Heinrich Heine
Universität
Düsseldorf

INSTITUT FÜR INFORMATIK
Softwaretechnik und Programmiersprachen
Universitätsstr. 1 D–40225 Düsseldorf

Ein JavaScript Abhängigkeitsgraph

Kevin Schwarz

Bachelorarbeit

Beginn der Arbeit:	16. Oktober 2019
Abgabe der Arbeit:	27. Januar 2019
Gutachter:	Prof. Dr. Michael Leuschel
	Dr. John Witulski

Ehrenwörtliche Erklärung

Hiermit versichere ich die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Düsseldorf, den 27. Januar 2019

Kevin Schwarz

Zusammenfassung

Ziel dieser Bachelorarbeit war es im Rahmen des Bauhaus Projekts ein Programm zu entwickeln, welches eine JavaScript-Datei in das RFG-Format übersetzt. Eine große Herausforderung war es den dynamischen Charakter dieser Programmiersprache auf eine statische Darstellung abzubilden. Es sind alle nötigen Komponenten und Abhängigkeiten eines JavaScript Programms implementiert wurden, die für Analysen, wie Erkennung von totem Code, Zyklenerkennung und Architekturanalysen, nötig sind. Es wird erläutert durch welche Methoden ein RFG aus einem Quelltext generiert wird und warum die Darstellung als RFG so effizient und nützlich für viele Analysen ist.

Inhaltsverzeichnis

1 Motivation

Diese Bachelorarbeit behandelt das Thema der Software-Erosion, ein Begriff, der von der Axivion GmbH[Axia] geprägt wurde. Software-Erosion bezeichnet den steten und schleichenden Verfall der inneren Software-Struktur[1], welcher unumgänglich bei großer und auch kleiner Software ist.

Da die Wartung von Software eine Menge Arbeitszeit beansprucht und somit weniger Zeit in die eigentliche Entwicklung der Software fließt, ist es von hohem Interesse diese benötigte Zeit auf ein Minimum zu reduzieren. Die Axivion Bauhaus Suite beinhaltet Tools für Architekturmodellierungen, Architekturprüfungen, Klonerkennung und -management, Prüfung von Coderichtlinien, statische Code-Analysen, Erkennung von totem Code, Zyklenerkennung, etc. für die Programmiersprachen C, C++ und C#.

Ziel dieser Bachelorarbeit war es im Rahmen der Axivion Bauhaus Suite das Angebot der Programmiersprachen, welches sich im Moment auf die Sprachen C, C++, C# und Java beschränkt, auch auf JavaScript auszubauen. JavaScript ist eine Skriptsprache, welche vorzugsweise und in großem Ausmaß in Webbrowsern neben HTML und CSS, wie auch auf Servern und Microcontrollern zum Einsatz kommt. Daher ist es von großem Interesse für diese weitverbreitete Programmiersprache ebenfalls die vielfältigen Analysen der Axivion Bauhaus Suite bereitzustellen.

2 Grundlagen

Dieses Kapitel dient dazu die Bestandteile und Grundlagen dieser Bachelorarbeit, welche sich in den darauffolgenden Kapiteln wiederfinden und zum Verständnis dieser Arbeit nötig sind, vorzustellen bzw. zu erläutern.

2.1 JavaScript

JavaScript[GMNR10] ist zurzeit die am meisten benutzte Programmiersprache weltweit. Dieses Resultat ergibt sich aus einer Umfrage von Stack Overflow[2]. Laut dieser Umfrage liegt JavaScript auf Platz 1 gefolgt von HTML/CSS, SQL, Python und Java. Anfangs wurde sie in Webbrowsern neben HTML und CSS benutzt, um Webseiten ein dynamisches und interaktives Verhalten zu verleihen. Heutzutage kann JavaScript auch im Backend-Bereich von Servern oder auch dank des kleinen Sprachkerns auf Microcontrollern eingesetzt werden. Die Syntax ähnelt der Programmiersprache C und ist minimal gehalten.

[1] Zitiert von `https://www.axivion.com/de`

[2] `https://insights.stackoverflow.com/survey/2019`

Sie ist eine prototypenbasierte und dynamisch typisierte Skriptsprache. Dynamisch typisiert bedeutet, dass der Datentyp einer Variable zur Laufzeit geändert werden kann. Der Datentyp ist also keine Eigenschaft der Variable, sondern die des Wertes, der der Variable zu gewiesen ist. Prototypenbasierte Programmierung bedeutet, dass Objekte nicht durch das Instanziieren einer Klasse erzeugt werden, sondern durch Klonen eines Objektes, dem Prototyp. Außerdem benutzt JavaScript das Konzept des Duck-Typings, bei dem der Typ eines Objekts nicht durch seine Klasse festgelegt ist, sondern durch dessen Attribute und Methoden.

JavaScript wurde ursprünglich von Brendan Eich unter dem Namen LiveScript entwickelt. Netscape Communications und Sun Microsystems entwickelten die Skriptsprache anschließend unter dem Namen JavaScript weiter. Die aktuellste Umsetzung nennt sich ECMAScript [Ecm19], welche eine von Ecma International in ECMA-262 und ISO / IEC 16262 standardisierte Spezifikation für Skriptsprachen ist.

```
1  var p = new Person("Peter", 54);
2
3  // Prototypenbasierte Syntax
4  function Person(name, age) {
5    this.name = name;
6    this.age = age;
7  }
8
9  // Klassenbasierte Syntax
10 class Person {
11   constructor(name, age) {
12     this.name = name;
13     this.age = age;
14   }
15 }
```

Listing 1: Prototypenbasierte Syntax vs. klassenbasierte Syntax

ECMAScript führte, neben anderen Funktionalitäten, eine neue Syntax für Prototypen ein, die der Syntax einer von klassenbasierten Programmiersprachen ähnelt. Listing 1 zeigt die neue Syntax im Vergleich zur prototypenbasierten Syntax.

