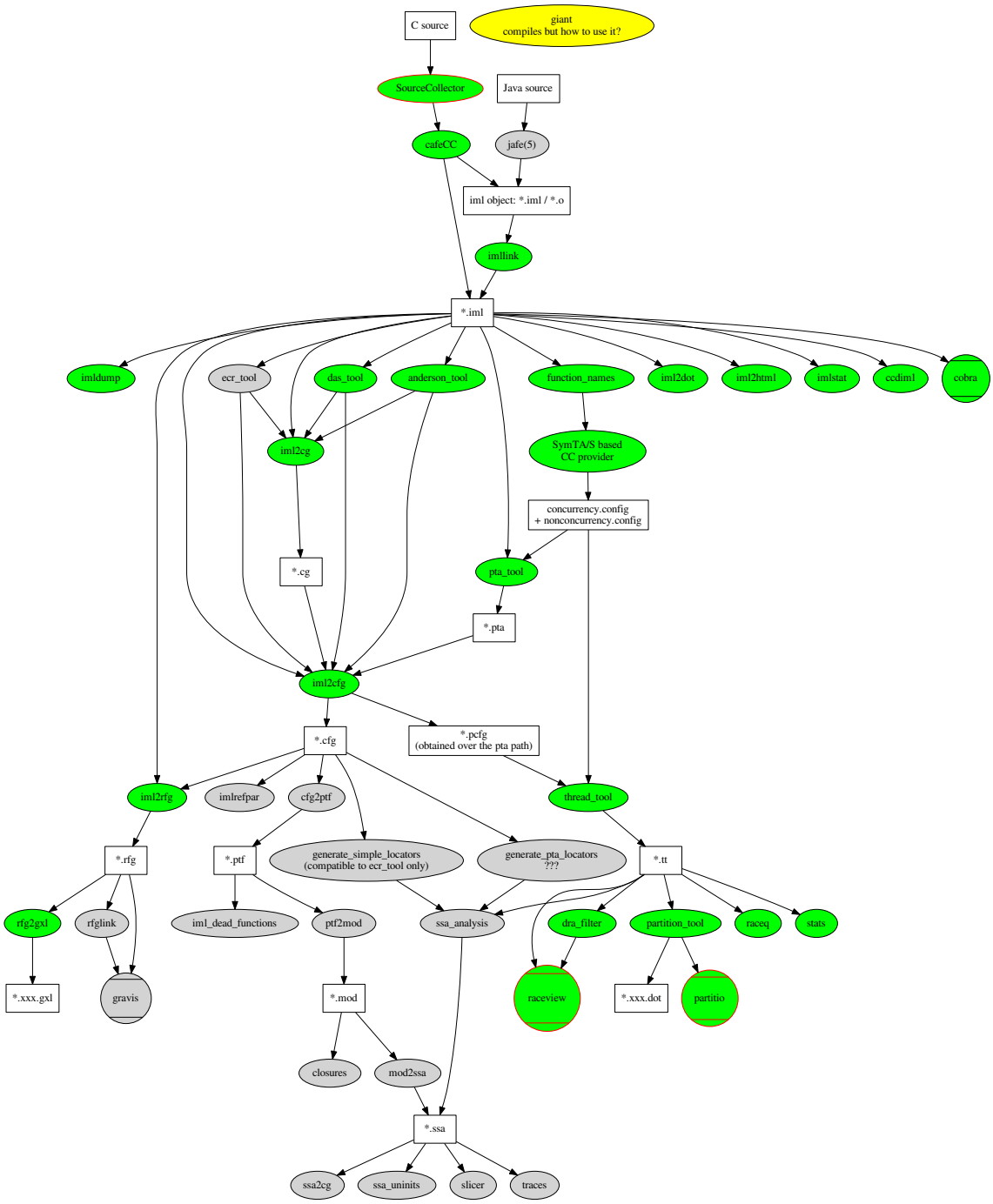

Abbildung 2.1.: Graphische Repräsentation des bereitgestellten Abhängigkeitsgraphen

In dieser Darstellung fehlen einige Knoten. Hier erscheint es so, dass manche Werkzeuge eine Ausgabedatei erzeugen, andere jedoch direkt mit anderen Werkzeugen verbunden sind, ohne dass eine Datei dazwischen generiert wird. Diese Fälle gibt es bei den dargestellten Werkzeugen jedoch nicht. Jedes Werkzeug, welches in diesem Graph ausgehende Kanten besitzt, erzeugt eine Datei als Ausgabe. Einige Werkzeuge erzeugen keine Dateien, sondern stellen eine graphische Oberfläche bereit um ihre Ergebnisse anzuzeigen. Diese Werkzeuge besitzen in dem Graph keine Ausgehenden Kanten. Es gibt jedoch auch Werkzeuge, die in diesem Graphen keine ausgehenden Kanten besitzen, aber trotzdem eine Ausgabedatei erzeugen. Dies Bedeutet, dass in einem vollständig definierten Abhängigkeitsgraphen für Bauhaus nur Kanten zwischen Werkzeugen und Dateitypen existieren.

Betrachtet man den Vollständigen Abhängigkeitsgraphen genauer, so kann man folgendes Feststellen:

Ein Werkzeug kann immer nur eine Datei als Ein- oder Ausgabe verwenden. Es ist in Bauhaus nicht möglich ein Werkzeug als Eingabe für ein anderes Werkzeug zu benutzen, oder als Ergebnis eines anderen Werkzeugs zu erzeugen. Da eingehende Kanten in einen Knoten, der ein Werkzeug repräsentiert eine mögliche Eingabe, und Ausgehende Kanten eine Ausgabe darstellen, können keine Kanten zwischen zwei Werkzeugen vorkommen. Da Dateien keine Ein- oder Ausgabe erzeugen können, sondern lediglich als solche dienen, kann auch keine Kante zwischen zwei Dateitypen vorkommen. Demzufolge ist der Abhängigkeitsgraph bipartit.

Da eine Kante entweder Ein- oder Ausgabe darstellt, zeigt sie zwangsläufig in eine Richtung. Für eine Eingabe muss eine Datei in ein Werkzeug hineingegeben werden, für eine Ausgabe gibt das Werkzeug die Datei heraus. Daraus folgt, dass der Graph gerichtet ist.

Rein theoretisch ist es keine zwingende Eigenschaft des Abhängigkeitsgraphen keine Zyklen zu enthalten, da er nicht nur direkte, sondern auch indirekte Abhängigkeiten enthält. Würde der Graph nur direkte Abhängigkeiten enthalten, so wären Zyklen ausgeschlossen, da ein Werkzeug nie ausgeführt werden kann bevor es ausgeführt wird, es kann also nicht direkt von sich selbst abhängen. Es spricht jedoch zunächst nichts dagegen, dass ein Werkzeug seine eigene Ausgabedatei als Eingabe akzeptiert. Ich gehe trotzdem von der Annahme aus, dass der Graph azyklisch ist, da es in der Praxis keinen Sinn ergibt, einem Werkzeug seine Eigene Ausgabedatei als Eingabedatei zu übergeben. In diesem Fall steht dem Werkzeug keine neue Information zur Verfügung, die bei der ersten Ausführung nicht vorhanden war. Ein weiterer Grund dafür ist, dass Dateitypen in dem Abhängigkeitsgraphen den Informationsgehalt darstellen. Durch weitere Analysen werden immer mehr Informationen gesammelt, welche in die jeweilige Ausgabedatei geschrieben werden, was bedeutet, dass eine Datei mehr Informationen enthält, je weiter ihr Dateityp im Abhängigkeitsgraphen von einer Quelle des Graphen entfernt ist. Betrachtet man nun einen Fall, indem dem Ergebnis eines Werkzeuges von einem anderen Werkzeug weitere Informationen hinzugefügt werden, mit denen das erste Werkzeug nun ebenfalls neue Erkenntnisse sammeln kann, so macht es keinen Sinn den gleichen Dateityp für die Ausgabe zu benutzen, da dieser für den geringeren Informationsgehalt steht. Erzeugt man hier nun einen weiteren Dateityp, der dem ursprünglichen zwar ähnelt, jedoch verdeutlicht, dass hier mehr Informationen enthalten sind, kann man das Werkzeug ein zweites Mal mit

diesem Dateityp als Eingabe und einer veränderten Ausgabe konfigurieren. Nun kann das gleiche Werkzeug an einer anderen Stelle im Graph eine andere Rolle einnehmen, und damit auch indirekt seine eigene Ausgabe akzeptieren, jedoch nicht erneut die gleiche Ausgabe erzeugen, wodurch hier kein Zyklus entsteht.

Zyklen im Graph zu vermeiden ist notwendig um zu gewährleisten, dass bei gleichem Aufruf auch die gleichen Werkzeuge ausgeführt werden. Ein Zyklus im Graph kann beliebig oft durchlaufen werden, sodass ein Pfad in dem Abhängigkeitsgraph nicht eindeutig identifizierbar ist, weshalb man bei der Ausführung entweder eine feste Anzahl an Durchläufen wählen, oder den Zyklus ignorieren muss. Beides ist auf die oben genannte Weise machbar, ohne den Zyklus wirklich in den Abhängigkeitsgraphen einzubauen.

Es ergibt sich also die Annahme, dass es sich bei dem Abhängigkeitsgraphen um einen bipartiten, gerichteten, azyklischen Graphen handelt, wobei sich die Annahme der Freiheit von Zyklen, lediglich auf den für die automatische Analyse angepassten Graphen bezieht, jedoch nicht zwangsläufig so sein muss. Diese Annahme habe ich aus praktischen Gründen getroffen, um eine automatisierte Analyse, bei welcher der Ablauf durch den Aufruf des Steuerungs-Werkzeugs vollständig definiert ist, zu ermöglichen.

Ich betrachte in dieser Arbeit nur sinnvolle, also nicht alle möglichen Analysepfade. Es ist Beispielsweise möglich, ein Werkzeug zu verwenden, welches bei jeder Ausführung zufällig irgendeine Analyse ausführt. Ein solches Werkzeug wäre nach den oben genannten Bedingungen nicht zu konfigurieren, es wäre jedoch in keinem Anwendungsfall sinnvoll, ein solches Werkzeug zu benutzen. Ich betrachte solche Werkzeuge, welche bei gleichem Aufruf, das heißt bei gleicher Eingabedatei und gleicher Parametrisierung, auch jedes Mal die gleiche Analyse durchführen, um selbst Konsistenz zu gewährleisten.

2.3. Vorteile eines Steuerungs-Werkzeugs

Der größte Vorteil eines Werkzeugs zu Steuerung von Bauhaus ist die Analyse in Unkenntnis der Abhängigkeiten. Kennt das Steuerungs-Werkzeug sowohl die Abhängigkeiten zwischen den Werkzeugen, als auch die nötigen Parameter um ein Werkzeug auszuführen, so muss ein Anwender nur noch angeben was er Analysieren möchte und das Steuerungs-Werkzeug übernimmt die Analyse für ihn.

Selbst wenn der Anwender alle Abhängigkeiten kennt, nimmt ihm das Werkzeug Arbeit ab, da er für jeden Pfad des Abhängigkeitsgraphen, welchen er ausführen möchte, nur einen Aufruf benötigt, anstatt für jedes Werkzeug einen Aufruf zu benötigen.

Ein weiterer Vorteil den ein solches Werkzeug bietet, ist die Möglichkeit eine Analyse zu Wiederholen. Ohne so ein Werkzeug ist die Wiederholung einer Analyse genauso aufwendig, wie die erste Analyse.

Ein Vorteil eines Werkzeuges zur Steuerung gegenüber dem bereits existierenden Make basierten Ansatz ([IST16b]) ist beispielsweise, dass der gewählte Analysepfad nicht in den Dateinamen kodiert werden muss. Außerdem ist es mit diesem Ansatz nicht möglich Werkzeuge wie `cobra` auszuführen, die keine Datei generieren, sondern eine graphische Oberfläche bereitstellen.

Das Steuerungs-Werkzeug ist nicht auf Bauhauswerkzeuge beschränkt. Man könnte beispielsweise einen Browser als Anzeigewerkzeug konfigurieren, welches `.html` Dateien Anzeigen kann oder aus generierten `.dot`-Dateien mit Hilfe der Anwendung `dot` eine `.pdf`-Datei erstellen.

