Entwicklung einer Call-Graph-Transformation von Soot zu Neo4j

Dean Jonas Finkes

Kiel University
Department of Computer Science
24098 Kiel, Germany

Zusammenfassung. Call-Graphen sind spezielle, gerichtete Graphen und ein effektives Instrument für Untersuchungen des Quellcodes. Sie beschreiben die internen Beziehungen zwischen verschiedenen Methoden und bilden somit die Basis für verschiedene Programmanalysen.
In dieser Ausarbeitung liegt das Hauptaugenmerk auf der automatisierten Erstellung von Call-Graphen für Programme in Java. Die Konstruktion des Graphen wird dabei mit Hilfe des Frameworks Soot durchgeführt. Um den erstellten Graphen genauer inspizieren zu können, werden die ermittelten, temporären Daten anschließend in die Graphdatenbank Neo4j übertragen und persistent gespeichert.

1 Einleitung

Objektorientierte Software hat die Eigenschaft, dass bereits relativ simple Programme aus einer Vielzahl von Klassen und Methoden bestehen und daher schwer zu analysieren sind. Meist ist es aber gewünscht, sei es aus Gründen der Architektur, Performance oder Wartbarkeit, zu wissen, wie eine Software aufgebaut ist bzw. welche Abhängigkeiten zwischen bestimmten Programmabschnitten vorliegen. Call- und Kontrollflussgraphen können Auskunft darüber geben und sind deshalb im Bereich der Softwareanalyse ein gängiges Werkzeug.

In dieser Arbeit wird mit Hilfe des Java-Frameworks Soot und einer Neo4j-Graphdatenbank ein automatisiertes Transformationsprogramm beschrieben. Im ersten Schritt wird mittels Soot der Call-Graph für ein zu analysierendes Java-Programm berechnet. Aufbauend auf dem Graphen werden die analysierten Methoden, Klassen und Funktionen als Knoten in die Graphdatenbank übertragen und mit entsprechenden „Caller-“ bzw. „Callee-Kanten“ ergänzt. Das Ergebnis ist ein Datenbankzustand, der den Call-Graphen des analysierten Programms möglichst gut approximiert. Der entstandene Graph verfügt über zwei Abstraktionsebenen der Analyse. Auf der gröberen Ebene ist beispielsweise zu sehen, welche Funktionen einer bestimmten Klasse angehören und wie die Aufrufabhängigkeiten zwischen den Funktionen aussehen. Allerdings sind in dieser Ansicht keine Informationen über die Reihenfolge der Aufrufe von den einzelnen Funktionen zu sehen. Die zweite Ebene hingegen spiegelt die Ordnung der aufrufenden Anweisungen innerhalb einer Funktion wider. Auf diese Weise steht dem Benutzer sowohl der Call-Graph als auch ein ausgedünnter Kontrollflussgraph für

weitere Analysen zur Verfügung. Durch die zugrundeliegende Neo4j-Datenbank ist es möglich, den Graphen gezielt mit Anfragen zu untersuchen, um beispielsweise herauszufinden, welche Funktionen eine gewisse bestimmte Funktion als Nachfolger hat.

Neben der Einleitung besteht diese Arbeit aus fünf Kapiteln. Im folgenden Abschnitt 2 werden die Grundlagen zur Verständlichkeit dieser Arbeit gelegt. Es wird erläutert wie ein allgemeiner Graph aufgebaut ist, und inwieweit sich ein Call- bzw. Kontrollflussgraph von einem gewöhnlichen Graphen unterscheidet. In den anschließenden Kapiteln 3 und 4 werden sowohl das Java-Framework Soot als auch die Graphdatenbank Neo4j beschrieben. Das Kapitel 5 beinhaltet eine Erklärung der genutzten Architektur für das Transformationsprogramm und in welchen Schritten die Transformation von Java-Programm zu Call-Graph in Neo4j stattgefunden hat. Im letzten Kapitel wird neben dem abschließenden Fazit ein Ausblick gegeben, inwieweit das Transformationsprogramm noch erweitert werden kann.

2 Grundlagen

Graphen finden in der Informatik viele Anwendungsfälle. Sie werden sowohl im Bereich der Softwareanalyse und Softwarearchitektur als auch zum Veranschaulichen von bestimmten Problemen genutzt. Als Grundlage der folgenden Definitionen und Eigenschaften eines Graphen wurden die Bücher von Reinhard Diestel [2] und Uday Khedker [6] herangezogen.

Ein Graph G besteht aus zwei Mengen E und V, wobei E die Knoten und V die Kanten von G beinhalten. Eine Kante v_1 beschreibt die Abhängigkeit zwischen bestimmten Knoten und besteht immer aus genau zwei Knoten e_1, e_2. Je nachdem, ob es sich bei G um einen gerichteten oder ungerichteten Graphen handelt, lassen sich e_1 und e_2 als Start- bzw. Zielknoten definieren. Sollte es verschiedene Kanten mit den gleichen Start- und Zielknoten geben, können diese Kanten durch eine Mehrfachkante ersetzt werden. Mehrfachkanten bestehen neben den beteiligten Knoten zusätzlich aus einer natürlichen Zahl, welche für die Anzahl an ersetzten einfachen Kanten zwischen den Knoten steht. Ein Pfad P beschreibt einen Weg, um von einem Knoten zu einem Anderen zu gelangen. P besteht aus einer endlichen alternierenden Folge von Knoten und Kanten, welche mit zwei Knoten (Start- und Zielknoten) beginnt bzw. endet. Ein Graph ist zyklenfrei, falls es keinen Pfad gibt, der einen beliebigen Knoten e_1 als Start- und Zielknoten hat.