2.2 Abstrakter Syntaxbaum

Ein abstrakter Syntaxbaum (engl. Abstract Syntax Tree, kurz AST) ist eine Baumstruktur, welche den syntaktischen Aufbau eines Quellcodes in einer abstrakteren Form darstellt. Abstrakt in dem Sinn, dass er nicht jedes Detail des Quellcodes direkt wiederspiegelt, sondern nur den strukturellen Programmaufbau abbildet und Elemente, wie Einrückungen oder Zeilenumbrüche, welche in der Regeln wenig bis keine nützlichen Informationen bereitstellen, ignoriert.

Listing 2 zeigt ein Beispiel einer in JavaScript geschriebenen Funktion, welche zwei Varia-

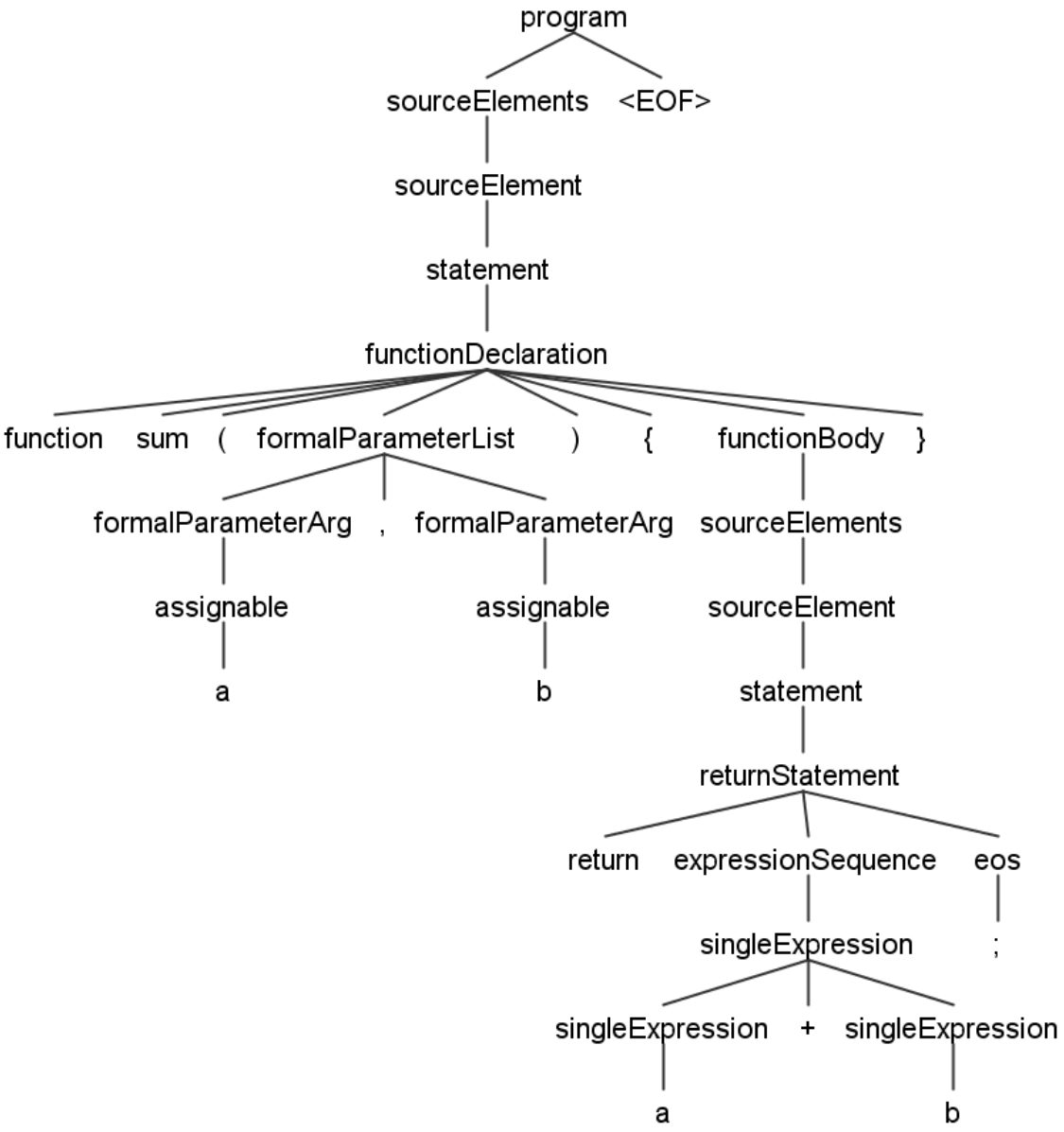

Abbildung 1: Syntaxbaum der Funktion zum Addieren zweier Variablen

```
1 function sum(a, b) {
2   return a + b;
3 }
```

Listing 2: JavaScript Funktion zum Addieren zweier Variablen

blen addiert und zurück gibt. In Abb. 1 ist der daraus resultierende abstrakte Syntaxbaum zu sehen, welcher durch AntLR mit einer bereitgestellten Grammatik[3] erzeugt wurde.

2.3 AntLR

AntLR [Par13] wird seit 1989 von Terence Parr, einem Professor für Informatik an der Universität von San Francisco, entwickelt und ist in der vierten Version frei erhältlich.

```
1 grammar Addition;
2
3 // Parser Regeln
4 sum: NUMBER '+' NUMBER;
5
6 // Lexer Regeln
7 NUMBER: [0-9]+;
```

Listing 3: Grammatik zum parsen einer Addition

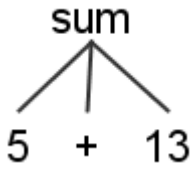

Abbildung 2: AST einer Addition

Über eine Grammatik, wie zum Beispiel die in Listing 3, generiert AntLR einen Parser, der abstrakte Syntaxbäume dieser Sprache erzeugen und traversieren kann. Durch eine Eingabe von zum Beispiel `5+13` wird der in Abb. 2 abgebildete AST erzeugt. AntLR stellt darüber hinaus einen Visitor bereit, der über den generierten Syntaxbaum traversieren kann. Dieser Visitor kann in ein Projekt eingebunden werden und somit über den AST traversieren um Aktionen an den verschiedenen Knoten des Syntaxbaums auszuführen.