2.4. Annahmen

In Kombination mit der Eingabe ist also nicht nur das Werkzeug, sondern auch die ausgeführte Analyse eindeutig identifizierbar. Der Name des Werkzeugs alleine reicht jedoch nicht aus, wie man an dem Beispiel des Werkzeuges iml2cfg sieht. Wird diesem Werkzeug eine Datei vom Typ `.pta` übergeben, so wird eine Datei von Typ `.pcfg` erzeugt, bei anderen Eingaben eine vom Typ `.cfg`. Eine `.pcfg` Datei kann von anderen Werkzeugen benutzt werden als eine `.cfg` Datei. Diese Variante wird genutzt um einen bestimmten Pfad zu bedingen. Für die Analysen, welche mit einer `.pcfg` Datei weiterführend durchgeführt werden können, ist es notwendig zuerst die Notwendigen Analysen durchzuführen um eine `.pta` Datei zu erstellen. Dies ist wichtig um während der Ausführung stets das richtige Werkzeug zu verwenden. Dies bedeutet ebenfalls, dass man bei bekannter Eingabe und Namen auf den genauen Aufruf des Werkzeuges schließen kann, was wiederum bedeutet, dass die erzeugte Ausgabe eines Werkzeugs ebenfalls durch die Eingabe und den Namen definiert ist. Die erzeugte Ausgabe eines Werkzeuges eindeutig der Eingabe zuzuordnen bedeutet, bei gleicher Eingabe auch die gleiche Ausgabe zu erzeugen. Dies schränkt die Ausführung nicht ein, sondern lediglich die Variation an Dateiendungen, die man einer Datei mit dem gleichen Informationsgehalt geben kann. In der Praxis ist es einem Werkzeug oft egal, welche Endung die Datei hat, solange sie die richtigen Informationen enthält. Dies erleichtert die Konfiguration, da man jedem Werkzeug einen eindeutigen Ausgabetyp zuweisen kann, ohne dabei die Funktionalität oder die Kompatibilität zwischen Werkzeugen zu beeinflussen.

Für jedes Werkzeug kann mit dem Parameter `-config_dir` der Pfad zu einer Werkzeugkonfiguration angegeben werden. Dieser Parameter wird in Bauhaus–commandline.adb definiert und sollte daher in jedem Werkzeug vorhanden sein.

Jedes Werkzeug kann aus der eingegebenen Datei auslesen, mit welchen Parametern zuvor ausgeführte Analysen gelaufen sind. Dies kann aus den `Tool_Infos` von Bauhaus ausgelesen werden, welche jedes Werkzeug pflegen sollte.

2.5. Voraussetzungen für die Ausführung

Um eine korrekte Ausführung zu ermöglichen, müssen alle Werkzeuge, die verwendet werden sollen gebaut und ausführbar sein. Ausführbar bedeutet hier, dass jedes Werkzeug in Unkenntnis des genauen Verzeichnisses in dem es liegt, aufgerufen werden kann, so als würde man das Werkzeug über eine Konsole starten.

3. Methodik der Umsetzung

3.1. Konfigurierbarkeit

Die Konfiguration des Analyse-Werkzeugs muss einerseits die Abhängigkeiten zwischen den Werkzeugen und andererseits die Parametrisierung für jedes einzelne Werkzeug beinhalten. Um diese Anforderung der Konfigurierbarkeit des Analyse-Werkzeugs abzudecken, habe ich mich dafür Entschieden, für jedes zu Konfigurierende Werkzeug eine eigene Konfigurationsdatei zu erstellen. Zusätzlich zu den Werkzeugkonfigurationen gibt es noch eine Basiskonfiguration.

Mit einer Konfigurationsdatei müssen verschiedene Arten von Werkzeugen abgedeckt werden. Standard Analyse Werkzeuge, Anzeigewerkzeuge und Konfigurationswerkzeuge.

Standard Analyse Werkzeuge sind die häufigste und einfachste Form. Sie bekommen eine Eingabedatei zur Analyse und generieren eine Ausgabedatei, die gesammelte Informationen enthält. Die erzeugten Ausgabedateien können von anderen Werkzeugen weiter verwendet werden.

Anzeigewerkzeuge bekommen ebenfalls eine Eingabedatei, die sie Analysieren, das Ergebnis der Analyse wird jedoch nicht in eine weitere Analysedatei geschrieben, sondern so aufbereitet, dass der Anwender diese Informationen sehen kann. Hierzu können entweder Dateien in einem zur Anzeige geeigneten Format erstellt werden, wie Beispielsweise `.dot` oder `.html`, oder das Werkzeug stellt selbst eine graphische Oberfläche bereit.

Die dritte Art Werkzeug, die Konfigurationswerkzeuge, erstellen aus der eingegebenen Datei eine Konfigurationsdatei, die von anderen Tools benutzt wird. Diese Datei wird dann an einen konfigurierten Ort gespeichert, wo sie von dem zu konfigurierenden Werkzeug genutzt werden kann.

Abhängig von der Art des Werkzeugs, enthält die Konfigurationsdatei andere Werte.

Für alle Arten von Werkzeugen enthält die Konfigurationsdatei den Namen des Werkzeugs, die konfigurierten Parameter und die möglichen Eingabedateien. Des Weiteren können jedem Werkzeug Prädikate hinzugefügt werden. Die Bedeutung der Prädikate wird im Abschnitt 3.4 erklärt. Der Name des Werkzeugs entspricht dabei dem Befehl um das Werkzeug in einer Konsole auszuführen. Es ist auch möglich eine `.sh`-Datei anzugeben. Die Parameter entsprechen den Parametern eines Aufrufs in der Konsole, wobei Ein- und Ausgabedateien durch Escapesequenzen dargestellt werden, die dann während der Ausführung durch die passenden Dateinamen ersetzt werden. Für die möglichen Eingabedateien werden nur die Endungen,

beginnend mit einem Punkt definiert. Eingabedateien bilden eingehende Kanten im Abhängigkeitsgraphen ab, wobei der Abhängigkeitsgraph für den entsprechenden Dateityp einen Knoten enthält, von dem dann eine Kante zu dem Konfigurierten Werkzeug führt.

Für Standard Analyse Tools enthält die Konfiguration mögliche Ausgabedateien. Wie bei den Eingabedateien werden für die Ausgabedateien lediglich die Endungen angegeben. Es ist möglich, hier mehrere Werte anzugeben, was jedoch bedeutet, dass aus allen konfigurierten Eingabedateien alle möglichen Ausgabedateien erzeugt werden können, abhängig vom entsprechenden Aufruf oder von nachfolgenden Werkzeugen.

Ein Beispiel für ein Standard Werkzeug wäre `pta_tool`. Der Name ist `pta_tool`, da das Werkzeug so aufgerufen wird. Als mögliche Eingabedateien können beispielsweise `.iml` und `.cfg` konfiguriert werden. Beide Eingaben erzeugen eine Datei vom Typ `.pta` als Ergebnis. Um `pta_tool` auszuführen, gibt man zum Beispiel `pta_tool -o bsp.pta bsp.iml` in eine Konsole ein, wobei `bsp.pta` das Ergebnis und `bsp.iml` die Eingabedatei ist. In der Konfiguration würde für die Parameter also folgende Liste stehen: `["-o", "&o", "&i"]`. &o steht für die Ausgabe, also in diesem Fall `bsp.pta` und &i für die Eingabe, also `bsp.iml`.

Hat ein Werkzeug eine Restriktion, in der Art, dass eine bestimmte Eingabe auch eine bestimmte Ausgabe erzeugt, während andere Eingaben andere Ausgaben erzeugen, so muss dieses Werkzeug mehrmals konfiguriert werden.

Dies kann man am Beispiel des Werkzeuges `iml2cfg` sehen. Erhält dieses Werkzeug eine Datei vom Type `.pta` als Eingabe, so wird eine Datei vom Typ `.pcfg` generiert, bei anderen Eingaben, die beispielsweise vom Typ `.iml`, `.cg` oder `.ecr` sein können, wird jedoch eine Datei vom Typ `.cfg` erzeugt.

Für `iml2cfg` gäbe es also zwei verschiedene Konfigurationen: Beide hätten als Namen `iml2cfg` da der Aufruf der gleiche bleibt. Aus diesem Grund kann man auch die Parameter gleich konfigurieren, beispielsweise `["&i", "&o"]`, oder auch für beide Fälle unterschiedlich. Bei den Ein- und Ausgaben würden sich die Konfigurationen definitiv unterscheiden. Für den ersten Fall wären als mögliche Eingaben nur `.pta` konfiguriert und als Ausgaben nur `.pcfg`, im zweiten Fall wären `.iml`, `.cg` und `.ecr` als mögliche Eingaben und `.cfg` als Ausgabe konfiguriert. Beide Konfigurationen werden wie zwei unterschiedliche Werkzeuge betrachtet und auch so behandelt.

Konfigurationen für Anzeige und Konfigurationswerkzeuge enthalten jeweils einen entsprechenden booleschen Wert, der angibt, dass es sich um die jeweilige Art von Werkzeug handelt. Konfigurationen für Standard Analyse Werkzeuge können diese Beiden Parameter ebenfalls enthalten, diese müssen dann aber auf $false$ gesetzt werden.

Für Anzeige Werkzeuge können ebenfalls Ausgabedateien konfiguriert werden, wie bei Standard Analyse Tools, hier ist dieser Wert jedoch optional, da ein Anzeige Werkzeug zwar Dateien generieren kann, es jedoch nicht muss.

Das Werkzeug `cobra` ist ein Beispiel für ein Werkzeug, welches eine graphische Oberfläche bereitstellt und keine Ausgabedatei generiert. Die Konfiguration sähe wie folgt aus: Als Name

wäre `cobra` konfiguriert. Als Eingaben könnten Beispielsweise `.iml`, `.tt` und `.cfg` konfiguriert sein, die Liste für die Ausgabedateien wäre leer. Als Parameter reicht folgende Liste: `["&i"]`, denn `cobra` benötigt nur eine Eingabedatei. Um kenntlich zu machen, dass es sich um ein Werkzeug zur Anzeige handelt, würde der Wert *isToolForDisplay* auf *true* gesetzt werden.

Konfigurationen für Konfigurationswerkzeuge enthalten zusätzlich zu den bereits genannten Werten eine Liste an Werkzeugen, für welche sie eine Konfiguration bereitstellen. Zudem wird bei dieser Art Werkzeug der Pfad für die generierte Konfigurationsdatei als Ziel angegeben anstatt nur die Dateiendung.

Als Beispiel hier das Werkzeug `symta`: Als Name des Werkzeuges wird hier `symta.sh` angegeben, da es sich hierbei um eine `Java` Applikation handelt, die als `.jar`-Datei zur Verfügung steht. `.jar`-Dateien werden Momentan nicht unterstützt, weshalb ich hier den Aufruf `java -jar symta.jar` in die Datei `symta.sh` gekapselt habe um das Werkzeug ausführen zu können. `symta` erwartet zwei Dateien als Eingabe: Eine Datei vom Typ `.func_name`, welche die Ausgabe des Werkzeuges `function_names` darstellt und eine `.csv`-Datei. Momentan wird nur eine Eingabedatei unterstützt, weshalb hier die `.csv`-Datei in der Konfiguration fest angegeben werden muss. Die Parameter für `symta` könnten beispielsweise `["&i", "bsp.symta.csv"]` lauten und als Zieldatei wäre `Concurrency.config` angegeben. Die angegebene Zieldatei wird von dem Werkzeug in diesem Fall zwar nicht verwendet, es ist jedoch nicht vorgesehen für Konfigurationswerkzeuge keine Zieldatei anzugeben. Der Eingabetyp wäre `[".func_name"]` und da es sich um ein Konfigurationswerkzeug handelt, muss bei der Konfiguration der Wert für *isToolForConfig* auf `true` gesetzt werden.

Für jedes Werkzeug, egal von welchem Typ, kann zusätzlich eine Liste von Prädikaten angegeben werden. Die Bedeutung dieser Liste wird im Abschnitt 3.4 näher erläutert.

Die Basiskonfiguration enthält lediglich eine Liste an Pfaden zu den einzelnen Konfigurationsdateien, sowie einen Quelldateityp. Die Angabe des Quelldateityps erleichtert den Aufbau des Abhängigkeitsgraphen, da man diesen nicht erst ermitteln muss. Durch die Konfiguration der Pfade zu Konfigurationsdateien kann der Anwender mit geringem Aufwand verschiedene Konfigurationen, für verschiedene Sets von Werkzeugen pflegen, ohne jedes mal alle Dateien austauschen zu müssen.

Für jedes Werkzeug kann eigener Code in der Konfiguration angegeben werden, welcher nach der Ausführung des Werkzeuges ausgeführt wird. Hierfür wird der Pfad zu einer `.class`-Datei in der Konfiguration eingetragen. Beispielsweise für das Werkzeug `symta` mit dem Konfigurationseintrag `/home/baueras/resources/AfterSymta.class` für den Wert `afterExecutionClass`.

3.2. Automatisierte Analyse

Eine Analyse besteht aus zwei Teilen: dem Suchen des passenden Pfades im Abhängigkeitsgraphen und der Ausführung der Werkzeuge.