2.1 Call-Graph

Call-Graphen sind speziell gerichtete Graphen, die es ermöglichen ein Programm auf einem bestimmten Abstraktionsniveau zu analysieren. Zunächst werden sämtliche Methoden und Klassen als unterschiedliche Knoten abgebildet. Anschließend werden die Zusammenhänge der Knoten durch verschiedene Arten von

Kanten dargestellt. Eine gerichtete Kante zwischen zwei Knoten, welche Funktionen repräsentieren, beschreibt dabei die Caller-Callee-Beziehung. Rekursionen innerhalb eines Programms werden durch Zyklen dargestellt, welche sich nur über einen oder mehrere Knoten verteilen können. Aus Gründen der Transitivität und der Übersicht werden oftmals nur direkte Caller-Callee-Beziehungen in einem Call-Graphen mit einer entsprechenden Kante versehen. Die zweite Art von Kante verbindet Klassen und Methoden und beschreibt, welche Funktion zu welcher Klasse gehört. Um Auskunft über die Klassenhierarchie eines Programms zu geben, können optional auch Klassen durch gerichtete Kanten miteinander verbunden werden.

2.2 Kontrollflussgraph

Ein Kontrollflussgraph ist eine Verallgemeinerung eines Call-Graphen und beinhaltet zusätzlich Informationen über die Reihenfolge der aufgerufenen Anweisungen von den unterschiedlichen Methoden. Die Knoten in einem Kontrollflussgraph stehen für die verschiedenen Anweisungen, inklusive den Methodenaufrufanweisungen, einer Funktion. Eine gerichtete Kante $v_1 = (e_1, e_2)$, welche die Knoten e_1 und e_2 verbindet, beschreibt eine zeitliche Vorgänger- bzw. Nachfolger-Beziehung der beteiligten Instruktionen. In diesem Fall bedeutet es, dass die Anweisung e_2 bei einer Ausführung der Funktion chronologisch nach der Anweisung e_1 ausgeführt wird. Innerhalb eines Kontrollflussgraphen verfügen die Knoten meist immer über mindestens eine eingehende und mindestens eine ausgehende Kante. Es gibt nur zwei spezielle Arten von Knoten, bei denen dies nicht der Fall ist. Der Startknoten beschreibt die erste Anweisung, die ausgeführt wird, und hat demzufolge keinen zeitlichen Vorgänger bzw. keine eingehende Kante. Generell gibt es genau einen Startknoten in einem Kontrollflussgraphen. Das Gegenstück zum Startknoten ist der Zielknoten. Diese Art von Knoten verfügen über keinen Nachfolger und beschreiben üblicherweise die terminierenden Anweisungen einer Funktion. Je nach Aufbau der Funktion und eventuell anfallenden Fehlermeldungen kann ein Kontrollflussgraph über mehrere solcher Zielknoten verfügen.

3 Soot: a Java Optimization Framework

Soot ist ein Framework zur Codeanalyse bzw. Optimierung von Java Programmen [7]. Es bietet eine Menge von Einsatzmöglichkeiten und wurde oftmals als Basis für eine spezifische Programmanalyse verwendet. Streng genommen ist Soot ein Compiler, der den eingegebenen Quellcode transformiert und optimiert und in einem gewünschten Ausgabeformat bereitstellt. Soot kann sowohl in der Kommandozeile als auch als Plugin für Eclipse verwendet werden. Die Verwendung als Plugin ist jedoch zu empfehlen, da verschiedene Optionen von Soot über die bereitgestellte GUI zugänglicher sind.

Um eine möglichst effektive Codeanalyse zu garantieren, verfügt Soot über vier interne Coderepräsentationen: Jimple, Shimple, Baf, Grimp [8].

Jimple ist die primär verwendete Repräsentation von Soot und besteht aus einem stack-unabhängigen 3-address-Code. Dies bedeutet, dass jede Anweisung aus maximal drei Komponenten x, y, z besteht, die meist folgendermaßen verknüpft sind: $z = x\ op\ y$. op steht in diesem Zusammenhang für eine der 19 Jimple-Operationen. Die Stackunabhängigkeit wurde durch das Hinzufügen von weiteren lokalen Variablen erreicht. Aufgrund der geringen Komplexität von Jimple im Vergleich zum Java Bytecode, der mehr als 200 Operationen unterstützt und stack-basiert ist, eignet sich diese Form der Repräsentation wesentlich besser zur Analyse und Optimierung.

Eine zu Jimple sehr ähnliche Repräsentation ist Shimple, bei der es sich um eine Erweiterung handelt, die im gesamten Quellcode nur Einmal-Zuweisungen von Variablen erlaubt. Im Allgemeinen wird diese Art von Code-Repräsentation SSA (Static-Single-Assignment) genannt. Auf Grund der identischen Syntax und den gleichen Operationen lassen sich beide Repräsentationen nahezu identisch verwenden und es müssen kaum bis keine Änderungen in den Analysen vorgenommen werden. Das Ziel dieser Art von Repräsentation ist die Kombination der Vorteile von Jimple mit denen von SSA. Konstanten-Propagierung, bei der bereits zur Kompilierung Konstanten mit ihren entsprechenden Werten ersetzt werden, lässt sich beispielsweise wesentlich besser mit der Shimple-Repräsentation durchführen.

Die beiden beschriebenen Repräsentationen vereinfachen den zu analysierenden Code zwar stark, ermöglichen jedoch keine stack-basierte Analyse bzw. Optimierung. Die Coderepräsentation Baf ist im Gegensatz zu Jimple und Shimple eine stack-abhängige Repräsentation. Im Vergleich zu Java Bytecode besteht Baf jedoch nur aus etwa 60 Operationen, wobei eine Baf-Operation für unterschiedliche Instruktionen im Java Bytecode stehen kann. Zum Beispiel wurde von den verschiedenen Varianten einer Additionsoperation (double, integer,...) abstrahiert und eine einzelne allgemeine Operation eingeführt.

Der Nachteil an den bisher beschriebenen internen Repräsentationen von Soot ist die schlechte Lesbarkeit für menschliche Benutzer. Grimp ist eine Version von Jimple in der mehrere Operationen zusammengefasst werden können, wodurch der entstandene Code leichter zu lesen ist. Ein Beispiel für die unterschiedlichen Repräsentation von Jimple und Grimp liefert Tabelle 1.

Soot unterstützt verschiedene Quellen als Eingabeformat wie Java als Quell- und Bytecode, Android-Bytecode oder auch Jimple-Code. Nachdem sämtliche Transformationen getätigt worden sind, bietet Soot die Möglichkeit den optimierten Code ebenfalls in verschiedenen Formaten auszugeben und abzuspeichern. Die unterstützen Formate sind Java-Bytecode, Android-Bytecode, Jimple, Jasmin [1] und Dava [2], wobei letztere Soot-fremde Formate sind. Zu bemerken ist, dass Soot eine beliebige Kombination der verschiedenen Ein- und Ausgabeformaten erlaubt [3].