[3] `https://github.com/antlr/grammars-v4/tree/master/javascript`

2.4 Projekt Bauhaus / Axivion Bauhaus Suite

Das Projekt Bauhaus[EKP+99] entstand aus der Zusammenarbeit des Instituts für Informatik der Universität Stuttgart, Abteilung Programmiersprachen und Compiler, und der Fraunhofer Einrichtung für Experimentelles Software-Engineering (FhG IESE), Kaiserslautern. Um die Wartung einer Software zu vereinfachen, wurden Techniken erforscht um die Architektur von Software zu analysieren, darzustellen und gegen eine Soll-Architektur zu prüfen. Wird die Architektur bei Änderungen an der Software nicht wiederholt geprüft, kann es zu einem architektonischen Verfall führen. Zum Beispiel kann die Ist-Architektur verletzt werden oder es kann zum unnötigen Duplizieren von Code führen. Inzwischen wird das Projekt auch an der Universität Bremen und durch die Axivion GmbH weiterentwickelt.

Das Hauptprodukt der Axivion GmbH ist die Axivion Bauhaus Suite[RVP06]. Sie ist eine ausgereifte Toolsuite, die viele Werkzeuge zur Architekturmodellierung, Architekturprüfung, Klonerkennung und -management[BYM+98][Ber07], Prüfung von MISRA Regeln, Prüfung von Coderichtlinien, statischen Code-Analyse, Race-Conditions-Analyse, Delta-Analyse, Metrikenanalyse, Erkennung von totem Code und Zyklenerkennung basierend auf den Programmiersprachen C, C++,C# und Java[Bay13] bereitstellt.

2.5 Ressourcenflussgraph

Der Ressorucenflussgraph (engl. Resource Flow Graph, kurz RFG) ist ein gerichteter Graph, der den Ressourcenfluss eines Programms auf einer höheren Abstraktionsebene abbildet, indem er Sprachelemente wie Variablen, Funktionen, Klassen, etc. als Knoten und die Abhängigkeiten zwischen diesen Sprachelementen als gerichtete Kanten darstellt. Desweiteren können Knoten und Kanten Attribute besitzen wie zum Beispiel die Stelle im Quellcode, wo das Sprachelement deklariert wurde. Ein RFG besteht aus mehreren *Views*, die jeweils einen Teilgraph der Gesamtarchitektur beinhalten. Die `Declaration Facts` *View* zum Beispiel beinhaltet alle deklarationsbezogenen Informationen, wohingegen die `Declaration Call` *View* alle Funktionsaufrufe beinhaltet. Die `File` *View* enthält wiederum eine hierarchische Sicht der Dateien aus denen ein Programm besteht. Die Namensgebung der Knoten, Kanten und *Views* ist an keine Richtlinien gebunden. Sie können frei gewählt und angepasst werden. Das hier entwickelte Programm hält sich allerdings an die Konvention der schon existierenden Programme für C, C++ und C#/.NET.

Auf dem RFG lassen sich viele graphentheoretische Probleme definieren, wodurch er viele Analysen ermöglicht. Dazu zählt das Erkennen von statischen Aufrufzyklen und das Erkennen von statisch totem Code. Desweiteren ist eine Architekturanalyse möglich, indem eine modellierte Architektur mit dem RFG auf Konsistenz abgeglichen wird.

Der RFG kann mit Hilfe des Graphical Visualisers (kurz GraVis) aus der *Axivion Bauhaus Suite* visualisert und manipuliert werden. Gravis bietet die Möglichkeit die oben

genannten Analysen durchzuführen. Eine weitere nützliche Funktion von Gravis ist die Quellcode-Anbindung. Wenn bei der Erzeugung eines RFGs den Knoten und Kanten Informationen über ihr Auftreten im Quellcode übermittelt werden, dann bietet Gravis die Möglichkeit die Position eines Knotens oder einer Kante im Quellcode anzuzeigen.

3 Implementierung

Dieses Kapitel beschreibt die Vorgehensweise bei der Implementierung des Programms *js2rfg*, welches aus einer übergebenen JavaScript Datei einen RFG erzeugen soll.

Der Hauptteil dieser Arbeit befasst sich damit eine dynamische Programmiersprache statisch zu analysieren. Dies stellt allerdings einige Schwierigkeiten dar, weil viele dynamische Komponenten gar nicht oder nur als große Überabschätzung auf eine statische Darstellung übertragbar sind. Hier war zu entscheiden welche dieser Komponenten dargestellt werden können und welche nicht.

3.1 behandelt den Aufbau des RFGs und die Darstellung der Komponenten und Abhängigkeiten von JavaScript als RFG. 3.2 erläutert wie *js2rfg* aufgebaut ist und wie JavaScript in den RFG übersetzt wird.

3.1 JavaScript als RFG

Für die Implementierung wurde JavaScript zu erst auf seine Komponenten bzw. Sprachelemente analysiert. In den folgenden Unterkapiteln werden die Sprachelemente und Abhängigkeiten, die in den RFG übersetzt werden, vorgestellt. Es wird erläutert, welche Stellen des Quellcodes *js2rfg* erkennt und übersetzt. Der Abschnitt 3.1.1 listet alle Knoten, die es im RFG geben wird, auf. Zu jedem Knoten wird ein Quellcode-Beispiel und der daraus generierte RFG gezeigt. Abschnitt 3.1.2 verwendet das gleiche Darstellungsmuster für die Kanten im RFG. In Abschnitt 3.1.3 werden die verschiedenen Views, aus denen der RFG besteht, beschrieben und es wird erläutert welche Knoten und Kanten in welchen View gehören. Zu jedem View gibt es ein RFG-Schema, welches die verschiedenen Knoten und dessen möglichen Abhängigkeiten zueinander darstellt. Als Vorlage dienten die bereits Vorhanden RFG-Schemata aus der Axivion Dokumentation[Axib].