Da für die korrekte Funktionalität des Analyse-Werkzeuges vorausgesetzt wird, dass alle Werkzeuge gebaut und auch Verfügbar sind, müssen die entsprechenden Werkzeuge nur mit den jeweils konfigurierten Parametern Aufgerufen werden. Durch die Annahme, dass ein Pfad im Abhängigkeitsgraphen keine Verzweigung enthält, und somit auch nie zwei Werkzeuge parallel ausgeführt werden können, da jedes Werkzeug genau das Ergebnis eines vorangegangenen Werkzeuges benötigt, habe ich eine rein sequentielle Ausführung gewählt. Für jedes Zwischenergebnis wird dabei eine Temporäre Datei erstellt.

Den richtigen Pfad für eine Analyse zu berechnen, ist hier der wesentlich höhere Aufwand. Unabhängig von dem Anwendungsfall, muss der Pfad immer mit dem Dateityp der Eingabedatei beginnen und die Zieldatei zumindest enthalten. Außerdem muss der Pfad zusammenhängend sein und darf keine Elemente mehr als einmal enthalten, oder Verzweigungen beinhalten. Für den einfachsten Anwendungsfall hat der Pfad genau diese Eigenschaften, mit der Spezialisierung, dass die Zieldatei das Ende des Pfades darstellt. Er beginnt mit der Eingabe- und endet mit der Zieldatei. Ein so definierter Pfad ist in dem Graphen nicht eindeutig, also muss eine weitere Einschränkung gewählt werden. Aus diesem Grund habe ich hier die Annahme getroffen, dass eine so gestartete Analyse nur das Ziel hat eine Datei vom Typ der eingegebenen Zieldatei zu erstellen, ohne dass dabei mehr Analysen als notwendig ausgeführt werden. Um diesen Fall abzubilden, wird bei nicht näher spezifizierten Eingabe lediglich der kürzeste Pfad zwischen der Eingabe- und der Zieldatei ermittelt und ausgeführt. Zur Ermittlung des kürzesten Pfades benutze ich eine Breitensuche, bei der Vorgängerknoten gespeichert werden. Dies ist möglich, da der Abhängigkeitsgraph gerichtet und azyklisch ist.

Ein derart einfacher Aufruf schließt einige Werkzeuge zwangsläufig aus, da diese keine Ausgabedateien generieren, sondern eine graphische Oberfläche bereitstellen. Solche Werkzeuge müssen in einem Aufruf explizit genannt werden, um sie bei einer Analyse benutzen zu können. Die explizite Nennung eines oder mehrerer Werkzeuge, beschreibt einen genaueren Pfad für eine Analyse. Ein solcher Aufruf kann den Pfad einfach nur um ein entsprechendes Anzeigewerkzeug erweitern, er kann allerdings auch einen längeren weg zwischen Ein- und Ausgabe definieren. Zur Berechnung eines solchen Pfades, müssen die eingegebenen Werkzeuge und die Zieldatei zunächst in eine Sinnvolle Reihenfolge gebracht werden. Hierzu wird zunächst eine Liste bestehend aus der Zieldatei und jedem eingegebenen Werkzeug erstellt und jedem Element dieser Liste eine Zahl a_j zugeordnet, die angibt, wie viele der eingegebenen Werkzeuge im Abhängigkeitsgraph von diesem Werkzeug aus erreicht werden können. Dies wird mittels Breitensuche ermittelt. Diese Zahlen haben folgende Eigenschaften:

- $0 \leq i \leq n$
- $0 \leq a_i \leq n$

- Für alle a_i, a_j mit $i \neq j$ gilt $a_i \neq a_j$

n ist hier die Anzahl der explizit eingegebenen Werkzeuge. Ist eine dieser Eigenschaften nicht erfüllt, ist die Eingabe ungültig, da die Elemente der Liste nicht auf dem gleichen Pfad liegen. Die erste Eigenschaft bedeutet, dass jedem Element eine eigene Zahl zugeordnet wird. Die zweite Eigenschaft ist immer erfüllt, da eines der Elemente nicht mehr Nachfolger haben kann, als Elemente existieren, die ein Nachfolger sein könnten. Die dritte Eigenschaft stellt sicher, dass alle eingegebenen Elemente auf dem gleichen Pfad liegen, da in einem Pfad nie zwei Elemente die gleiche Anzahl an Nachfolgern besitzen können. Sortiert man diese Liste nun absteigend nach den a_i, erhält man die korrekte Reihenfolge für die Ausführung. Zwischen den Werkzeugen können durchaus noch weitere Werkzeuge liegen, die nicht explizit eingegeben wurden. Aus diesem Grund wird nun jeweils der kürzeste Pfad zwischen zwei aufeinanderfolgenden Elementen ermittelt. Anschließend werden diese Pfade in der entsprechenden Reihenfolge aneinandergereiht, sodass ein valider Pfad entsteht. Die Zieldatei kann dabei an beliebiger Stelle im Pfad stehen. Falls ein Anzeigewerkzeug angegeben wurde, muss es am Ende stehen, da es keine Ausgabedatei erzeugt und somit keine weiterführende Analyse ermöglicht. Dabei erhält dieses Werkzeug nicht zwangsläufig die letzte erzeugte Datei, sondern die letztmögliche Datei, die verarbeitet werden kann. Dies ermöglicht die Durchführung von Analysen, deren Ergebnis nicht angezeigt werden kann, bei gleichzeitiger Anzeige eines Zwischenergebnisses.

Eine weitere Möglichkeit ist die Nutzung eines konfigurierten Präferenzmechanismus. Diese Variante wir im Abschnitt 3.4 genauer beschrieben.

Die beiden Möglichkeiten zur genaueren Spezifikation des gewünschten Pfades sind kombinierbar. Werden beide zusammen verwendet, so wird zwischen den angegebenen Werkzeugen der durch den Präferenzmechanismus beschriebene Pfad gewählt, anstatt den kürzesten zu nehmen.

Sämtliche Varianten dieser Berechnung werden nicht auf dem vollständigen Abhängigkeitsgraphen ausgeführt, sondern auf dem Subgraph, der von der eingegebenen Quelldatei aus erreichbar ist. Jeder valide Pfad geht dadurch von der Wurzel dieses Subgraphen aus, sodass keine unerreichbaren Werkzeuge berücksichtigt werden. Ist ein Werkzeug oder die Zieldatei nicht in diesem Subgraph enthalten, so ist eine solche Analyse nicht möglich, beziehungsweise nicht vorgesehen.

Bei der Automatisierten Analyse findet eine Linearisierung der Abhängigkeitspfade statt, sodass kein Ausführungspfad eine Verzweigung besitzt. Dies bedeutet insbesondere, dass alle Werkzeuge nur eine einzige Datei als Eingabe erhalten. Dies bildet die Realität nicht vollständig ab, da manche Werkzeuge, wie beispielsweise $iml2cfg$ mehrere Dateien als Eingabe akzeptieren. Dies ermöglicht parallele Ausführungen von Werkzeugen. Zum Beispiel können `ecr_tool` und `das_tool` beide Ausgeführt werden und die Ergebnisse gemeinsam an `iml2cfg` übergeben werden. Diese Art von Analysen konnte ich aus Zeitgründen nicht mehr berücksichtigen, sie könnten aber in einer Zukünftigen Arbeit behandelt werden.

3.3. Wiederholbarkeit

Eine Analyse zu wiederholen bedeutet, die gleichen Werkzeuge mit der gleichen Eingabe ein weiteres Mal auszuführen. Um dies zu ermöglichen, muss der genaue Ablauf einer Analyse gespeichert werden, sodass die selben Informationen bei einem zweiten Aufruf zur Verfügung stehen. Bei der nächsten Ausführung wird diese Information dann eingelesen und erneut Ausgeführt.

Es ist wichtig bei der Wiederholung sämtliche Informationen aus dem ursprünglichen Aufruf zu Verfügung zu haben, sodass dieser genau nachvollzogen werden kann, deshalb habe ich mich dafür entschieden, den genauen Aufruf mit sämtlichen Parametern als Identifikator für die gespeicherte Ausführung zu verwenden. Des weiteren wird eine Liste von verwendeten Dateitypen gespeichert, wobei das erste Listenelement dem Typ der Eingabedatei entspricht und das letzte Element dem Typ der Zieldatei.

Außerdem werden alle Ausgeführten Werkzeuge mit den Verwendeten Parametern gespeichert, wobei die Ein- und Ausgabedateien durch Escapesequenzen ersetzt werden, die bei der Ausführung dann durch sinnvolle Dateinamen ersetzt werden können. Ein Aufruf wird in genau einer Textdatei gespeichert, wodurch diese durch den Anwender angepasst werden kann. Beim Einlesen der gespeicherten Datei wird überprüft, ob die verwendeten Dateiformate zu den jeweiligen Werkzeugen passen, bevor diese Ausgeführt werden. Ansonsten wird die Konfiguration bei der Wiederholung nicht berücksichtigt, sodass hier die Parametrisierung von Werkzeugen einmalig verändert werden kann, ohne die Konfiguration anpassen zu müssen. Des weiteren kann so ein Werkzeug ersetzt, entfernt, oder ein anderes eingefügt werden.

Die Wiederholung kann explizit Ausgeführt werden, wird jedoch auch implizit genutzt, wenn der Aufruf mit dem Identifikator des letzten Aufrufs übereinstimmt und es nicht explizit gewünscht ist keine Wiederholung durchzuführen. Die Implizite Wiederholung wird immer vollständig ausgeführt und unterscheidet sich daher nicht wesentlich vom ursprünglichen Aufruf. Wird die Wiederholung jedoch explizit angestoßen, so kann der Anwender wählen, ob er die Analyse vollständig, oder nur teilweise wiederholen will. Hierzu werden die bei der letzten Ausführung verwendeten Dateitypen ausgegeben, sodass der Anwender wählen kann, welchen Teil er davon wiederholen will.

3.4. Präferenzmechanismus

Der Präferenzmechanismus dient dazu, aus mehreren möglichen Pfaden einen bestimmten auszuwählen, ohne diesen bei jeder Ausführung explizit angeben zu müssen. Um dies zu ermöglichen, habe ich die Konfiguration von Prädikaten, die den entsprechenden Werkzeugen zugeordnet werden können, hinzugefügt. Diese Prädikate werden in der Konfiguration angegeben, wobei ein Werkzeug mehrere Prädikate haben kann. Beim Aufruf wird dann ein gewünschtes Prädikat angegeben, und danach der entsprechende Pfad ermittelt. Hierbei wird

vom eingegebenen Dateityp aus dem Pfad, der durch das eingegebene Prädikat beschrieben ist, gefolgt, solange die Zieldatei noch erreichbar ist.

Der durch Prädikate markierte Pfad kann Werkzeuge enthalten, denen ein anderes oder gar kein Prädikat zugeordnet ist, weshalb zwischen zwei markierten Werkzeugen immer der kürzeste Pfad ermittelt wird. Um das nächstgelegene Werkzeug mit dem definierten Prädikat zu finden, wird eine Breitensuche ausgeführt, bei der ein Pfad nicht weiter verfolgt wird, wenn bereits ein Werkzeug mit dem Definierten Prädikat in diesem Pfad gefunden wurde. So erhält man alle vom momentanen Knoten direkt erreichbaren Werkzeuge mit dem Prädikat. Wird in einem Schritt mehr als ein Werkzeug gefunden, so wird, mit der gleichen Methode wie im Abschnitt 3.2 für explizit eingegebene Werkzeuge beschrieben, die Reihenfolge der gefundenen Werkzeuge ermittelt. Es muss unter den gefundenen Werkzeugen immer genau ein erstes geben, da ein Prädikat genau einen Pfad markieren soll. Gibt es mehr als ein erstes Werkzeug, so wurden mehrere parallele Pfade mit dem gleichen Prädikat markiert, wodurch das Prädikat nicht mehr genau einem Pfad zugeordnet werden kann und deshalb keine eindeutige Ausführung mehr zulässt. Eine solche Konfiguration ist also ungültig. Bei korrekter Konfiguration wird das gefundene erste Werkzeug nun genutzt, um von dort aus das nächste mit dem entsprechenden Prädikat zu suchen.

Es kann immer nur ein gefundenes Werkzeug zum Pfad direkt hinzugefügt werden, da nicht gewährleistet ist, dass das nächste Werkzeug im durch das Prädikat markierten Pfad von dem momentan als Ausgangspunkt benutzten Werkzeug direkt erreichbar ist.

Der Pfad ist vollständig, wenn entweder die Zieldatei das Ergebnis eines mit dem Prädikat markierten Werkzeuges ist, oder kein Pfad vom nächsten gefundenen Werkzeug zu dem Zieldateityp existiert.

Dies wird am Beispiel in Abbildung 3.1 deutlich.