[1] http://jasmin.sourceforge.net/about.html Last access: 2015-01-30
[2] http://www.sable.mcgill.ca/dava/ Last acess: 2015-01-30
[3] http://sable.github.io/soot/ Last access: 2015-01-15

(1)	return this.a.x++ + m*n* (x*100+10);

(a) Java

(1)	$r0=this.a;
(2)	$i0=$r0.x;
(3)	$i1=$i0+1;
(4)	$r0.x=$i1;
(5)	$i2=m*n;
(6)	$i1=this.x;
(7)	$i1=$i1*100;
(8)	$i1=$i1+10;
(9)	$i2=$i2*$i1;
(10)	$i3=$i0+$i2;
(11)	return $i3;

(b) Jimple

(1)	$r0=this.a;
(2)	$i0=$r0.x;
(3)	$r0.x=$i0+1;
(4)	return $i0+m*n * this.x*100+10;

(c) Grimp

Tabelle 1. Vergleich von Java Bytecode, Jimple und Grimp [8]

3.1 Architektur von Soot

Soot basiert auf einer paketorientierten Architektur. Je nach gewünschter Art der Analyse lassen sich verschiedene Pakete, sogenannte Phasen, in den Ablauf von Soot einbinden [4]. Zusätzlich lassen sich die Phasen über verschiedene Einstellungen anpassen, so dass zum Beispiel gewisse Optimierungen innerhalb einer Phase nicht durchgeführt werden [4]. Auf diese Weise wird dem Benutzer die Möglichkeit gegeben Soot an die eigenen Bedürfnisse anzupassen und überflüssige Analyseschritte zu deaktivieren. So könnten im Falle einer stack-unabhängigen Codeanalyse sämtliche Phasen, die mit der Erzeugung und Optimierung von Baf-Code verknüpft sind, übersprungen werden. Die Abbildung 1 veranschaulicht, welche Pakete bei einer intraprozeduralen Analyse zur Verfügung stehen und in welcher Reihenfolge sie angewendet werden. Intraprozedural bezieht sich hierbei auf die Eigenschaft, dass jede Methode unabhängig analysiert wird und die ausgewählten Phasen einzeln auf die Methodenrümpfe angewendet werden. Im Folgenden wird genauer auf die einzelnen Phasen und deren Ausführungsreihenfolge eingegangen.

Der erste Schritt besteht in der Erzeugung einer Jimple-Repräsentation des zu analysierenden Codes. Dies geschieht, je nach Art des vorliegenden Codes, durch die Pakete Jimple Body Creation (jb) für class- und Jimple-Dateien oder Java To Jimple Body Creation (jj) für Java-Dateien. Je nach gewählten Einstellungen können in diesem Schritt bereits lokale, nicht verwendete Variablen oder sogar ganze Abschnitte von nicht erreichbarem Code eliminiert werden.

Nachdem die Jimple-Repräsentation des Quellcodes vorliegt, werden anschließend sämtliche Jimple-Phasen, bestehend aus Jimple Transformation Pack (jtp), Jimple Optimization Pack (jop) und Jimple Annotation Pack (jap), angewandt. Je nachdem, ob zusätzlich eine Shimple-Repräsentation erstellt werden soll, können zuvor die Pakete Shimple Transformation Pack (stp), Shimple Optimizati-

[4] https://ssebuild.cased.de/nightly/soot/doc/soot_options.htm Last access: 2015-01-09

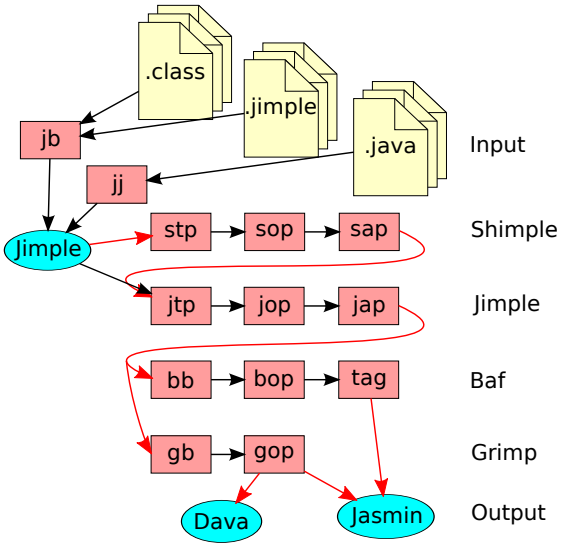

Abb. 1. Verfügbare Phasen bzw. Pakete von Soot (in rot gefärbt) bei einer intraprozeduraler Analyse [4]

on Pack (sop) und Shimple Annotation Pack (sap) durchlaufen werden. Diese Phasen sind allerdings für die gleichen Analyseschritte wie die entsprechenden Jimple-Pakete zuständig, mit dem Unterschied, dass weitere Transformationsmöglichkeiten auf Basis der Shimple-Repräsentation zur Verfügung stehen. Falls eine auf der Shimple-Repräsentation basierende Analyse gewünscht ist, lassen sich über das Paket Shimple Control (shimple) verschiedene Einstellungen zur Erstellung und Transformation vornehmen.

Die Pakete jtp und stp beinhalten bei einer unveränderten Soot-Variante keine Transformationen. Sie sind für eigens erstellte intraprozedurale Analysen seitens des Benutzer vorgesehen und können nach Belieben erweitert werden. Die beiden Optimierungsphasen jop und sop sind hingegen nicht leer und beinhalten bereits vorgefertigte Jimple- bzw. Shimple-orientierte Optimierungen. So können etwa mathematische Berechnungen, die lediglich von Konstanten abhängen, bereits während der Kompilierung durch das Ergebnis substituiert werden. Des Weiteren gibt es die Möglichkeit if-Anweisungen in der Jimple-Repräsentation, welche stets zu wahr oder falsch ausgewertet werden, durch entsprechende goto-Anweisungen zu ersetzen. Neben den beschriebenen Transformations- und Optimierungsprozessen bietet Soot ebenfalls die Möglichkeit den Quellcode mit gewissen Anmerkungen zu versehen. Hierfür stellt Soot sogenannte Tags zur Verfügung, die während der beiden Phasen jap und sap mit den zu markierenden Methoden oder Anweisungen verknüpft werden. Auf diese Weise erhält der Benutzer eine Auskunft über bestimmte Abschnitte seines Quellcodes. Zum Beispiel bietet die Option „null pointer colourer“ die Möglichkeit null-pointer-Referenzen als solche zu markieren und in Eclipse farbig hervorzuheben.