3.1.1 Knoten

3.1.1.1 Entry: Einstiegspunkt

JavaScript ist eine interpretierte Programmiersprache. Das bedeutet, dass der Programmcode Zeile für Zeile abgearbeitet wird. Der Einstiegspunkt des Programms ist im Gegen-

satz zu kompilierten Programmiersprachen, wo der Einstiegspunkt eine bestimmte Methode ist, in Zeile 1. Um die Analyse von totem Code zu ermöglichen muss der RFG einen Startknoten besitzen von dem aus die Analyse starten kann. Bei kompilierten Programmiersprachen, bei denen der Einstiegpunkt bereits eine Methode ist und somit auch schon als Knoten im RFG nach der Übersetzung existiert, wird kein zusätzlicher Knoten, der den Einstiegspunkt simuliert, benötigt. In interpretierten Sprachen wie JavaScript wird allerdings solch ein Knoten gebraucht. Dem RFG wird ein künstlicher Knoten vom Typ *Routine*, `.entry` genannt, hinzugefügt.

```
1 f();
2 function f() {}
```

Listing 4: Funktionsaufruf im globalen Kontext

```
1 function .entry() {
2   f();
3   function f() {}
4 }
```

Listing 5: Funktionsaufruf im globalen Kontext

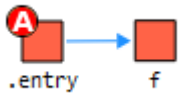

Abbildung 3: RFG Beispiel Funktionsaufruf im globalen Kontext

Listing 4 zeigt einen Funktionsaufruf im globalen Kontext. Da das ganze Programm von einer künstlichen Methode umschlossen wird, beispielsweise in Listing 5 veranschaulicht, gibt es eine Abhängigkeit, nämlich einen Funktionsaufruf, von Knoten `.entry` und Knoten `f`. Diese Abhängigkeit symbolisiert die dunkelblaue gerichtete Kante zwischen den beiden Knoten in Abb. 3.

3.1.1.2 Variable: Variablen

Variablen werden in JavaScript durch die die Schlüsselwörter `var`, `let` oder `const`, gefolgt vom Namen der Variable und einem Semikolon, deklariert. Optional kann die Variable direkt initialisiert werden. Variablen, die mit `var` und `let` beginnen, dürfen im späteren Verlauf neue Werte zugewiesen werden. Eine Variable, die mit `const` beginnt, darf nicht neu zugewiesen werden.

```
1 var x;
2 let y;
3 const z = 4;
```

Listing 6: Mögliche Deklarationen von Variable

Listing 6 zeigt einige Beispiele möglicher Deklarationen. *js2rfg* erkennt diese Stellen und generiert für jede deklarierte Variable einen Knoten vom Typ *Variable* wie Abb. 4 zeigt.

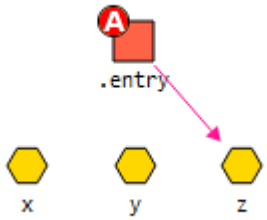

Abbildung 4: Der RFG von Listing 6

Zu beachten ist, dass der RFG nur die globalen Variablen abbildet, da lokale Variablen in Funktionen zu wenig Bedeutung für die gewünschten Analysen haben und sie sonst den RFG unübersichtlicher machen.

3.1.1.3 Routine: Funktionen

Funktionen können über mehrere Möglichkeiten deklariert werden. Diese Möglichkeiten werden in Listing 7 aufgelistet.

```
1  function f() {
2    return 'f';
3  }
4
5  var g = function() {
6    return 'g';
7  }
8
9  var h = () => {
10   return 'h';
11 };
```

Listing 7: Mögliche Deklarationen von Funktionen

Die am weitesten verbreitete Möglichkeit, sowie sie in vielen anderen Programmiersprachen benutzt wird, ist in Zeile 1-3 zu sehen. Das Schlüsselwort `function` signalisiert, dass eine Funktion deklariert wird. Bei den zwei anderen Möglichkeiten, die in Zeile 5-7 und in Zeile 9-11 zu sehen sind, wird einer Variable eine Funktion zugewiesen. Hier wird von einer *Anonymen Funktion* gesprochen. In JavaScript ist eine Funktion nämlich nichts anderes als ein Objekt. Dieses Verhalten erweist sich als problematisch, da wenn eine Funktion nur ein Objekt ist, welches einer Variable zugewiesen werden kann und diese Variable dynamisch typisiert ist, dann kann der Variable im späteren Programmablauf

ein anderer Wert z. B. eine Zahl zugewiesen werden und die Funktion ist somit überschrieben worden. Das gleiche Problem tritt bei der ersten Möglichkeit auch auf. Es wird zwar mit dem Schlüsselwort `function` eine Deklaration einer Funktion eingeleitet, aber im Hintergrund wird eine Variable namens `f` erzeugt, welche durch eine Funktion initialisiert wird. `f` kann im späteren Programmablauf weiter als Variable verwendet werden wie Listing 8 demonstriert.

```
1  function f() {
2    return 'f';
3  }
4  f = 5;
```

Listing 8: Funktion als Variable verwenden

Variablen sollten niemals für verschiedene Zwecke verwendet werden. Es sollte für jeden Verwendungszweck eine eigene Variable deklariert werden um Verwirrungen und unsauberer Programmierung vorzubeugen. Um Fälle abzudecken, in denen Variablen doch für mehrere Zwecke verwendet werden, generiert *js2rfg* für jede Deklaration einer Funktion einen *Routine* Knoten, der die Funktion repräsentiert, und einen *Variable* Knoten. Somit ist gewährleistet, dass, wenn Variablen, die Funktionen verkörpern sollen, anderweitig benutzt werden, der RFG auch diese Vorkommen als Variable- und nicht als Funktion-Knoten darstellen kann.