Der markierte Pfad sei $abcdef$ und die Ausgabe von Werkzeug h die Zieldatei. Hier sind bis auf c alle Knoten von a aus erreichbar, sodass die Reihenfolge ermittelt werden muss. Das Ergebnis hiervon ist nun b als direktester Nachfolger as. Hier sieht man nun, dass das vorherige Ergebnis nicht weiter benutzt werden kann, da es c nicht enthält, und c hier der direkteste Nachfolger bs ist. Auf diese Weise wird der Pfad nun bis d verfolgt. Nun wäre der nächste Knoten e, es führt jedoch kein Pfad von e zu h, sodass der Pfad nun noch um den kürzesten Pfad von e zu h ergänzt wird und darauf das Ergebnis $abcdgh$ lautet.

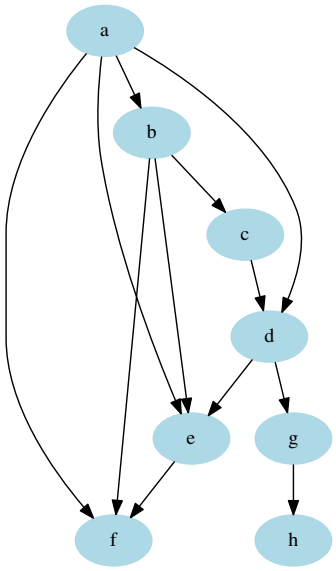

Abbildung 3.1.: Beispiel für die Pfadsuche mit Prädikat

4. Realisierung

In diesem Kapitel ist die Implementierung des Steuerungs-Werkzeugs beschrieben. Die Umsetzung des Steuerungs-Werkzeuges erfolgte in Java. Für das Einlesen und Parsen der `JSON` Dateien habe ich das Paket `org.json` (Siehe [16d]) verwendet.

4.1. Architektur

Das Werkzeug ist in fünf Pakete unterteilt:

- graph
- executionorder
- toolexecution
- settings
- main

Im Paket `main` befindet sich lediglich die Klasse `Main`, welche den Programmablauf steuert.

Im Paket `settings` befindet sich sämtliche Logik zum einlesen und verwerten der Konfiguration.

Das Paket `executionorder` enthält die Logik zum einlesen und erstellen der Datei `LastExecutionOrder.txt`.

Sämtliche Logik zum Aufbau des Abhängigkeitsgraphen und zur Suche auf diesem, sowie Klassen zur Repräsentation von Werkzeugen und Dateien befinden sich im Paket `graph` zusammen mit der Logik zum erstellen einer `.dot` Datei.

Im Paket `toolexecution` befindet sich die Logik um Werkzeuge auszuführen.

4.1.1. Programmablauf

Beim Aufruf des Werkzeugs müssen zunächst drei verschiedene Ausführungsarten berücksichtigt werden:

1. Start einer Analyse
2. Wiederholung einer Analyse
3. Hilfsfunktion zu Unterstützung des Anwenders

Zu Beginn einer Ausführung wird zunächst geprüft, ob die Datei `LastExecutionOrder.txt` existiert und falls ja wird diese Eingelesen und der Identifikator in der ersten Zeile mit den übergebenen Parametern verglichen. Stimmen die Parameter mit dem Identifikator überein, so wird eine implizite Wiederholung gestartet. Ist dies nicht der Fall, oder ist der Parameter `-f` beziehungsweise `--force-new` gesetzt, so wird eine neue Analyse gestartet.

Start einer Analyse Wird eine Analyse gestartet, so wird zunächst die Konfiguration eingelesen und daraus der Abhängigkeitsgraph aufgebaut. Anschließend wird in diesem der Pfad zur Ausführung, wie im Abschnitt 4.1.2 beschrieben, gesucht. Existiert für eines der im gefundenen Pfad enthaltenen Werkzeuge eine Konfigurationswerkzeug, welches für dieses eine Konfiguration erzeugt, so wird ein Pfad von der Eingabedatei zu diesem Konfigurationswerkzeug gesucht und ausgeführt. Ist das Konfigurationswerkzeug nicht erreichbar, wird die Ausführung ohne dieses Werkzeug fortgesetzt. Wurde ein gültiger Pfad gefunden und für jedes Werkzeug falls möglich eine Konfiguration erstellt, werden die darin enthaltenen Werkzeuge sequentiell ausgeführt. Der gefundene Pfad wird während der Ausführung, wie in Abschnitt 4.2.4 beschrieben, in die Datei `LastExecutionOrder.txt` gespeichert.

Wie im Abschnitt 3.2 erwähnt, können derzeit nicht zwei Dateien zusammen an ein Werkzeug übergeben werden. Wäre dies möglich, könnte es parallele Pfade geben, die in einem Werkzeug mit zwei Eingabedateien wieder vereinigt werden. Ohne diese Variante, ist eine parallele Ausführung von Werkzeugen nicht möglich, weshalb momentan nur eine sequentielle Ausführung stattfindet.

Wiederholung einer Analyse Es gibt drei Möglichkeiten, eine Wiederholung der letzten Analyse auszuführen:

1. Implizit
2. Explizit vollständig
3. Explizit teilweise

Eine implizite Wiederholung wird ausgeführt, wenn die gleichen Parameter beim Start übergeben werden, wie bei der letzten Analyse. Der Parameter `-f` beziehungsweise `--force-new` wird dabei nicht berücksichtigt. Wird dieser Parameter angegeben, so wird keine implizite Wiederholung gestartet.

Zu Beginn einer impliziten oder einer vollständigen expliziten Wiederholung wird die Datei `LastExecutionOrder.txt` eingelesen und aus deren Inhalt die ausgeführte Analyse ermittelt. Anschließend wird die Konfiguration eingelesen, um sicherstellen zu können, dass die eingelesenen Werkzeuge zu den Dateiendungen passen. Die Parameter werden nicht überprüft, da hier Änderungen möglich sein sollen. Wurden keine Fehler in der Datei festgestellt, so werden die Werkzeuge mit den eingelesenen Parametern ausgeführt.

Bei einer teilweisen Wiederholung wird der Anwender nach dem Einlesen der gespeicherten Analyse gefragt, welchen Teil davon er wiederholen möchte. Hierzu wird dem Anwender die Liste an Dateiendungen jeweils mit einem Index versehen angezeigt, sodass dieser nun Wählen kann, wo die Analyse starten und enden soll. Nun gibt der Anwender zwei Zahlen ein und wenn diese sinnvoll sind, das heißt die Zahlen befinden zwischen 0 und dem höchsten Index und der Start-Index ist kleiner als der Ziel-Index, so wird dieser Teil der Analyse ausgeführt.

Bei dieser Auswahl werden Anzeigewerkzeuge noch nicht berücksichtigt. Wurde bei der vorherigen Analyse ein solches Werkzeug ausgeführt, so wird der Anwender gefragt, ob er dieses ebenfalls ausführen will.

Hilfsfunktionen Dem Anwender stehen verschiedene Hilfsfunktionen zur Verfügung, die ihm die Bedienung des Werkzeugs erleichtern sollen:

- Ausgabe aller möglichen Eingabeparameter
- Erstellen einer `.dot` Datei aus dem konfigurierten Abhängigkeitsgraphen
- Ausgabe aller Prädikate

Über den Parameter `-h` beziehungsweise `--help` wird die Anzeige aller möglichen Parameter gestartet. Zu jedem Parameter wird eine kurze Beschreibung angezeigt, wozu dieser dient und ob nach diesem Parameter noch eine oder mehrere Eingaben erwartet werden.

Das erstellen einer `.dot` Datei wird über den Parameter `--dot` gestartet. Nach diesem Parameter muss der Name der zu erstellenden Datei angegeben werden. Diese Datei enthält sowohl den Aufbau des Abhängigkeitsgraphen, als auch die Prädikate, die für ein Werkzeug angegeben sind. Der durch diese Datei dargestellte Graph enthält für jede Konfiguration einen eigenen Knoten, sodass Werkzeuge hier, wie auch in der Konfiguration, mehrmals vorkommen können. In Werkzeugnamen, die einen `.` enthalten, wird dieser durch `_` ersetzt, um korrekte `.dot` Syntax zu gewährleisten.

Wird der Parameter `--show-predicates` angegeben, so wird eine Liste aller konfigurierten Prädikate angezeigt, ohne Informationen über die dazugehörigen Werkzeuge.

Ich habe diese Funktionen eingebaut, um es einem Anwender zu ermöglichen, Analysen durchzuführen und den vollen Umfang dieses Steuerungs-Werkzeuges zu nutzen, ohne dass er zusätzliche Informationen benötigt.

4.1.2. Pfadsuche

Grundlage der Pfadsuche in dieser Anwendung ist die Breitensuche. Eine Breitensuche kann hier nur auf Grund der günstigen Eigenschaften des Abhängigkeitsgraphen, im speziellen, dass es sich um einen azyklischen, gerichteten Graphen handelt, verwendet werden.

Je nach Aufruf müssen zwei verschiedene Fälle berücksichtigt werden:

1. Einfacher Aufruf
2. Angegebenes Prädikat und explizite Werkzeugangabe

Einfacher Aufruf Bei einem Einfachen Aufruf genügt eine einzelne Breitensuche. Hierzu wird ausgehend vom Dateityp der Eingabedatei eine Breitensuche ausgeführt, bei der zu jedem Knoten der Vorgänger gespeichert wird, bis der Dateityp der Zieldatei erreicht ist, oder der von dem Eingabedateityp erreichbare Teil des Abhängigkeitsgraphen vollständig durchsucht wurde. War die Suche erfolgreich, so wurde der kürzeste Pfad von der Eingabe- zu Zieldatei ermittelt.

Ein Beispiel für einen Einfachen Aufruf wäre `-s bsp.iml -t bsp.rfg`, also `bsp.iml` als Eingabe und `bsp.rfg` als Zieldatei. Hier wird nun der kürzeste Pfad zwischen `.iml` und `.rfg` berechnet, welcher lediglich das Werkzeug `iml2rfg` enthält. Es werden hier also keine Zwischenergebnisse erzeugt. Der einzige Werkzeugaufruf lautet nun `iml2rfg bsp.iml bsp.rfg`.

Prädikat und Werkzeugangabe Es ist möglich, nur eine der beiden Eingaben zu machen, also eine leere Menge an angegebenen Werkzeugen, oder eine leeres Prädikat zu benutzen. Wurden Werkzeuge explizit angegeben, so müssen diese zunächst, wie im Abschnitt 3.2 angegeben, in die richtige Reihenfolge gebracht werden.

Nachdem die Eingabedatei, die Ausgabedatei und sämtliche Werkzeuge in die richtige Reihenfolge gebracht wurden, wird nun schrittweise, der durch das angegebene Prädikat markierte Pfad, zwischen den einzelnen Elementen ermittelt und anschließend alle gefundenen Pfade aneinandergehängt, um einen Pfad für die Ausführung zu erhalten. Wurde kein Prädikat angegeben, so wird stattdessen der kürzeste Pfad ermittelt.

Der durch ein Prädikat markierte Pfad wird wie im Abschnitt 3.4 beschrieben ermittelt.

Da es möglich ist, ein Werkzeug zweimal für unterschiedliche Eingaben zu konfigurieren, muss dies zunächst für jedes Werkzeug geprüft werden. Falls dies für eines der eingegebenen Werkzeuge zutrifft, so wird diejenige Konfiguration genutzt, die von der Eingabedatei erreichbar

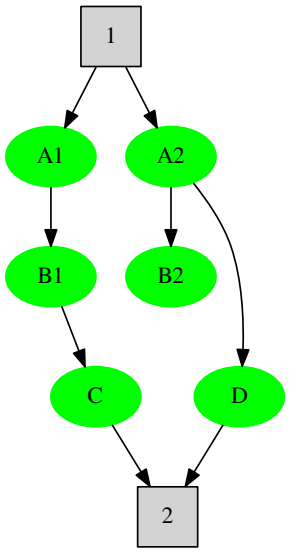

Abbildung 4.1.: Beispiel mit 2 verschiedenen Konfigurationen pro Werkzeug

ist und von deren Ausgabe aus der Zieldateityp erreichbar ist. Es wird hier nicht geprüft, ob mehrere Konfigurationen möglicherweise zum Ziel führen. Stattdessen wird die erste zutreffende genutzt. Um diese Unterscheidung sinnvoller umzusetzen, hatte ich nicht mehr genügend Zeit. Es wäre an dieser Stelle wünschenswert, diejenige Konfiguration zu wählen, welche im Zusammenhang mit den übrigen Werkzeugen und dem eingegebenen Prädikat am sinnvollsten erscheint, insofern dass möglichst alle Werkzeuge erreichbar sein sollten und der durch das Prädikat markierte Pfad möglichst lange verfolgt werden sollte.

Nur zu prüfen, ob alle nachfolgenden Werkzeuge und der Zieldateityp von einer der Konfigurationen aus erreichbar sind, genügt jedoch nicht. Es muss zusätzlich geprüft werden, ob diese auf dem selben Pfad liegen, wie das Beispiel in Abbildung 4.1 zeigt.