Nachdem sämtliche Shimple- und Jimplephasen abgeschlossen worden sind, werden abschließend entweder die Inhalte der Baf- oder Grimpphase angewendet. Mit den Paketen bb (Baf Body Creation) und gb (Grimp Body Creation) wird zunächst für jede optimierte und annotierte Methode die entsprechende Baf- bzw Grimp-Repräsentation erstellt. Aufbauend auf diesen Repräsentationen wird eine der beiden Optimierungsphasen bop (Baf Optimimization) und gop (Grimp Optimization) durchgeführt. Obwohl in der aktuellen Version von Soot beide Pakete keine Transformationen zur Verfügung stellen, können auch hier seitens der Benutzer eigene Analysen eingefügt und angewandt werden. Falls eine Baf-Repäsentation gewählt worden ist, bietet das Paket tag (Tag Aggregator) die Möglichkeit verschiedene Tags zu aggregieren. Dieses Vorgehen kann für das korrekte Annotieren von Anweisungen im schlussendlich erstellten Java-Bytecode von großer Bedeutung sein.

Soot bietet neben dem erwähnten intraprozeduralen Vorgehen ebenfalls die Möglichkeit einer Analyse, bei der das gesamte zu untersuchende Programm als eine Einheit aufgefasst wird. Dieses interprozedurale Vorgehen stellt Abhängigkeiten zwischen den einzelnen Klassen und Methoden dar und ermöglicht somit weitere Transformation- bzw. Optimierungsprozesse. Wie in Abbildung 2 zu sehen ist, werden hierbei vier weitere Pakete eingebunden.

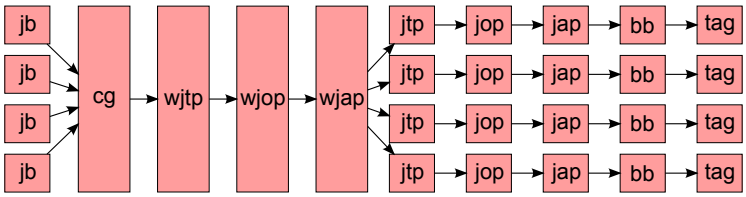

Abb. 2. Phasen bzw. Pakete von Soot bei einer interprozeduraler Analyse [4]

Anders als die bisher beschriebenen werden die Pakete cg (Call Graph Constructor), wstp (Whole Shimple Transformation Pack), wsop (Whole Shimple Optimization Pack), wjtp (Whole Jimple Body Creation), wjop (Whole Jimple Optimization Pack) und wjap (Whole Jimple Annotation Pack) nicht einzeln auf jede zu analysierende Methode angewendet sondern auf das gesamte Programm. Das Paket cg wird verwendet, wenn ein Call-Graph für eine Anwendung aufgestellt werden soll. In Abhängigkeit von den Phaseneinstellungen lassen sich unterschiedliche Algorithmen für das Erstellen des Graphen verwenden. Es stehen beispielsweise die auf einer statischen Analyse basierenden Algorithmen Class-Hierarchy-Analysis (CHA) und Rapid-Type-Analysis (RTA) zur Verfügung. Nachdem der Call-Graph berechnet worden ist, werden die Shimple-Pakete wstp und wsop, welche momentan keine zusätzlichen Funktionen bereitstellen, angewendet. Die weiteren Pakete wjtp, wjop und wjap entsprechen in ihrer Verwendung den intraprozeduralen Paketen jtp, jop und jap. Aufgrund der nicht auf Methoden beschränkten Anwendung lassen sich jedoch weitere

Transformationen und Optimierungen durchführen. Beispielsweise kann mit Hilfe von wjtp analysiert werden, welche Anweisungen für eine gleichzeitige Ausführung durch verschiedene Threads geeignet wären. Neben den bereits genannten Paketen stellt Soot zwei zusätzliche Preprozessor-Pakete, Whole Shimple Preprocessing Pack (wspp) und Whole Jimple Pre-processing Pack (wjpp), zur Verfügung. Diese standardmäßig leeren Pakete werden noch vor der Berechnung des Call-Graphen angewendet und können vom Nutzer erstellte Transformationen beinhalten.

4 Neo4j als Anwendungsbeispiel der NoSQL-Systeme

Die Vielzahl an verfügbaren Datenbankensystemen implementieren ein Modell, in dem gespeicherte Daten in Form von Tabellen vorliegen. Obwohl dies heutzutage ein etablierter Standard für Datenbanken ist, gibt es neben den relationalen Systemen eine Reihe weiterer, die über einen anderen Ansatz verfügen. NoSQL (Not only SQL) ist ein Begriff, der für Datenbanken verwendet wird, bei denen das zugrundeliegende Datenmodell nicht relational ist. Aufgrund der verschiedenen nicht-relationalen Datenmodellen unterliegen NoSQL-Systeme jedoch keiner eindeutigen Definition. Stattdessen müssen gewisse Eigenschaften erfüllt werden, um als No-SQL-System ausgezeichnet zu werden. Als Beispiel sollte das Datenbanksystem neben dem nicht-relationalen Datenmodell als Open Source frei verfügbar sein [3].