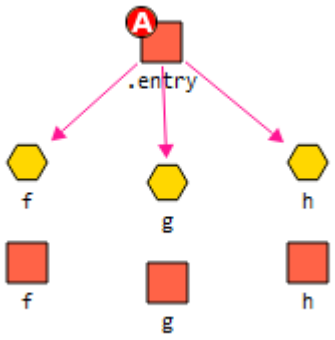

Abbildung 5: Der RFG von Listing 7

Abb. 5 zeigt den aus Listing 7 generierten RFG. Dort ist zu erkennen, dass für jede Funktion ein *Routine* Knoten, der die Funktion repräsentiert, und ein *Variable* Knoten, der die Variable in Fällen von Wiederverwendung repräsentiert, existiert. Zusätzlich wird für einen *Routine* Knoten die Anzahl der Parameter als Information abgespeichert. Diese Information kann bei späteren Analysen weiterzuverwendet werden.

3.1.1.4 Type: Prototypische Objekte

JavaScript ist eine prototypenbasierte Programmiersprache, was bedeutet, dass es keine Klassen gibt. Stattdessen gibt es Konstruktoren und prototypische Objekte. Ein Konstruktor ist eine normale Funktion. Zu einem Konstruktor wird diese Funktion, wenn sie nicht wie ein normaler Funktionsaufruf mit `f()`, sondern mit dem Schlüssewort `new` (`new f()`) aufgerufen wird.

```
1  var a = new Animal("Jaguar", 13);
2
3  function Animal(species, age) {
4    this.species = species;
5    this.age = age;
6  }
```

Listing 9: Konstruktor eines Objektes

Das Schlüsselwort `this` verweist auf den Kontext. Bei Verwendung von `new` wird für den folgenden Funktionsaufruf ein neuer Kontext geschaffen. Wenn nun in der Funktion `this` verwendet wird, bezieht sich das auf den neuen Kontext. `this.species = species` in Zeile 3 von Listing 9 gibt dem Kontext eine neue Eigenschaft. Am Ende der Funktion existiert ein verstecktes `return this`, welches den in der Funktion erzeugten Kontext zurückgibt und dann z. B. in Zeile 1 in die Variable `a` speichert. Die Funktion `Animal` ist weiterhin eine normale Funktion, die ohne das Schlüsselwort `new` aufgerufen werden kann. Dementsprechend, wie in 3.1.1.3 erklärt, ist `Animal` zeitgleich auch eine Variable. Der aus Listing 9 resultierende RFG ist in Abb. 6 zu sehen.

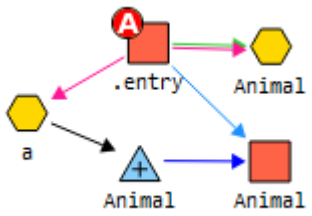

Abbildung 6: Der RFG von Listing 9

3.1.1.5 Class: Klassen

Dank ECMAScript können Prototypen auch durch eine klassenähnliche Syntax definiert werden. Im RFG werden diese Klassen jedoch nicht mit dem gleichen Knoten wie prototypische Objekte (3.1.1.4) abgebildet, sondern bekommen einen seperaten Knotentyp *Class*, damit auch im RFG die unterschiedliche Definition im Quellcode veranschaulicht wird.

```
1  var a = new Animal("Jaguar", 13);
2
3  class Animal {
4    constructor(species, age) {
```

```
5       this.species = species;
6       this.age = age;
7     }
8   }
```

Listing 10: klassenähnlicher Konstruktor

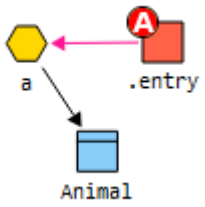

Abbildung 7: Der RFG von Listing 10

Listing 10 zeigt ein Beispiel einer solchen klassenähnlichen Objektdefinition. Abb. 7 zeigt den daraus generierten RFG.

3.1.1.6 Directory, File und Module: Verzeichnis und Datei

Ein RFG beinhaltet auch eine hierarchische Darstellung der Quelldateien. Der Knoten *Directory* repräsentiert die Verzeichnisse und der *File* bzw. *Module* Knoten repräsentiert die Datei. Der Konvention halber gibt es zwei verschiedene Knoten eine Datei im RFG darzustellen.

3.1.2 Kanten

3.1.2.1 Variable_Set bzw. Variable_Use: Variablen Zuweisung bzw. Verwendung

Variable können auf zwei verschiedene Arten benutzt werden. Die erste ist ihnen einen Wert zu zuweisen, die zweite ist deren Wert abzurufen.

```
1  var x = 1;
2  var y = x;
```

Listing 11: Variablen Zuweisung und Verwendung

Zeile 4 in Listing 11 demonstriert eine Zuweisung von `x` und Zeile 5 demonstriert wieder eine Zuweisung von `x`, aber auch eine Verwendung von `y`.

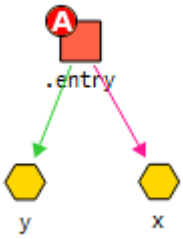

Abbildung 8: Der RFG von Listing 11

Zuweisungen werden durch die Kante *Variable_Set* (pinke Kante) repräsentiert und Verwendungen durch die Kante *Variable_Use* (hellgrüne Kante) wie in Abb. 8 illustriert. Die gerichtete Kante zeigt von einer Funktion, in der die Zuweisung oder Verwendung auftritt, auf die dementsprechende Variable.

3.1.2.2 Static_Call: Aufruf von Funktionen

```
1 function f() {}
2
3 function g() {
4   f();
5 }
```

Listing 12: Beispiel vom Aufrufen von Funktionen

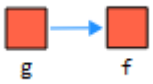

Abbildung 9: Der RFG von Listing 12

In Listing 12 wird die Funktion `f` von der Funktion `g` aufgerufen. Bei solchen Aufrufen wird eine gerichtete Kante *Static_Call* (hellblau) von der Funktion, die aufruft, zur Funktion, die aufgerufen wird, im RFG erzeugt wie Abb. 9 zeigt.