1 ist hier die Eingabedatei und 2 die Zieldatei. Es wurden hier die Werkzeuge A und B explizit angegeben. $B1$ und $B2$ sind Konfigurationen für B, $A1$ und $A2$ sind Konfigurationen für A. Betrachtet man nun die Konfigurationen für A so stellt man fest, dass sowohl das Werkzeug B als auch die Zieldatei 2 von beiden Konfigurationen erreichbar ist, jedoch nur von Konfiguration $A1$ aus ein Pfad zu 2 führt, der auch B enthält. Dies zeigt, dass stets der vollständige Pfad berücksichtigt werden muss.

Um dem durch ein Prädikat markierten Pfad möglichst lange zu folgen, müsste jeder mögliche Pfad über unterschiedliche Konfigurationen für eingegebene Werkzeuge ermittelt und anschließend derjenige genutzt werden, welcher die größte Anzahlt an Werkzeugen mit dem angegebenen Prädikat enthält.

Ein Beispiel für einen solchen Aufruf wäre `-s bsp.iml -t bsp.rfg -p fast using-tools iml2cg`.

Das Prädikat `fast` markiert in diesem Beispiel das Werkzeug `ecr_tool`. Die eingegebenen Werkzeuge werden sortiert, sodass sich die Reihenfolge `iml2cg`, `.rfg` ergibt.

Nun wird der Pfad mit Prädikat `fast` zwischen `.iml` und `iml2cg` ermittelt. Auf diesem Pfad liegt das Werkzeug `ecr_tool`, das `.iml` als Eingabe akzeptiert und dessen Ausgabe `.ecr` von `iml2cg` verarbeitet werden kann.

Danach wird der Pfad von `iml2cg` nach `.rfg` ermittelt. Auf diesem Pfad liegen die Werkzeuge `iml2cfg` und `iml2rfg`

Der gefundene Pfad sieht nun folgendermaßen aus:
`.iml` → `ecr_tool` → `.ecr` → `iml2cg` → `.cg` → `iml2cfg` → `.cfg` → `iml2rfg` → `.rfg`.

Es werden also Zwischenergebnisse vom Typ `.ecr`, `.cg` und `.cfg` erzeugt.

Nun werden folgende Werkzeugaufrufe getätigt:

```
ecr_tool -iml .BAD/temp.ecr bsp.iml
iml2cg .BAD/temp.ecr .BAD/temp.cg
iml2cfg .BAD/temp.cg .BAD/temp.cfg
iml2rfg .BAD/temp.cfg bsp.rfg
```

4.1.3. Ordnerstruktur

Das Analyse-Werkzeug erwartet eine Bestimmte Ordnerstruktur. Die Ordnerstruktur sieht so aus:

- .BAD - In diesem Ordner werden Zwischenergebnisse temporär gespeichert
- BADlog - Hier liegen erstellte Log Dateien. Der Name einer Log Datei besteht aus dem Zeitstempel des Starts der Ausführung gefolgt von dem Namen der Eingabedatei. Die von einem Werkzeug erzeugten Ausgaben werden während der Ausführung in eine solche Datei geschrieben, zusammen mit Infomeldungen über den Aufruf der einzelnen Werkzeuge
- resources - Hier ist die Basisdatei gespeichert, sowie eventuell Konfigurationen, die von Werkzeugen benötigt werden.

> BAD.jar - Die ausführbare Jar Datei, als die das Werkzeug bereitgestellt wird

> LastExecutionOrder.txt - In dieser Textdatei wird die letzte Ausführung gespeichert

Die Ordner `.BAD` und `BADlog` werden bei der ersten Ausführung automatisch erzeugt. Der Ordner `resources` muss für eine erfolgreiche Ausführung vorhanden sein, da er die Basiskonfiguration erhält. Ist diese Konfiguration oder der Ordner bei der ersten Ausführung nicht vorhanden, so wird der Ordner angelegt und eine leere `Settings.json` Datei an der Stelle erstellt, an der sie von dem Steuerungs-Werkzeug erwartet wird. Die Datei `LastExecutionOrder.txt` wird bei jeder Ausführung, mit Ausnahme von Wiederholungen, erzeugt.

4.2. Designentscheidungen

In diesem Abschnitt werden sämtliche Designentscheidungen, welche ich bei der Entwicklung des Steuerungs-Werkzeugs getroffen habe beschrieben und begründet.

4.2.1. Factory Pattern

Alle Klassen sind entweder Singeltons, oder ihre Instanzen werden von einer Factory-Klasse erstellt. Die erstellten Instanzen werden mit einem eindeutigen Schlüssel in einer `Map` gespeichert und falls diese Instanz ein zweites Mal benötigt wird, so wird sie erneut zurückgegeben und keine neue Instanz der Klasse erstellt. So wird keine Instanz doppelt erstellt.

Dies hat vor allem im Bezug auf Objekte im Abhängigkeitsgraphen einen großen Vorteil: Die Dateitypen und Werkzeuge müssen nicht im Graphen gesucht werden, sondern die entsprechende Instanz kann von der Factory-Klasse geholt werden. Die zurückgegebene Instanz ist die gleiche, welche auch im Graphen selbst vorhanden ist, lediglich eine andere Referenz darauf, sodass sämtliche Beziehungen im Graphen zur Verfügung stehen. Dies ist vor allem im Hinblick auf die Laufzeit ein großer Vorteil.

4.2.2. Konfiguration

Als Format für das Erstellen der Konfigurationsdateien habe ich mich für JSON entschieden. JSON hat den Vorteil, dass sich sehr einfach key-value Paare mit verschiedenen Typen hinterlegen lassen. So kann einem Key ein unter anderem String oder auch ein Array ohne feste Größe zugewiesen werden, was vor allem die Konfiguration einer variablen Anzahl an beispielsweise Eingabedateien oder Parametern wesentlich erleichtert. Außerdem gibt es für JSON bereits APIs für Java, um die Konfigurationsdateien zu parsen. Dateien im JSON Format enthalten auch keine header oder ähnliches, sodass die Konfigurationsdateien sehr kompakt und übersichtlich bleiben und sich auch Problemlos selbst erstellen lassen.

Hier noch ein paar Beispielhafte Konfigurationsdateien, sowie ein Beispiel für die Basiskonfiguration:

```
{
"name":"iml2rfg",
"source":[".iml",".cfg"],
"target":[".rfg"],
"params":["&i","&o"]
}
```

Listing 4.1: Konfiguration für iml2rfg

Die geschweiften Klammern in der ersten und letzten Zeile sind bei einem `JSON` Objekt notwendig. Der Name des Werkzeugs ist `iml2rfg`, dies sieht zeigt der Wert hinter dem : in der zweiten Zeile. In der dritten Zeile sieht man die Listendarstellung für mögliche Eingaben. Dieses Werkzeug kann Dateien vom Typ `.iml` und `.cfg` verarbeiten. Das Ergebnis der Analyse ist dann vom Typ `.rfg` wie man in der vierten Zeile sehen kann. Die Parameter für den Aufruf des Werkzeuges stehen in der fünften Zeile, wieder als Listendarstellung. Dieses Werkzeug bekommt lediglich den Namen, beziehungsweise den Pfad der Ein- und der Ausgabedatei übergeben, wie man an den Platzhaltern sehen kann. `&i` steht hier für eine Eingabedatei, `&o` für eine Ausgabedatei.

```
{
"name":"imldump",
"source":[".iml"],
"target":[".html"],
"params":["-html","&i",">>","&o"],
"isToolForDisplay" : true
}
```

Listing 4.2: Konfiguration für imldump

Hier sieht man eine Beispielhafte Konfiguration für das Werkzeug `imldump`. Wie beim vorherigen Beispiel sieht man hier den Namen, die möglichen Eingaben, sowie die Ausgaben. Wie oben werden auch hier die Platzhalter `&i` und `&o` für die Ein- beziehungsweise Ausgabedatei verwendet. Der Parameter `-html` gibt an, dass hier eine `.html`-Datei generiert werden soll. Das Werkzeug gibt den `html`-Code normalerweise einfach aus, da hier aber eine Ausgabedatei gewünscht ist, muss diese Ausgabe in eine Datei umgeleitet werden. Dies geschieht durch die Angabe von ».

Ich habe » als Parameter zur Umleitung der Ausgabe gewählt, weil dies einem Aufruf in einer Konsole entspricht. Ähnlich wie bei einem Konsolen-Aufruf muss der Platzhalter für die Datei, in welche die Ausgabe umgeleitet werden soll direkt danach angegeben werden.

Die Basiskonfiguration kann zum Beispiel so aussehen:

```
{
"toolPaths":["resources/tools"],
"roots":[".c"]
}
```

Listing 4.3: Basiskonfiguration

Die umgebenden Klammern sind auch hier `JSON` Syntax. In der ersten Zeile stehen die angegebenen Pfade für Werkzeugkonfigurationen. Hier ist nur ein Pfad angegeben, an dem sich alle weiteren Werkzeuge befinden. Als Quelle für den Abhängigkeitsgraphen ist hier `.c` angegeben, wie man in der dritten Zeile sehen kann. Beide Werte sind als Liste mit nur einem Element angegeben.

Die komplette Konfiguration, die ich im Rahmen dieser Bachelorarbeit erstellt habe befindet sich im Anhang.

4.2.3. Analyse

Eine Analyse ist im Kontext dieses Steuerungs-Werkzeuges die Generierung einer definierten Datei unter Verwendung einer variablen Anzahl sequentiell ausgeführter Werkzeuge, mit optionaler, anschließender Ergebnisanzeige, beziehungsweise -aufbereitung.

4.2.4. Wiederholung

Wie bereits im Abschnitt 3.3 beschrieben, wird für jede Ausführung eine Textdatei mit den genannten Werten gespeichert.

Eine solche Datei kann beispielsweise so aussehen:

```
LastCall: -s bsp.iml -t bsp.rfg --using-tools cobra --predicate fast
Used File Types: .iml, .ecr, .cg, .cfg, .rfg
ecr_tool -iml &o &i
iml2cg &i &o
iml2cfg &i &o
iml2rfg &i &o
cobra &i
```

Listing 4.4: Gespeicherte Ausführung

In der ersten Zeile, nach `LastCall:` stehen Parameter, die bei der letzten Ausführung verwendet wurden. Hier wurde `bsp.iml` als Eingabe- und `bsp.rfg` als Zieldatei angegeben. Das Werkzeug `cobra` wurde explizit angegeben und das Prädikat `fast` gewählt. In der zweiten Zeile stehen nach `Used File Types:` alle verwendeten Dateiendungen. In allen folgenden Zeilen steht jeweils ein Werkzeug Aufruf, wobei hier die Platzhalter `&i` und `&o` für Ein- beziehungsweise Ausgabedateien stehen.

Das Prädikat `fast` markiert hier die beiden Werkzeuge `ecr_tool` und `iml2cg`. Die Werkzeuge `iml2cfg` sowie `iml2rfg` sind nicht markiert, müssen aber ausgeführt werden, um eine `.rfg`-Datei aus einer `.cg`-Datei zu erzeugen.

Es ist nicht erkennbar, welche Datei als Eingabe für `cobra` verwendet wurde. Auch bei einer Wiederholung wird neu berechnet, welche Datei sich als Eingabe eignet.

Ich habe mich für eine einfach strukturierte Textdatei entschieden, um dem Anwender die Möglichkeit zu bieten, die Wiederholung zu beeinflussen. Dies kann in Form einer Veränderten Parametrisierung, oder auch einer Anpassung der verwendeten Werkzeuge geschehen. Um die Parametrisierung anzupassen genügt es, die gewünschte Veränderung in der Zeile für das entsprechende Werkzeug vorzunehmen. Möchte man jedoch die verwendeten Werkzeuge anpassen, so muss man darauf achten, dass die Zeile mit den erzeugten Dateitypen zu den verwendeten Werkzeugen passt, da diese beiden Komponenten vor der Wiederholung verglichen und dadurch validiert werden.

Beim Einlesen der Datei wird aus der ersten Zeile die Eingabe- und die Zieldatei herausgelesen. Danach werden zwei Listen gespeichert. Eine enthält die in der zweiten Zeile der Textdatei gespeicherten Dateiendungen, die andere enthält sämtliche Werkzeugaufrufe. Für die Ausführung werden die gespeicherten Werkzeuge mit entsprechender Parametrisierung von oben nach unten Ausgeführt. Dabei wird bei der Ausführung, dem Werkzeug an stelle i in der Liste, eine Datei mit der Endung an Stelle i als Eingabe- und eine Datei mit der Endung an Stelle $i + 1$ als Zieldatei zugeordnet. Nun wird die Konfiguration für dieses Werkzeug überprüft, ob die Endung an Stelle i in den Konfigurierten Eingaben und die Endung an Stelle $i + 1$ in den konfigurierten Ausgaben enthalten ist. Handelt es sich bei einer Dateiendung um die Ein- oder Ausgabedatei des Aufrufs, so wird die entsprechende Datei, ansonsten der Dateiname für eine Temporäre Datei verwendet.