Allgemein lassen sich NoSQL-Systeme, gemäß dem verwendeten Datenmodell, in verschiedene Kategorien einteilen. Graphdatenbanken, dokumentbasierte Datenbanken und Key-Value-Datenbanken sind nur einige dieser Kategorien und bringen unterschiedliche Vor- bzw. Nachteile mit sich, die bei verschiedenen Anwendungsfällen unterschiedlich stark ins Gewicht fallen. Daher lässt sich die Frage, ob relationale Datenbanken oder NoSQL-Systeme besser geeignet sind, auch nicht einheitlich beantworten. Ein wichtiger Faktor für die Wahl des verwendeten Datenbanksystems sind die vorliegenden Daten. Oftmals ist es nicht zu empfehlen Informationen eines Datenmodells (z.B. tabellenbasiert) in ein anderes, unnatürlicheres Modell (z.B. graphenbasiert) zu überführen, da es bei großen Datenmengen unter Umständen zu Performanceproblemen kommen kann. Außerdem kann das korrekte Formulieren von präzisen Datenbankanfragen wesentlich schwieriger ausfallen, zumal gewisse Graphoperationen, wie zum Beispiel die Berechnung des kürzesten Weges zwischen zwei Knoten, nicht unterstützt werden. Bei Call- oder Kontrollflussgraphen ist es somit sinnvoll ein System, wie Neo4j zu wählen, welches Knoten und Kanten auf möglichst natürliche Weise darstellen kann.

4.1 Die Graphdatenbank Neo4j

Neo4j ist ein NoSQL-System und gliedert sich in die Reihe der Graphdatenbank ein [5]. Daten liegen somit nicht in Form von Tabellen vor, sondern werden durch

[5] http://neo4j.com/docs/2.1.6 Last access: 2015-01-07

Knoten und Kanten beschrieben, wobei jedes abgelegte Objekt über eine eindeutige ID verfügt. Knoten stehen in diesem Zusammenhang für die konkreten Entitäten wie Methoden oder Klassen eines Call-Graphen. Kanten beschreiben die Beziehungen zwischen den einzelnen Knoten und setzen die Objekte miteinander in Beziehung. Des Weiteren verfügt Neo4j über Konzepte wie Eigenschaften und Labels. Eine Eigenschaft besteht dabei immer aus einem Schlüssel-Wert-Paar und kann an einen Knoten oder eine Kante angehängt werden. So können Objekte innerhalb eines Graphen genauer beschrieben und von anderen Knoten der gleichen Art abgegrenzt werden. Labels hingegen sind für das Gruppieren von mehreren Knoten vorgesehen. Diese Art der Gruppierung erleichtert sowohl das Schreiben als auch das Auswerten von Anfragen an eine bestehende Datenbank.

Seit 2007 ist Neo4j als Open-Source-Projekt erhältlich und wird von der Firma NeoTechnology stets weiterentwickelt. Neben der frei verfügbaren Variante wird außerdem eine kommerzielle Version mit exklusivem Support angeboten [3]. Neo4j ist eine hauptsächlich in Java geschriebene Datenbank und wurde früher ausschließlich als eingebettete Bibliothek innerhalb einer JVM (Java Virtual Machine) benutzt. Heutzutage lässt sich Neo4j ebenfalls als eigenständiger Server konfigurieren und kann durch verschiedenen APIs, beispielsweise Java, jRuby, Python und C#, oder die HTTP/REST-Schnittstelle genutzt werden. Des Weiteren stellt Neo4j ein Transaktionsmanagement zur Verfügung, wodurch Änderungsoperationen in Transaktionen gekapselt werden und den ACID-Eigenschaften unterliegen. Anfragen an die Datenbank können entweder über die bereitgestellten Schnittstellen oder die deklarative Abfragesprache Cypher, welche in Kapitel 4.2 beschrieben wird, gestellt werden.

Seit der Version 2.0 implementiert Neo4j einen Neo4j-Browser, der über die URL `http://localhost:7474/browser/` von einem regulären Browser aus erreicht werden kann [5]. Die einzige Bedingung liegt darin, dass die Datenbank als externer Datenbankserver und nicht innerhalb einer JVM gestartet worden ist. Der Neo4j-Browser ist eine JavaScript-Anwendung und erleichtert dem Benutzer die interaktive Arbeit mit der bestehenden Datenbank. Beispielsweise wird ein Editor mit Syntaxhervorhebung für Cypher-Anfragen bereitgestellt und es besteht die Möglichkeit, häufig gestellte Anfragen zu speichern. Außerdem werden Antworten auf Anfragen dem Benutzer graphisch aufbereitet und durch Knoten bzw. Kanten in passender Form präsentiert. Serverkonfigurationen können ebenfalls bequem über die Browserschnittstelle geändert werden.

4.2 Die Anfragesprache Cypher

Die Graphdatenbank Neo4j verfügt seit Version 1.4 über eine eigene deklarative Anfragesprache. Mit Hilfe von Cypher können sowohl einfache Anfragen, die als Ausgabe Knoten oder Kanten fordern, als auch komplexe Operationen wie die Bestimmung eines kürzesten Pfades gestellt bzw. durchgeführt werden. Die Struktur von Cypher orientiert sich an SQL, indem ebenfalls über mehrere Bedingungsklauseln zur Verfügung stehen, die je nach Art der Anfrage verwendet oder ignoriert werden können. Bei der Auswertung von Anfragen findet ein

Pattern-Matching statt, sodass entlang Knoten und Kanten, die ein bestimmtes angegebenes Muster erfüllen, durch den Graph traversiert wird. Um Cypher möglichst zugänglich und intuitiv zu gestalten, haben sich die Entwickler bei der Syntax für Kreise und Pfeile entschieden. So werden Knoten innerhalb einer Anfrage stets mit runden Klammern () und Kanten mit dem Pfeilsymbol - -> dargestellt. Ein simples Pattern hat die Form () –> () und ist in Listing 1.1 aufgeführt.

Listing 1.1. Anfrage in Cypher: Wer sind Tims Freunde?

```
MATCH (Tim {name: 'Tim'})-[:kennt]->(Freund)
RETURN Tim, Freund
```

Im Allgemeinen beginnt eine selektierende Anfrage mit einer Menge an Startknoten von denen aus durch den Graph traversiert wird (start-Klausel). Sind keine expliziten Startknoten genannt, wie in Listing 1.1, wird stattdessen jeder Knoten im Graph als potentieller Startknoten betrachtet. Anschließend werden zu erfüllende Muster in der match-Klausel angegeben, wodurch zutreffende Knoten und Kanten selektiert werden. Falls die Menge der Ergebnisse noch weiter gefiltert werden soll, kann dies mit Hilfe der where-Klausel geschehen. Die return-Klausel definiert abschließend eine Projektion auf die gewünschten Rückgabewerte der Anfrage. Je nachdem, ob die Ausgabe noch sortiert oder eine maximale Anzahl an Ergebnissen haben soll, kann dies mit entsprechenden sort by- bzw. limit-Klauseln realisiert werden. Eine beispielhafte Anfrage ist in Listing 1.2 zu sehen.