3.1.2.3 Constructed_By: Objekterzeugung über eine Funktion

Wie schon in 3.1.1.4 beschrieben, können Objekte durch normale Funktionen erzeugt werden. Diese Beziehung zwischen der Funktion, die den Bauplan für das Objekt darstellt, und des Objekts, welches daraus resultiert, wird im RFG durch die gerichtete Kante

Constructed_By (dunkelblau) dargestellt. Ein Beispiel dazu ist in Listing 9 und Abb. 6 zu sehen.

3.1.2.4 Of_Type: Typisierung

Obwohl die Variablen in JavaScript dynamisch typisiert sind, gibt es im RFG eine Beziehung zwischen Variablen und prototypischen Objekten bzw. Klassen, damit eine objektorientierte Analyse des RFGs möglich ist. Es kann allerdings nicht abgebildet werden, dass, wenn eine Variable im späteren Programmverlauf einen anderen Typ annimmt, gar keine Beziehung mehr zu diesem Objekt bzw. zu dieser Klasse hat. Da ein RFG allerdings in der Regel eine Überabschätzung darstellt, ist dies nicht weiter schlimm. Eine derartige Beziehung wird in Abb. 6 durch die schwarze gerichtete *Of_Type* Kante vom Knoten a zum Knoten Animal dargestellt.

3.1.2.5 Enclosing: Verschachtelung

Eine *Enclosing* Kante wird verwendet, wenn ein Knoten einem anderen Knoten untergeordnet ist. Er wird zum Beispiel zwischen den Directory Knoten verwendet. In Gravis wird er als eine Art Verschachtelung dargestellt.

3.1.3 Views

3.1.3.1 Declaration Facts und Code Facts

Für die auf C basierenden Programmiersprachen gibt es zwei verschiedene Views, Declaration Facts und Code Facts. Der Konvention halber existieren die zwei Views auch im JavaScript RFG, haben aber die gleiche Bedeutung, da es keine Seperation von Deklaration und Implementierung in JavaScript gibt. Abb. 10 zeigt das RFG-Schema der beiden Views.

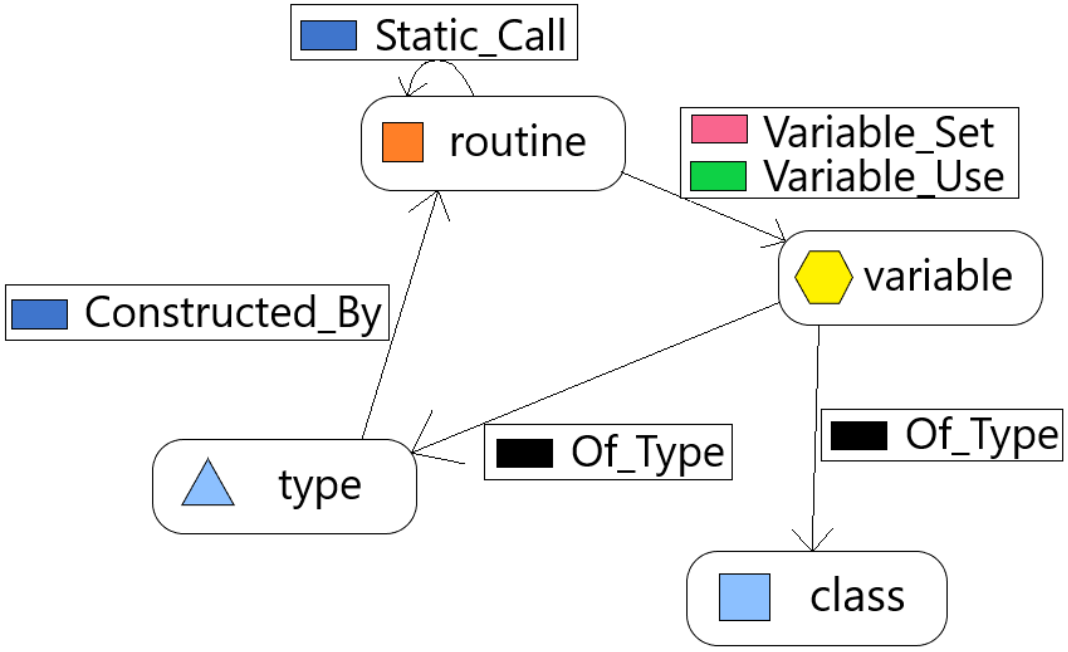

Abbildung 10: Das RFG-Schema von Declaration Facts und Code Facts

3.1.3.2 Declaration Call und Call

Die Views Declaration Call bzw. Call beinhalten jeweils einen Untergraph von Declaration Facts bzw. Code Facts. Da Declaration Facts und Code Facts den gleichen Graphen enthalten, sind auch Declaration Call und Call gleich. Sie bestehen aus Knoten vom Typ *Routine* und aus Kanten vom Typ *Static_Call* wie Abb. 11 darstellt.