4.2.5. Prädikate

Für die Umsetzung des geforderten Präferenzmechanismus habe ich mich für die Verwendung von Prädikaten entschieden, da diese einfach zu konfigurieren und bei der Ausführung leicht zu benutzten sind. Da der Name eines Prädikats frei gewählt werden kann, ist es auch möglich hier kurze Beschreibungen des markierten Pfades anzugeben, sodass ein Anwender aus dem Namen eines Prädikats gegebenenfalls auch Eigenschaften des Pfades ablesen kann und dadurch weiß, welches Prädikat er für seine Zwecke nutzen kann.

Sind an einer Stelle im Graphen viele verschiedene Pfade möglich, so kann es zu einer hohen Anzahl an Prädikaten kommen, sodass dies für einen Anwender sehr unübersichtlich wird. Um dem vorzubeugen, habe ich die Möglichkeit eingebaut Werkzeuge explizit anzugeben, sodass ein bestimmter Pfad gewählt werden kann, ohne dafür ein Prädikat konfigurieren zu müssen.

4.2.6. Werkzeuge zur Anzeige

Werkzeuge, deren Funktion es ist Informationen für den Anwender aufzubereiten und entweder selbst Anzuzeigen, oder zu speichern, müssen in der Ausführung speziell behandelt werden. Zwar könnten diese Werkzeuge genauso behandelt werden wie andere, es ist jedoch, im Hinblick auf ihre Funktionalität sinnvoller, diese gesondert zu betrachten. Es muss bei der

Ausführung explizit angegeben werden, welches Werkzeug zur Anzeige genutzt wird, um diese Sonderstellung zu ermöglichen.

Ist ein solches Werkzeug angegeben, so wird zunächst die Analyse durchgeführt, als ob keines angegeben wurde. Ist diese abgeschlossen, so wird die Eingabe für das gewählte Anzeigewerkzeug aus allen erzeugten Ausgaben gewählt, indem der Ausgeführte Pfad zurückverfolgt wird und für jede erzeugte Datei geprüft wird, ob diese eine gültige Eingabe für das Anzeigewerkzeug ist.

Es ist derzeit nur möglich ein einzelnes Anzeigewerkzeug pro Aufruf zu verwenden. Die Verwendung von zwei solchen Werkzeugen würde bei Werkzeugen mit graphischer Oberfläche zu Komplikationen führen, da das zweite, auf Grund der sequentiellen Ausführung, erst dann gestartet wird, wenn das erste geschlossen wird. Dies würde nur für Verwirrung sorgen, sodass ich die Anzahl der verwendeten Anzeigewerkzeuge auf eines begrenzt habe. In einer weiterführenden Entwicklung könnte diese Begrenzung entfernt werden, indem für jedes gefundene Anzeigewerkzeug ein eigener Thread gestartet wird.

Die explizite Angabe ist auf Grund der Definition im Abschnitt 4.2.3 notwendig. Eine Möglichkeit die explizite Angabe zu umgehen wäre, das Verständnis einer Analyse zu variieren, sodass sowohl ein Angegebenes Werkzeug, als auch eine Zieldatei als gültiges Ziel der Analyse gewährtet wird. Da die Anzeigewerkzeuge jedoch meist mehrere Dateitypen als Eingabe akzeptieren, müsste der gewünschte Typ in diesem Fall meist explizit angegeben werden, sodass hierdurch kein Vorteil entsteht. Des Weiteren könnten so Werkzeuge angegeben werden, die abhängig von der Eingabe eine andere Ausgabe erzeugen, wodurch das Ergebnis der Analyse nicht eindeutig genannt werden kann.

Wiederholung mit Anzeigewerkzeugen Nicht nur bei der ersten Ausführung, sondern auch bei jeder Wiederholung muss ein Anzeigewerkzeug besonders berücksichtigt werden. Ist in der vorherigen Ausführung ein solches Werkzeug ausgeführt worden, sieht man das daran, dass die beiden im Abschnitt 3.3 erwähnten Listen für Dateiendungen und Werkzeugaufrufe die gleiche Größe haben. Wurde kein Anzeigewerkzeug ausgeführt, so enthält die Liste mit Werkzeugaufrufen ein Element weniger, als die Liste mit Dateiendungen. Dies liegt daran, dass für jedes Standard Werkzeug zwei Dateiendungen, nämlich Ein- und Ausgabe, existieren, die Ausgabe des einen Werkzeugs jedoch gleichzeitig die Eingabe des anderen darstellt. So existiert pro Werkzeug eine Eingabedatei, plus eine nicht wiederverwendete Ausgabedatei für das letzte Werkzeug. Ist das letzte Werkzeug der Liste zur Anzeige gedacht, so wird keine Ausgabe erstellt, wodurch beide Listen die gleiche Länge haben.

Wurde festgestellt, dass ein Anzeigewerkzeug enthalten ist, wird das letzte Element der Liste gesichert und anschließend der Teil der Liste ermittelt, der wiederholt werden soll. Im Anschluss daran muss der Anwender angeben, ob er das zuvor ausgeführte Anzeigewerkzeug ebenfalls erneut Ausführen will. Dabei wird hier aus dem Wiederholten teil das am weitesten unten stehende Ergebnis, welches von dem Anzeigewerkzeug ermittelt werden kann, als Eingabe gewählt.

4.2.7. Konfigurationswerkzeuge

Konfigurationen werden vor der eigentlichen Analyse erstellt. Obwohl hier eine Parallelisierung möglich wäre, habe ich diese jedoch aus Zeitmangel nicht implementiert, da der Synchronisationsaufwand zu hoch war.

Die eigentliche Analyse könnte so lange parallel zu der Erzeugung von Konfigurationen ausgeführt werden, bis ein Werkzeug ausgeführt werden soll, welches eine der Konfigurationen benötigt, die Ausführung müsste in diesem Fall pausiert werden, bis die Konfiguration vorliegt.

Bei einer Wiederholung werden die entsprechenden Konfigurationen nicht neu erzeugt, sondern wiederverwendet.

4.2.8. Anwenderspezifischer Quellcode

Wie bereits im Abschnitt 3.1 erwähnt, kann für jedes Werkzeug eine `JAVA`-Klasse angegeben werden, die Code enthält, welcher im Anschluss an die Ausführung des Werkzeuges ausgeführt wird.

Die entsprechende Klasse muss das Interface *IAfterExecution* implementieren, welches im Paket *de.uni.stuttgart.bauhaus.toolexecution* liegt. Dieses Interface enthält eine Methode `execute` welche von dem Steuerungswerkzeug aufgerufen wird.

Die Klasse muss als `.class`-Datei vorliegen. Da ein Interface aus dem Steuerungs-Werkzeug benötigt wird, muss beim kompilieren der Klasse die `jar`-Datei, als welche das Steuerungs-Werkzeug vorliegt, als Bibliothek angegeben werden.

Am Beispiel des Werkzeuges `symta`, welches `Concurrency.config` und `Non-Concurrency.config` Dateien erstellt, erkennt man den Grund für diese Funktion.

Das Werkzeug `symta` erhält eine Datei vom typ `.func_names` und eine `.csv`-Datei, woraus sie die beiden Zieldateien erstellt. Für diese Dateien kann jedoch kein Zielpfad gesetzt werden und es kann nicht angegeben werden, ob nur eine der Dateien erstellt werden soll. Beide Dateien werden immer im Verzeichnis, in dem `symta` ausgeführt wird erzeugt.

Durch Anwenderspezifischen Code kann nun beispielsweise die Datei `Concurrency.config` in den Ordner `.bauhaus` im `home` Verzeichnis des Nutzers kopiert werden und dort sinnvoll verwendet werden. Zusätzlich kann die Datei `Non-Concurrency.config` gelöscht werden, falls diese nicht benötigt wird.

Ohne diese Funktionalität wäre es in solchen Fällen nicht möglich, eine automatisierte Analyse durchzuführen.

4.3. Bedienung des Werkzeugs

Ich habe das Steuerungs-Werkzeug als Konsolen-Anwendung entwickelt.

Beim Aufruf des Werkzeugs können verschiedene Parameter mitgegeben werden. Wird kein Parameter mitgegeben, so wird eine Liste aller möglichen Parameter ausgegeben.

Diese Parameter können dem Werkzeug mitgegeben werden:

- `-h`, `--help` oder `--usage`
- `-s` oder \verbsource|
- `-t` oder \verbtarget|
- `-r` oder \verbrepeat|
- `-ra` oder \verbrepeat-all|
- `-f` oder \verbforce-new|
- `-p` oder \verbpredicate|
- `--dot`
- `--using-tools`
- `--show-predicates`

Für einige Parameter gibt es mehrere verschiedene Varianten. Diese haben alle dieselbe Funktionalität und dienen nur zum besseren Verständnis beziehungsweise zur besseren Übersicht. Im folgenden werde ich nur die Kurzform der Parameter benutzen.

Die Parameter `-h`, `-r`, `-ra` und `--show-predicates` können nur einzeln benutzt werden, während alle anderen Parameter beliebig kombiniert werden können, wobei nicht jede Kombination zu einem Sinnvollen Ergebnis führt.

Der Parameter `-h` dient zur Ausgabe aller möglichen Parameter.

Die Parameter `-r` und `-ra` dienen zur Wiederholung der letzten Analyse, wobei der Parameter `-ra` eine vollständige Wiederholung startet, während der Anwender bei `-r` den Abschnitt, welcher wiederholt werden soll angeben kann.

Bei `--show-predicates` werden alle konfigurierten Prädikate ausgegeben.

Gibt man den Parameter `--dot` gefolgt von einem Dateinamen, der die Endung `.dot` haben sollte, an, so wird der konfigurierte Abhängigkeitsgraph im `.dot`-Format in die angegebene Datei gespeichert.

`-s` und `-t` führen nur wenn beide Angegeben sind zu einer Sinnvollen Ausführung. `-s` gefolgt von einem Dateinamen beziehungsweise einem Pfad zu einer Datei, gibt die Eingabedatei für die Analyse an. `-t` gefolgt von einem Dateinamen gibt die Zieldatei für die Analyse an.

Die Parameter `-p`, `--using-tools` und `-f` führen nur dann zu einer Sinnvollen Ausführung, wenn die Parameter `-s` und `-t` beide gesetzt sind. Auf `-p` folgt das Prädikat, welches bei der Analyse verwendet werden soll und auf `--using-tools` folgt eine Liste an Werkzeugen, welche bei der Analyse ausgeführt werden sollen.

Der Parameter `-f` gibt an, dass keine Wiederholung stattfinden soll, stattdessen soll der Pfad neu berechnet werden. Dies ist sinnvoll, falls der Anwender etwas an der Konfiguration geändert hat oder falls die Datei `LastExecutionOrder.txt` verändert wurde, diese Änderung aber nicht berücksichtigt werden soll. Diese Datei zu löschen oder die erste Zeile nach *LastCall:* zu verändern, sodass diese nicht mehr zu den eingegebenen Parametern passt, hätte den gleichen Effekt.

4.4. Tests

Die Validität dieses Ansatzes und der Umsetzung soll an mindestens zwanzig Bauhaus Werkzeugen gezeigt werden.

Ich habe folgende Werkzeuge für die Tests konfiguriert:

1. anderson_tool
2. ccdiml
3. cobra
4. das_tool
5. dump_cg
6. ecr_tool
7. function_names
8. iml2cfg
9. iml2cg
10. iml2dot
11. iml2html
12. iml2rfg
13. imldump
14. imlstat
15. kcg_stats
16. partition_tool
17. pta_tool
18. raceq
19. rfg2gxl
20. stats
21. symta
22. thread_tool

Name	Größe
mksyntax.iml	62kiB
mkeyes.iml	117,1 kiB
aget.iml	280,4 kiB
uuname.iml	787,4 kiB
nano.iml	1.8 miB
cu.iml	3,1 miB

Tabelle 4.1.: Verwendete Dateien für die Tests

Die folgenden Testfälle habe ich mit diesen Werkzeugen abgedeckt:

- Positivtests:
  - Suche nach dem kürzesten Pfad zwischen Ein- und Ausgabedatei
  - Suche nach dem kürzesten Pfad mit anschließender Anzeige
  - Pfadsuche mit Prädikat
  - Pfadsuche mit Prädikat und anschließender Anzeige
  - Angabe eines expliziten Werkzeuges zwischen Ein- und Ausgabedatei
  - Explizite Angabe eines Werkzeuges, welches nach der Ausgabedatei ausgeführt werden soll
  - Ausführung einer Analyse mit Konfigurationswerkzeug
  - Kombination aus expliziter Werkzeugangabe und Prädikat
  - Wiederholung
  - Wiederholung mit Anzeige
- Negativtests:
  - Unmöglicher Pfad (Ein- und Ausgabedatei sind nicht durch einen Pfad im Abhängigkeitsgraphen verbunden)
  - Korrekter Pfad plus expliziter Angabe eines so nicht Ausführbaren Werkzeuges
  - Eingabe einer nicht unterstützten Datei
  - Verwendung eines nicht existenten Prädikats
  - Angabe eines nicht nutzbaren Prädikats

Eingabetyp	Zieldateityp	Ausgeführte Werkzeuge
.iml	.rfg	iml2rfg
.iml	.tt	pta_tool, iml2cfg, thread_tool
.iml	.html	iml2html
.iml	.gxl	iml2rfg, rfg2gxl

Tabelle 4.2.: Tests für kürzesten Pfad

Eingabetyp	Zieldateityp	Typ der angezeigten Datei	Ausgeführte Werkzeuge
.iml	.rfg	.rfg	iml2rfg
.iml	.tt	.race	pta_tool, iml2cfg, thread_tool, raceq
.iml	.html	.iml	iml2html
.iml	.gxl	.rfg	iml2rfg, rfg2gxl

Tabelle 4.3.: Tests für kürzesten Pfad mit Anzeige

Die für die Tests benutzten Dateien stehen in Tabelle 4.1

Die Dateien sind alle vom Typ .iml es gehen also sämtliche Ausführungspfade von diesem Dateityp aus. Die unten aufgeführten Tests wurden jeder mit allen Eingabedateien durchgeführt.