Listing 1.2. Selektierende Anfrage mit den wichtigsten Klauseln [3]

```
START Person = (1,2)
MATCH (Person)-[:kennt]->(Freund)
WHERE Freund > 18
RETURN Freund.Name, Freund.Alter, Freund.Wohnort
SORT BY Freund.Name
```

Neben den selektierenden Anfragen gibt es in Cypher auch die Möglichkeit Knoten und Kanten mittels der create- oder delete-Klausel zu inserieren bzw. zu löschen. Attribute von Objekten können ebenfalls mit gewissen set- und remove-Klauseln verändert werden.

5 Implementierung des Transformationsprogramms

Die Transformation eines Java-Programms zu einem Call-Graphen in einer Neo4j-Datenbank besteht aus mehreren Schritten (Abbildung 3). Zuallererst wird mit dem beschriebenen Framework Soot eine temporäre Jimple-Repräsentation des Java-Quellcodes erstellt. Aufbauend auf dieser Coderepräsentation wird anschließend mittels des cg-Pakets von Soot ein approximierter Call-Graph berechnet. Für die Berechnung des Graphen wird der statisch orientierte Class-Hierarchy-Analysis Algorithmus (CHA) [1] verwendet, da sich dieser als gutes Mittelmaß zwischen Genauigkeit der Approximation des Call-Graphen und benötigter Laufzeit erwies. Im letzten Schritt wird durch den vorliegende Call-Graph traversiert und ein Abbild in die Neo4j-Datenbank übertragen.

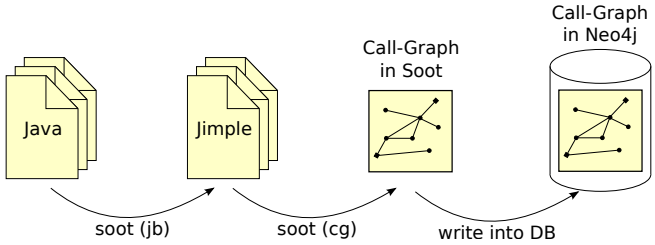

Abb. 3. Transformationsprozess von Java Code zum Call-Graphen in einer Neo4j-Datenbank

Der temporär durch Soot angelegte Graph besteht aus verschiedenen Soot-Klassen wie SootClass und SootMethod, welche die Klassen und Methoden des analysierten Programms beschreiben. Um die besuchten Objekte in die Datenbank zu kopieren, wird eine neue Klasse *DB* erstellt, innerhalb welcher sämtliche Einstellungen für die Neo4j-Datenbank vorgenommen werden. So werden beispielsweise der Speicherort, die verfügbaren Typen von Knoten und Relationen sowie ihre Eigenschaften festgelegt. Außerdem werden Listen zum Dokumentieren von bereits eingefügten Knoten und Kanten erstellt und Methoden zum Einfügen von jenen Objekten bereitgestellt. Sowohl das Starten bzw. Beenden der Datenbank als auch das Hinzufügen von Knoten oder Kanten wurde mittels der Java-API implementiert. Auf diese Weise können zum Beispiel Knoten durch die Neo4j-Klasse *Node* äußerst einfach erstellt und mit entsprechenden Attributen ergänzt werden (*set-Properties*-Methode). Für den weiteren Verlauf wird die *DB*-Klasse entsprechend dem Observer-Entwurfmuster als Listener bei der *CallGraphPublisher*-Klasse eingehängt, innerhalb derer der Code für die Transformation implementiert ist. Auf diese Weise befindet sich der Quellcode für die Datenbankanbindung und jener für den Umgang mit Soot-Objekten in einer jeweils eigenständigen Datei. Sobald ein neues Objekt zur Datenbank hinzugefügt werden soll, informiert der Publisher die DB-Klasse lediglich über die Art des Objekts, den Namen und eventuelle Eigenschaften. Daraufhin werden über die Datenbankanbindung sämtliche weiteren Schritte für das Erstellen des Objekts eingeleitet und darauf geachtet, dass für eine bereits übertragene Funktion oder Klasse kein zweiter Knoten konstruiert wird. Allerdings ergibt sich durch diese Überprüfung auch ein Problem auf das später in diesem Kapitel eingegangen wird.

Nachdem der Initialisierungsvorgang abgeschlossen ist und sowohl der Call-Graph als auch die Datenbank zur Verfügung stehen, beginnt die eigentliche Transformation. Im ersten Schritt werden, durch eine von Soot bereitgestellten Funktion, sämtliche Methoden ausgewählt, die von mindestens einer Kante im Call-Graphen den Startknoten bilden. Mittels einer Iteration über diese Menge, die als Grundlage der gesamten Transformation zu sehen ist, werden die nachfolgenden Schritte zunächst für die jeweilige Methode komplett abgeschlossen, bevor mit der nächsten Start-Methode gleichermaßen verfahren wird. Dabei lässt

sich einstellen, ob nur ein grober Überblick oder eine detaillierterer Graph mit erkennbarem Kontrollfluss erstellt werden soll. Sobald über alle Methoden aus der Start-Menge iteriert worden ist, wird der entstandene Datenbankzustand gespeichert und die Anwendung terminiert.