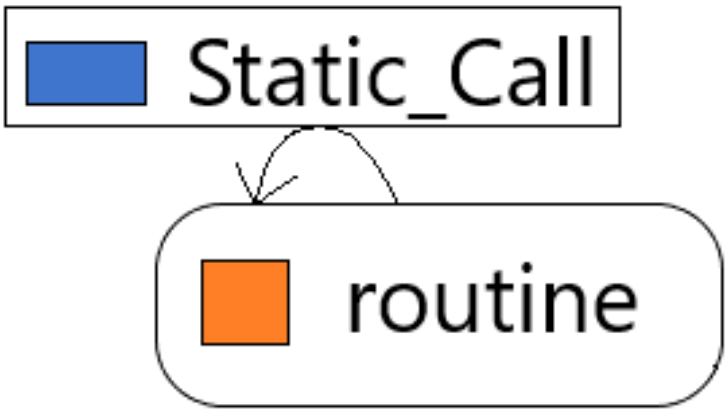

Abbildung 11: Das RFG-Schema von Declaration Call und Call

3.1.3.3 File und Module

File bzw. Module stellen eine hierarchische Sicht dar. Der RFG besteht aus *Directory* Knoten und einem *File* bzw. *Module* Knoten. Der *File* bzw. *Module* Knoten enthält einen Untergraph des Declaration Facts und Code Facts Views, der alle Knoten, aber keine Kanten enthält. Abb. 12 bzw. Abb. 13 illustrieren das RFG-Schema von File bzw. Module. Die Kanten die dort zu sehen sind, repräsentieren *Enclosing* Kanten.

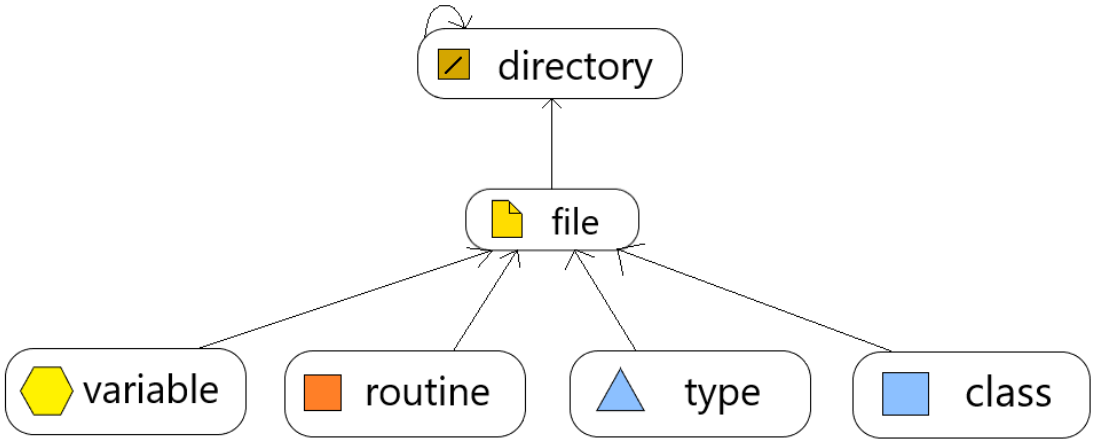

Abbildung 12: Das RFG-Schema von File

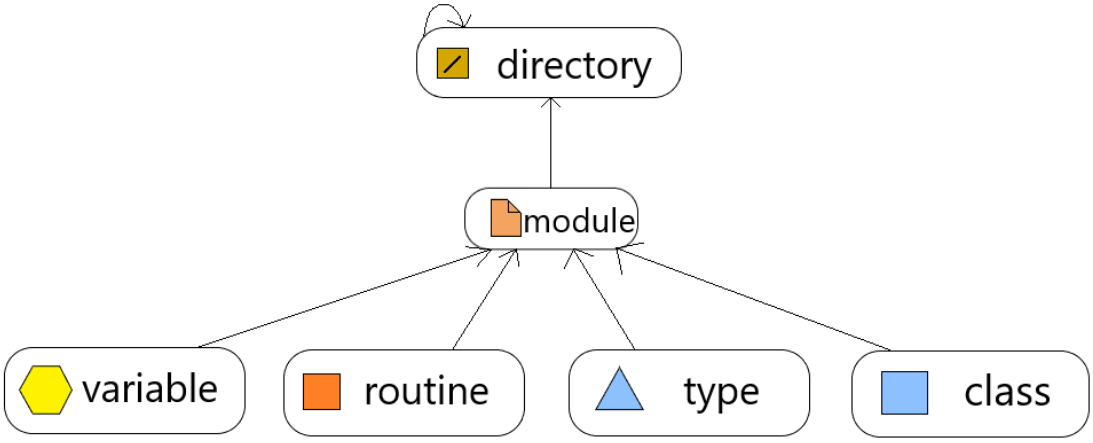

Abbildung 13: Das RFG-Schema von Module

3.1.3.4 Entries

Der Entries View beinhaltet lediglich den in 3.1.1.1 beschriebenen Einstiegsknoten. Ein RFG-Schema für diesen View zu erstellen ist überflüssig, da es nur aus einem Knoten bestehen würde.

3.2 Aufbau des Programms

Das hier implementierte Programm *js2rfg* ist in mehrere Abstraktionsebenen unterteilt. Das folgende Flussdiagramm in Abb. 14 stellt diesen Porgrammablauf dar.

Das Programm bekommt eine JavaScript Datei übergeben, welche es zunächst durch AntLR in einen AST übersetzt. Abschnitt 3.2.1 beschreibt diesen Vorgang genauer. Über den AST wird anschließend mit mehreren Visitorn traversiert. Die einzelnen Komponenten und Abhängigkeiten eines JavaScript Programms werden analysiert und in eine interne Zwischendarstellung abgespeichert. Zuletzt übersetzt der RFGCreator diese Zwischendarstellung in das gewünschte RFG-Format.