Positivtests Sinn der Positivtests ist, die Funktionalität des Steuerungs-Werkzeuges bei korrekter Anwendung zu prüfen.

Kürzester Pfad Tests und Ergebnisse stehen in Tabelle 4.2.

Alle Werkzeuge wurden Ausgeführt und sämtliche Ergebnisse und Zwischenergebnisse erzeugt und gespeichert.

Einfacher Pfad plus Anzeige Als Anzeigewerkzeug wurde hier cobra verwendet. Tests und Ergebnisse stehen in Tabelle 4.3.

Alle Werkzeuge wurden Ausgeführt und sämtliche Ergebnisse und Zwischenergebnisse erzeugt und gespeichert. Das Anzeigewerkzeug wurde stets mit der angegebenen Datei ausgeführt, beziehungsweise gar nicht für die Erzeugung von .html-Dateien.

Eingabetyp	Zieldateityp	Verwendetes Prädikat	Ausgeführte Werkzeuge
.iml	.rfg	fast	ecr_tool, iml2cg, iml2cfg, iml2rfg
.iml	.rfg	exact	anderson_tool, iml2cg, iml2cfg, iml2rfg
.iml	.race	execute_red	pta_tool, iml2cfg, red, thread_tool, red, raceq
.iml	.html	dump	imldump
.iml	.gxl	fast	ecr_tool, iml2cg, iml2cfg, iml2rfg, rfg2gxl
.iml	.gxl	exact	anderson_tool, iml2cg, iml2cfg, iml2rfg, rfg2gxl

Tabelle 4.4.: Tests für Pfad mit Prädikat

Pfad mit Prädikat Tests und Ergebnisse stehen in Tabelle 4.4.

Alle Werkzeuge wurden Ausgeführt und sämtliche Ergebnisse und Zwischenergebnisse erzeugt und gespeichert, abgesehen von der Zieldatei vom Typ `.race`. Bei dieser wurden zwar alle Werkzeuge ausgeführt, die Zieldatei jedoch nicht erzeugt. Beim Versuch das Werkzeug `raceq` unabhängig von dem Steuerungs-Werkzeug auszuführen wurde jedoch ebenfalls keine Ausgabedatei erzeugt und auch keine Fehlermeldung angezeigt. Die Ausgeführten Werkzeuge, die nicht auf dem kürzesten Weg liegen, waren alle mit dem entsprechenden Prädikat versehen oder notwendig um Lücken zwischen Werkzeugen zu schließen.

Pfad mit Prädikat und Anzeige Tests und Ergebnisse stehen in Tabelle 4.5. Es wurden zwei Durchläufe durchgeführt, einmal mit `cobra` und einmal mit `imldump` als Anzeigewerkzeug.

Auch diese Tests verliefen erfolgreich. Sämtliche Werkzeuge wurden ausgeführt und die entsprechenden Dateien wurden angezeigt. Beim zweiten Durchlauf wurde für jede Zieldatei die Datei `temp.html` im Ordner `.BAD` erstellt.

Explizite Werkzeugangabe Hier werden die Tests für Werkzeuge zwischen Ein- und Ausgabedatei, sowie solche für Werkzeuge nach er Ausgabedatei beschrieben. Tests und Ergebnisse stehen in Tabelle 4.6. Die Tabelle enthält zwei Testfälle, bei denen der Dateityp von Ein- und Ausgabedatei der gleiche ist, der ausgeführte Pfad also nur durch explizite Werkzeugangabe

Eingabetyp	Zieldateityp	Verwendetes Prädikat	Angezeigter Dateityp	Ausgeführte Werkzeuge
.iml	.rfg	fast	.rfg	ecr_tool, iml2cg, iml2cfg, iml2rfg
.iml	.rfg	exact	.rfg	anderson_tool, iml2cg, iml2cfg, iml2rfg
.iml	.race	execute_red	.rett	pta_tool, iml2cfg, red, thread_tool, red, raceq
.iml	.html	dump	.iml	imldump
.iml	.gxl	fast	.rfg	ecr_tool, iml2cg, iml2cfg, iml2rfg, rfg2gxl
.iml	.gxl	exact	.rfg	anderson_tool, iml2cg, iml2cfg, iml2rfg, rfg2gxl

Tabelle 4.5.: Tests für Pfad mit Prädikat mit Anzeige

beschrieben ist. Dies ist keine falsche Eingabe, sondern soll ermöglichen, ein bestimmtes Werkzeug auch ohne Kenntnis des konfigurierten Ausgabeformats auszuführen, da der gefundene Pfad vollständig durch die Eingabedatei und die Angegebenen Werkzeuge definiert ist.

Analyse mit Konfigurationswerkzeug Unter den mir zur Verfügung gestellten Werkzeugen war lediglich `symta` als Konfigurationswerkzeug. Deshalb habe ich hierfür nur zwei Pfade getestet: Von `.iml` nach `.tt` als einfacher Aufruf und von `.iml` nach `.stat` mit Prädikat `execute_red`.

Im ersten Fall wurden zunächst folgende Werkzeuge ausgeführt: `function_names` und `symta`, wodurch eine `Concurrency.config` erstellt und anschließend nach `.bauhaus` kopiert wurde. Danach wurden die Werkzeuge `pte_tool`, `iml2cfg` und `thread_tool` ausgeführt, wobei die erstellte `Concurrency.config` genutzt wurde.

Beim zweiten Fall wurden ebenfalls die Werkzeuge `function_names` und `symta` ausgeführt und die erstellte `Concurrency.config` nach `.bauhaus` kopiert. Hier wurden daraufhin folgende Werkzeuge ausgeführt: `pta_tool`, `iml2cfg`, `red`, `thread_tool`, `red` und `stats`.

Die Tests verliefen erfolgreich und die Konfigurationsdateien sowie die Ergebnisse wurden erstellt.

Eingabetyp	Zieldateityp	Explizit angegebene Werkzeuge	Ausgeführte Werkzeuge
.iml	.rfg	das_tool	das_tool, iml2cfg, iml2rfg
.iml	.rfg	iml2cg	iml2cg, iml2cfg, iml2rfg
.iml	.tt	ecr_tool, red	ecr_tool, iml2cg, pta_tool, iml2cfg, red, thread_tool
.iml	.cg_dump	ecr_tool	ecr_tool, iml2cg, dump_cg
.iml	.gxl	iml2cfg	iml2cfg, iml2rfg, rfg2gxl
.iml	.iml	imlstat	imlstat
.iml	.cfg	iml2cg, rfg2gxl	iml2cg, iml2cfg, iml2rfg, rfg2gxl
.iml	.iml	stats	pta_tool, iml2cfg, thread_tool, stats

Tabelle 4.6.: Tests für Pfad mit expliziter Werkzeugangabe

Kombination aus expliziter Werkzeugangabe und Prädikat Die Ergebnisse für diese Tests stehen in der Tabelle 4.7.

Wiederholung Um die Wiederholungsfunktion zu testen habe ich zwei Pfade ausgeführt und diese einmal vollständig und drei mal teilweise, die erste Hälfte, die zweite Hälfte und einen Abschnitt aus der Mitte des Pfades, wiederholt.

Die beiden Pfade waren:

- pta_tool, iml2cfg, red, thread_tool und red
- erc_tool, iml2cg, iml2cfg, iml2rfg und rfg2gxl

Die Teilweisen Wiederholungen waren für die erste Analyse:

- pta_tool, iml2cfg, red, thread_tool und red
- pta_tool, iml2cfg und red
- red, thread_tool und red
- iml2cfg, red und thread_tool

Für die zweite Analyse wurden folgende Pfade ausgeführt:

Eingabetyp	Zieldateityp	Verwendetes Prädikat	Angegebene Werkzeuge	Ausgeführte Werkzeuge
.iml	.rfg	fast	rfg2gxl	ecr_tool, iml2cg, iml2cfg, iml2rfg, rfg2gxl
.iml	.cfg	exact	iml2dot	anderson_tool, iml2cg, iml2cfg, iml2dot
.iml	.tt	execute_red	stats	pta_tool, iml2cfg, red, thread_tool, red, stats
.iml	.tt	execute_red	ccdiml	pta_tool, iml2cfg, red, thread_tool, ccdiml

Tabelle 4.7.: Tests für Pfad mit Prädikat und Expliziter Werkzeugangabe

- erc_tool, iml2cg, iml2cfg, iml2rfg und rfg2gxl
- erc_tool, iml2cg und iml2cfg
- iml2cfg, iml2rfg und rfg2gxl
- iml2cg, iml2cfg und iml2rfg

Sämtliche Pfade wurden vollständig ausgeführt und die entsprechenden Ergebnisse neu erzeugt.

Wiederholung mit Anzeige Für die Tests der Wiederholung mit Anzeige wurden die gleichen Analysen, die im Abschnitt 4.4 beschrieben wurden. Als Anzeigewerkzeug wurde cobra benutzt. Jeder der wiederholten Abschnitte wurde einmal mit und einmal ohne Wiederholung des Anzeigewerkzeugs ausgeführt.

Die folgenden Dateitypen wurden für die getesteten Pfade ausgeführt:

- pta_tool, iml2cfg, red, thread_tool und red → .rett
- pta_tool, iml2cfg und red → .re
- red, thread_tool und red → .rett

- `iml2cfg`, `red` und `thread_tool` → `.tt`
- `erc_tool`, `iml2cg`, `iml2cfg`, `iml2rfg` und `rfg2gxl` → `.rfg`
- `erc_tool`, `iml2cg` und `iml2cfg` → `.cfg`
- `iml2cfg`, `iml2rfg` und `rfg2gxl` → `.rfg`
- `iml2cg`, `iml2cfg` und `iml2rfg` → `.rfg`

Negativtests Ich habe die folgenden Negativtests durchgeführt, um die Robustheit des Steuerungs-Werkzeugs zu prüfen.

Nicht existenter Pfad Ich habe folgende Pfade getestet: `.rfg` → `.tt` und `.ecr` → `.race`. In beiden Fällen wurde keine Analyse ausgeführt und die Meldung *Could not compute execution order* ausgegeben.

Nicht existentes Werkzeug Bei diesem Test habe ich zu den beiden möglichen Pfaden `.iml` → `.tt` und `.iml` → `.stat` jeweils *no_real_tool* als Werkzeug explizit angegeben. Das falsche Werkzeug wurde bei sämtlichen Ausführungen ignoriert und stattdessen der kürzeste Pfad berechnet.

Zusätzlich habe ich die gleichen Pfade mit zusätzlicher expliziter Angabe von `cobra` durchgeführt. Auch hier wurde das falsche Werkzeug ignoriert und stattdessen der kürzeste Pfad gefolgt von `cobra` Ausgeführt.

Nicht unterstützte Datei Hierfür habe ich eine Datei mit der Endung `.nothing`, welche nicht in der Konfiguration existiert, einmal als Eingabe- und einmal als Zieldatei angegeben.

Es wurde keine Analyse ausgeführt und die Meldung *Could not compute execution order* ausgegeben

Nicht existierendes Prädikat Wurde ein nicht existentes Prädikat bei der Ausführung angegeben, so wurde der kürzeste Weg zwischen Eingabe- und Zieldatei, beziehungsweise der Weg über explizit angegebene Werkzeuge ausgeführt.

Für angegebenen Pfad nicht nutzbares Prädikat Das Verhalten in diesem Fall war gleich, wie das Verhalten bei einem nicht existierenden Prädikat.