Für die nachfolgenden Erläuterungen sei *src* eine stellvertretende Methode aus der Menge der Start-Methoden. Zunächst wird abermals mit Hilfe von Soot eine Menge von Ziel-Methoden *targets* gebildet, wobei jede dieser Methoden *src* als Startknoten im zugrundeliegenden Call-Graph hat. Für dieses Paar, bestehend aus *src* und *targets* wird anschließend die Funktion *processMethod* aufgerufen. Hierbei handelt es sich um eine erste Übertragung der bereits zugänglichen Informationen an die Datenbank. Soot stellt Methoden bereit, um aus einer Methode sowohl die zugehörige Klasse als auch die Superklasse zu ermitteln. Auf diese Weise werden neben *src* und *targets* ebenfalls die zugehörigen Klassen als Knoten in die Datenbank übertragen und mit entsprechend beschrifteten Kanten, wie beispielsweise „ruft auf“ oder „ist Superklasse von“, versehen. Falls vom Nutzer nur ein grober Call-Graph gewünscht ist, fallen für das Paar $src, target$ keine weiteren Berechnungen an und es wird die nächste Methode aus der Start-Menge herangezogen. Abbildung 4 zeigt einen einfachen Call-Graphen für den Quellcode aus Listing 1.3, wobei die Systemfunktionen aus Gründen der Übersichtlichkeit weggelassen worden sind. Diese Einstellung wird zum Programmstart festgelegt und im Laufe der Arbeit noch genauer beschrieben.

Listing 1.3. Codebeispiel

```
public class Car extends Vehicle{
        protected void drive(){
                int round = 0;
                getInCar();
                for (int i = round;i<=3;i++){
                        driveFaster();
                }
                Random ran = new Random();
                int position = ran.nextInt(3) + 1;
                getOutOfCar();
                if(position == 1){
                        winningPose();
                        System.out.println("Yeah!");
                }else{
                        loosingPose();
                }
        }
}
```

Wie zu sehen ist, lässt sich bereits eine Caller-Callee-Hierachie erkennen, allerdings ohne jegliche Reihenfolge zwischen den aufgerufenen Methoden. Diese fehlenden Kontrollflussinformationen werden durch den Aufruf der Funktion $processMethodBody$, welche nur *src* als Parameter erhält, ermittelt. Soot stellt Funktionen bereit, um für eine Methode einen Kontrollflussgraphen zu erstellen, durch den anschließend traversiert werden kann. Die einzelnen Anweisungen des zugrundeliegenden Jimple-Codes, wie Funktionsaufrufe oder Variablenzuweisungen, liegen dabei als Objekte verschiedener Statement-Klassen vor. Auf diese Weise lassen sich die funktionsaufrufenden Statements, welche die einzig

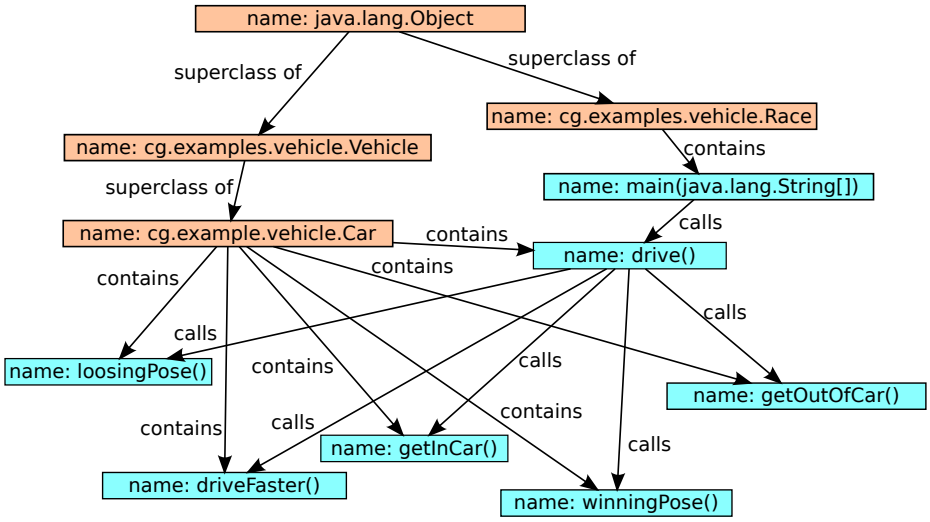

Abb. 4. Der Call-Graph zum Listing 1.3

relevanten Anweisungen für einen Call-Graphen sind, einfach filtern. Für den Großteil dieser gefilterten Statement ist das Übertragen in die Datenbank eine simple Angelegenheit, da nur eine nachfolgende Anweisung existiert und es zu keinen Verzweigungen kommt. Falls ein Statement jedoch ein Test ist, wie zum Beispiel $x >= 3$, sind meist zwei folgende Statements vorhanden und es müssen beide Pfade durch den Kontrollflussgraph berücksichtigt werden. Das Berechnen der folgenden Anweisung wird durch einen rekursiven Algorithmus durchgeführt, der für ein Statement s_1 jedes direkt folgende Statement s_2 und den damit verknüpften Pfad durch den Graph berücksichtigt.

Hierbei ist allerdings zu beachten, dass Anweisungen, die sich innerhalb einer Schleife befinden, theoretisch unendlich viele Nachfolger haben können und es durch den sich wiederholenden Schleifen-Test und die damit verbundene Verzweigung zu einem unendlichen Graphen kommen würde. Aus diesem Grund werden vor dem Ausführen des rekursiven Algorithmus sämtliche Schleifen-Statements mit einem sogenannten Soot-Tag markiert. Beim anschließenden Durchlaufen des Algorithmus werden diese Tags benutzt, um zu ermitteln wie oft die nachfolgenden Anweisungen für ein bestimmtes Statement bereits berechnet worden sind bzw. ob ein weiterer Durchlauf notwendig ist. Das Übertragen der Statements mit entsprechendem Vorgänger und Nachfolger wird, wie zuvor beschrieben, über die DB-Klasse durchgeführt. Zusätzlich werden die verschiedenen Statements mit ihren entsprechenden Methoden-Knoten im Call-Graph verknüpft, so dass dem Benutzer alle Eigenschaften eines Statements bzw. der zugehörigen Methode leicht zugänglich sind. In Abbildung 5 ist ein detaillierterer Call-Graphen für die Methode *cg.examples.vehicle.Car void drive()* zu sehen.