4.5. Laufzeit

Aus Zeitmangel konnte ich keine ausführliche Messung der Laufzeit durchführen. Ich habe jedoch, um das Verhältnis zwischen Berechnung des Analysepfades, also vom Start des Werkzeuges bis zum Beginn der ersten Analyse, und Ausführung der gefundenen Werkzeuge, also die Zeit vom Aufruf des ersten Werkzeuges bis zum Beenden des letzten Werkzeugs, abschätzen zu können, einige Ausführungen vermessen. Unter den gemessenen Ausführungen waren unter anderem Eingaben von nicht existenten Werkzeugen, da in diesen Fällen der gesamte Abhängigkeitsgraph durchsucht werden muss.

Das Testsystem, auf welchem ich diese Messung durchgeführt habe, ist der Rechner *pslx0* vom Institut `ISTE` an der Universität Stuttgart. Die Verbaute CPU ist ein AMD Opteron 6174. Die Größe des Arbeitsspeichers, der mir zur Verfügung stand war 252 Gigabyte. Das Folgende Betriebssystem war hier installiert: `Debian 3.16.7-ckt25-2+deb8u3`.

Für die Messungen habe ich die Eingabedateien `mkeyes.iml` und `uuname.iml` verwendet.

Für die Analysen mit `mkeyes.iml` lag die durchschnittliche Zeit für die Pfadsuche bei $257,25$ Millisekunden. Die durchschnittliche Zeit für die Ausführung lag bei $1800,625$ Millisekunden. Die Zeit für die Ausführung ist deutlich höher.

Die Messung mit `uuname.iml` konnte ich nicht abschließen, da die Analyse dieser Datei bei einigen Werkzeugen, wie zum Beispiel `pta_tool` dazu geführt hat, dass die `SSH` Verbindung unterbrochen wurde, bevor die Analyse beendet war.

Diese Messungen zeigen, dass die Ausführungszeit des von mir entwickelten Steuerungs-Werkzeuges im Vergleich mit der Analysedauer keine Rolle spielt und vor allem bei großen Dateien vernachlässigt werden kann.

5. Zusammenfassung und Ausblick

Ziel dieser Arbeit war es, eine Steuerung für Analysen mit Bauhaus zu entwerfen und zu implementieren. Es wurden Anwendungsfälle eines Steuerungs-Werkzeugs und Bauhaus' erstellt und Eigenschaften von Bauhaus analysiert. Darunter die Abhängigkeiten zwischen den einzelnen Werkzeugen, wie aus diesen Abhängigkeiten ein Graph erstellt werden kann und die Eigenschaften dieses Graphen. Anschließend wurde erklärt, wie dieser Graph für die Automatisierung angepasst werden kann und wieso die Eigenschaft der Zyklenfreiheit im Rahmen dieser Arbeit und der automatisierten Analyse hinzugefügt wurde. Anschließend wurde eine Möglichkeit vorgestellt, wie ein solches Werkzeug gestaltet werden kann, um sämtliche Anforderungen abzudecken, sowie möglichst einfach und benutzerfreundlich zu gestalten. Die Konfigurierbarkeit des Werkzeugs, sowie der Ablauf einer automatisierten Analyse, deren Wiederholung und ein Ansatz zur Verwendung eines Präferenzmechanismus zur Markierung von Pfaden im Abhängigkeitsgraphen werden durch den in dieser Arbeit beschriebenen Ansatz abgedeckt.

Anschließend wurde beschrieben, wie dieser Ansatz realisiert wurde, welche Probleme dabei aufgetreten sind und wie diese gelöst wurden. Zuletzt wurden die Testfälle und -Ergebnisse beschrieben, durch welche gezeigt wurde, dass der Ansatz valide und praktikabel ist und sämtliche Anforderungen an die Funktionalität eines Steuerungs-Werkzeugs abgedeckt wurden.

Das Ziel der Arbeit wurde erreicht.

Ausblick

Das im Rahmen dieser Arbeit implementierte Steuerungs-Werkzeug deckt zwar alle gestellten Anforderungen ab, es gibt jedoch noch einige Punkte, die verbessert oder hinzugefügt werden können.

Beispielsweise die Unterstützung von mehreren Eingabedateien oder die Parallele Ausführung von Analysepfaden. Eine Funktion zur Prüfung, ob eine Analyse durchgeführt werden muss, oder ob Dateien aus einer älteren Analyse wiederverwendet werden können, sowie eine erweiterte Fehlerbehandlung, mit der im Fehlerfall automatisch alternative Pfade im Abhängigkeitsgraph gesucht und ausgeführt werden.

Für den Anwendungsfall des Bauhaus-Systemtests könnte eine Funktion ergänzt werden, welche diesen Test mit einem einzelnen Aufruf durchführt.

Literaturverzeichnis

[16a] *Aufgabenstellung*. 2016 (zitiert auf S. 8).

[16b] *Bauhaus Demonstration*. 2016. URL: http://www2.informatik.uni-stuttgart.de/iste/ps/bauhaus/demo/index.html (zitiert auf S. 8).

[16c] *Bauhaus website*. 2016. URL: http://www.iste.uni-stuttgart.de/ps/projekt-bauhaus.html (zitiert auf S. 8).

[16d] *JSON Api for Java*. 2016. URL: https://github.com/stleary/JSON-java (zitiert auf S. 25).

[IST16a] ISTE. „Bauhaus Dokumentation“. 2016 (zitiert auf S. 10).

[IST16b] ISTE. „Makefile zur Ausführung von Analysen“. 2016 (zitiert auf S. 7, 14).

Alle URLs wurden zuletzt am 10. 08. 2016 geprüft.

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.

Ort, Datum, Unterschrift

A. Vollständige Konfiguration

```
{
"name" : "andersen\_tool",
"source" : [".iml"],
"target" : [".anderson"],
"predicates" : ["exact"],
"params" : ["\&i", "\&o"],
"isToolForDisplay" : false
}
```

Listing A.1: Konfiguration für andersen_tool

```
{
"name" : "cafeCC",
"source" : [".c"],
"target" : [".iml"],
"predicates" : [],
"params" : ["-o", "\&o", "\&i"],
"isToolForDisplay" : false
}
```

Listing A.2: Konfiguration für cafeCC

```
{
"name" : "ccdiml",
"source" : [".iml"],
"target" : [],
"predicates" : [],
"params" : ["\&i", "\&o"],
"isToolForDisplay" : false
}
```

Listing A.3: Konfiguration für ccdiml

```
{
"name" : "cobra",
"source" : [".iml", ".cfg", ".cg", ".tt", ".kcg", ".re"],
"target" : [],
"predicates" : [],
"params" : ["\&i"],
"isToolForDisplay" : true
}
```

Listing A.4: Konfiguration für cobra

```
{
"name" : "das\_tool" ,
"source" : [".iml"],
"target" : [".das"],
"predicates" : ["precise"],
"params" : ["\&i", "\&o"],
"isToolForDisplay" : false
}
```

Listing A.5: Konfiguration für das_tool

```
{
"name" : "dra\_filter",
"source" : [".tt"],
"target" : [".dra"],
"predicates" : [],
"params" : ["\&i", "\&o"],
"isToolForDisplay" : false
}
```

Listing A.6: Konfiguration für dra_filter

```
{
"name" : "dump\_cg",
"source" : [".iml", ".cg", ".cfg"],
"target" : [".cg\_dump"],
"params" : ["-o", "\&o", "\&i"],
"isToolForDisplay" : true
}
```

Listing A.7: Konfiguration für dump_cg

```
{
"name" : "ecr\_tool",
"source" : [".iml"],
"target" : [".ecr"],
"predicates" : ["fast"],
"params" : [ "-iml", "\&o", "\&i"]

}
```

Listing A.8: Konfiguration für ecr_tool

```
{
"name" : "function\_names",
"source" : [".iml"],
"target" : [".func\_name"],
"predicates" : [],
"params" : ["\&i", ">>", "\&o"],
"isToolForDisplay" : false
}
```

Listing A.9: Konfiguration für function_names

```
{
"name" : "iml2cfg",
"source" : [".iml", ".cg", ".das", ".anderson"],
"target" : [".cfg"],
"predicates" : ["exact"],
"params" : ["\&i", "\&o"],
"isToolForDisplay" : false
}
```

Listing A.10: Konfiguration für iml2cfg

```
{
"name" : "iml2cfg",
"source" : [".pta"],
"target" : [".pcfg"],
"predicates" : [],
"params" : ["\&i", "\&o"],
"isToolForDisplay" : false
}
```

Listing A.11: Konfiguration für iml2cfg

```
{
"name" : "iml2cg",
"source" : [".iml", ".anderson", ".das", ".ecr"],
"target" : [".cg"],
"params" : ["\&i", "\&o"],
"predicates" : ["exact", "precise"],
"isToolForDisplay" : false
}
```

Listing A.12: Konfiguration für iml2cg

```
{
"name" : "iml2dot",
"source" : [".iml"],
"target" : [".dot"],
"predicates" : [],
"params" : ["\&i", "\&o"],
"isToolForDisplay" : true
}
```

Listing A.13: Konfiguration für iml2dot

```
{
"name" : "iml2html",
"source" : [".iml"],
"target" : [".html"],
"predicates" : [],
"params" : ["\&i", "\&o"],
"isToolForDisplay" : true
}
```

Listing A.14: Konfiguration für iml2html

```
{
"name" : "iml2rfg",
"source" : [".iml",".cfg"],
"target" : [".rfg"],
"predicates" : [],
"params" : ["\&i", "\&o"],
"isToolForDisplay" : false
}
```

Listing A.15: Konfiguration für iml2rfg

```
{
"name" : "imldump",
"source" : [".iml", ".cfg", ".cg", ".tt", ".kcg", ".re", "rett"],
"target" : [".html"],
"predicates" : ["dump"],
"params" : ["-html", "\&i", ">>", "\&o"],
"isToolForDisplay" : true
}
```

Listing A.16: Konfiguration für imldump

```
{
"name" : "imlstat",
"source" : [".iml"],
"target" : [".stats"],
"predicates" : [],
"params" : ["\&i", "\&o"],
"isToolForDisplay" : false
}
```

Listing A.17: Konfiguration für imlstat

```
{
"name" : "kcg\_stats",
"source" : [".pta", ".tt"],
"target" : [".kcg"],
"params" : ["-o", "\&o", "\&i"]
}
```

Listing A.18: Konfiguration für kcg_stats

```
{
"name" : "partition\_tool",
"source" : [".tt"],
"target" : [".dot"],
"predicates" : [],
"params" : ["-o", "\&o", "\&i"],
"isToolForDisplay" : true
}
```

Listing A.19: Konfiguration für partition_tool

```
{
"name" : "pta\_tool",
"source" : [".iml", ".cg"],
"target" : [".pta"],
"predicates" : [],
"params" : ["-o", "\&o", "\&i"],
"isToolForDisplay" : false
}
```

Listing A.20: Konfiguration für pta_tool

```
{
"name" : "raceq",
"source" : [".tt", ".rett"],
"target" : [".race"],
"predicates" : [],
"params" : ["-o", "\&o", "\&i"],
"isToolForDisplay" : false
}
```

Listing A.21: Konfiguration für raceq

```
{
"name" : "red",
"source" : [".tt"],
"target" : [".rett"],
"params" : ["-o", "\&o", "\&i"],
"predicates" : ["execute\_red"]
}
```

Listing A.22: Konfiguration für red

```
{
"name" : "red",
"source" : [".pcfg"],
"target" : [".re"],
"params" : ["-o", "\&o", "\&i"],
"predicates" : ["execute\_red"]
}
```

Listing A.23: Konfiguration für red

```
{
"name" : "rfg2gxl",
"source" : [".rfg"],
"target" : [".gxl"],
"predicates" : [],
"params" : ["\&i", "\&o"],
"isToolForDisplay" : false
}
```

Listing A.24: Konfiguration für rfg2gxl

```
{
"name" : "stats",
"source" : [".tt", ".rett"],
"target" : [".stat"],
"predicates" : ["exact"],
"params" : ["-o", "\&o", "\&i"],
"isToolForDisplay" : false
}
```

Listing A.25: Konfiguration für stats

```
{
"name" : "symta.sh",
"source" : [".func\_name"],
"target" : ["Concurrency.config"],
"isToolForConfig" : true,
"toolsToConfigure" : ["pta\_tool", "thread\_tool"],
"params" : ["\&i", "/home/baueras/mockup.symta.csv"],
"afterExecutionClass" : "resources/AfterSymta.class"
}
```

Listing A.26: Konfiguration für symta

```
{
"name" : "thread\_tool",
"source" : [".pcfg", ".re"],
"target" : [".tt"],
"predicates" : [],
"params" : ["-o", "\&o", "\&i"],
"isToolForDisplay" : false
}
```

Listing A.27: Konfiguration für thread_tool