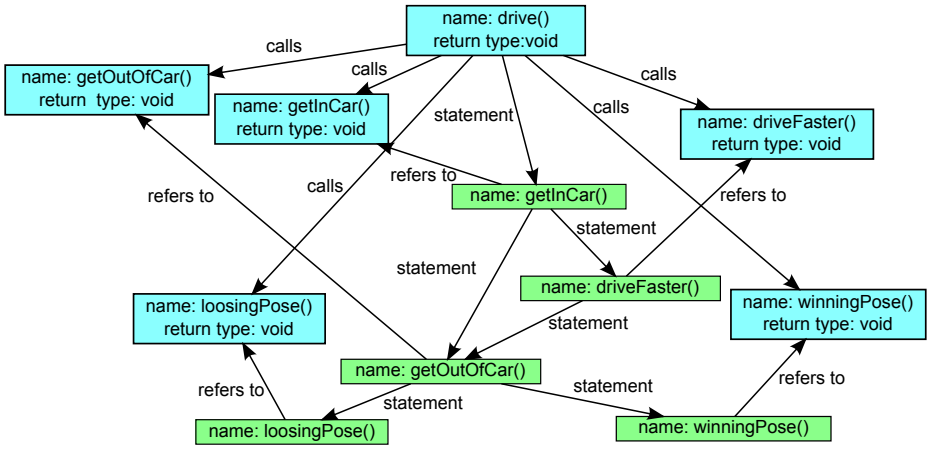

Abb. 5. Ein detaillierter Call-Graphen ohne Klassen zum Listing 1.3

Anzumerken ist, dass die fehlenden Knoten für die Systemfunktionen, wie beispielsweise *System.out.println*, in den Abbildungen 4 und 5 keine Fehler sind. Je nach Anwendungsfall kann es sinnvoll sein, sämtliche solche durch Standard-Java-Bibliotheken verfügbaren Methoden im Call-Graph zu berücksichtigen oder auch nicht. Aus diesem Grund lässt sich zum Programmstart festlegen, ob jene Funktionen ebenfalls in die Datenbank übertragen werden sollen.

Aufgrund der Überprüfung, ob es bereits einen Knoten für ein gewisses Objekt gibt, kann eine Anweisung nicht zwei Mal in dem Kontrollflussgraphen einer Funktion vorkommen. Falls ein Statement durch eine Schleife sich selbst als nachfolgendes Statement hat, besteht kein Problem und der entsprechende Knoten wird im Graphen mit einer reflexiven Kante versehen. Sobald allerdings eine Anweisung zwei Mal direkt hintereinander ausgeführt wird, ohne dass eine Schleife die Ursache dafür ist, wird kein Knoten für jede Anweisung erstellt. Stattdessen wird durch die Überprüfung beim Einfügen des zweiten Knoten festgestellt, dass bereits ein entsprechender Knoten existiert und somit nur noch eine reflexive Kante eingefügt werden muss. Im konstruierten Call-Graphen lässt sich daher nicht mehr unterscheiden, ob die gleiche Anweisung sequentiell mehrere Male hintereinander ausgeführt wird, oder ob es sich um dieselbe handelt.

6 Fazit und Ausblick

Die zu anfangs erwähnte Zielstellung der automatisierten Erstellung und Speicherung eines Call-Graphen für ein beliebiges Java-Programm konnte mit dem entwickelten Transformationsprogramm äußerst zufriedenstellend erreicht werden. Sowohl das Java-Framework Soot als auch die Graphdatenbank *Neo4j* ließen sich recht leicht für die gewünschten Zwecke konfigurieren und mit den entsprechenden APIs verknüpfen. Anzumerken ist, dass die Anbindung der Datenbank

über die Java-API von Neo4j tadellos verlief. Lediglich die Dokumentation von Soot war äußerst knapp und unübersichtlich gehalten, so dass es hierbei eine etwas größere Einarbeitungszeit benötigte. Für die Verifikation der Software wurde kein Beweisverfahren verwendet, sondern eine Reihe von unterschiedlich komplexen Anwendungsbeispielen. Obwohl bei diesen Beispielen stets ein korrekter Graph berechnet worden ist, sollte festgehalten werden, dass es keine Garantie für die Korrektheit der Implementierung gibt. Die Laufzeiten für die Transformationen der Anwendungsbeispiele lagen allesamt im Sekundenbereich, wobei *Soot* für die initialen Berechnungen sowie das Erstellen des Call-Graphen die meiste Zeit in Anspruch nahm. Für äußerst komplexe Software lässt sich allerdings keine Aussage über die Laufzeit bzw. die Skalierung dieser machen.

Um in Zukunft noch präzisere Call-Graphen erstellen zu können, sollte das bereits erwähnte Problem mit den reflexiven Kanten beseitigt werden. Die nächstliegendste Lösung ist ein Vergleich, ob es sich bei den zu übertragenden Statements um dasselbe oder das Gleiche handelt. Eine weitere Optimierungsmöglichkeit besteht im Laden eines vorher berechneten und abgespeicherten Call-Graphen in die Transformationssoftware. Für eine Software, bei der kleine Änderungen durchgeführt worden sind, müsste bei einer erneuten Erstellung des Call-Graphen nicht der komplette Graph neu berechnet und übertragen werden, sondern nur die veränderten Komponenten.

Literatur

[1] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In M. Tokoro and R. Pareschi, editors, *Proceedings of the 9th European Conference*, volume 952 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 1995.

[2] R. Diestel. *Graphentheorie*. Springer-Lehrbuch. Springer, 1996.

[3] S. Edlich, A. Friedland, J. Hampe, B. Brauer, and M. Brückner. *NoSQL: Einstieg in die Welt nichtrelationaler Web 2.0 Datenbanken*. Carl Hanser Verlag GmbH & CO. KG, München, 09 2011.

[4] A. Einarsson and J. D. Nielsen. A survivor's guide to java program analysis with soot, 2008. URL `http://www.brics.dk/SootGuide/`. Last access: 2015-01-12.

[5] M. Hunger. *Neo4j 2.0: Eine Graphdatenbank für alle*. Entwickler.press, Frankfurt am Main, 2014.

[6] U. Khedker, A. Sanyal, and B. Karkare. *Data Flow Analysis: Theory and Practice*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 2009.

[7] P. Lam, E. Bodden, O. Lhotak, and L. Hendren. The soot framework for java program analysis: a retrospective. In *Proceedings of the Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, October 2011.

[8] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99. IBM Press, 1999.