3.2.1 AST Generierung durch AntLR

AntLR stellt eine Sammlung von vorgefertigten Grammatiken für sämtliche Programmiersprachen[4] zur Verfügung, worunter auch eine Grammatik für JavaScript[5] zu finden ist, die

[4] `https://github.com/antlr/grammars-v4`
[5] `https://github.com/antlr/grammars-v4/tree/master/javascript/javascript`

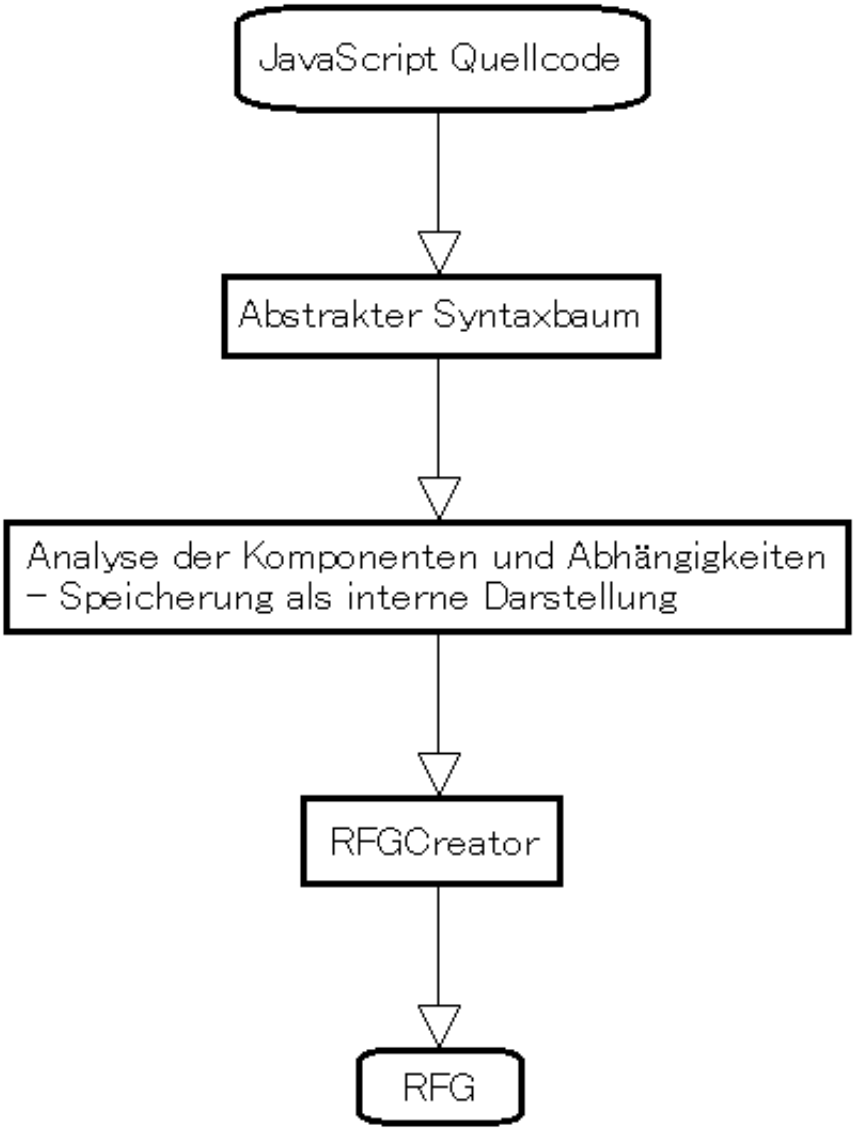

Abbildung 14: Abstraktionsebenen des Programms

von Positive Technologies[6] entwickelt wurde. Da eine eigene Grammatik zu entwickeln, den Rahmen dieser Bachelorarbeit überschreiten würde, basiert *js2rfg* auf der oben genannten Grammatik. Sie bildet die Sprache korrekt ab und erzeugt durch AntLR einen AST auf dem weitere Analysen statt finden können.

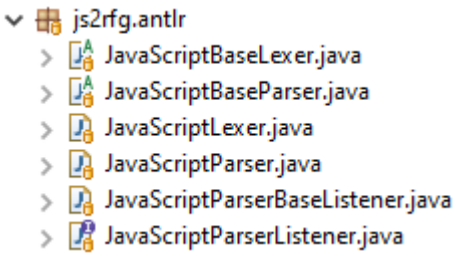

Abbildung 15: Generierte Java-Dateien von AntLR

AntLR generiert aus der Grammatik Java-Dateien. Diese sind an das Programm angebunden, wie Abb. 15 zeigt. Um einen eigenen Visitor zu erstellen, muss die Klasse `JavaScriptParserBaseListener.java` erweitert werden. Sie stellt für jeden Knoten des ASTs zwei Methoden bereit. Eine, die aufgerufen wird, wenn der Knoten betreten wird und noch eine, wenn der Knoten wieder verlassen wird. Die Methode beim Betreten beginnt mit `enter` und die beim Verlassen mit `exit`, gefolgt vom Knotennamen. Listing 13 zeigt ein Beispiel solch zweier Methoden. Der Parameter `VariableDeclarationContex ctx` repräsentiert den Knoten. Er bietet Verweise auf die Vorgänger- und Nachfolgerknoten, den Quellcode und sämtliche Informationen darüber, z. B. die Stelle (Zeile und Spalte) in der Quelldatei.

```
1  public class VariableListener extends JavaScriptParserBaseListener {
2
3    @Override
4    public void enterVariableDeclaration(VariableDeclarationContex ctx) {
5      ...
6    }
7
8    @Override
9    public void exitVariableDeclaration(VariableDeclarationContext ctx) {
10     ...
11   }
12 }
```

Listing 13: Methodenpaar eines Knotens

Listing 17 zeigt wie mit AntLR ein AST aus einer JavaScript Datei erzeugt wird. Wenn der Methode `parse(Path path)` ein Pfad zu einer JavaScript Datei übergeben wird, dann generiert sie den entsprechenden AST und gibt ihn als Rückgabewert zurück. In Zeile 3

[6] `https://www.ptsecurity.com/`

```
1  public ParseTree parse(Path path) {
2    try {
3      CharStream input = CharStreams.fromPath(path);
4      JavaScriptLexer lexer = new JavaScriptLexer(input);
5      TokenStream tokens = new CommonTokenStream(lexer);
6      JavaScriptParser parser = new JavaScriptParser(tokens);
7      return parser.program();
8    } catch (IOException e) {
9        e.printStackTrace();
10       return null;
11   }
12 }
```

Listing 14: Erzeugen eines ASTs über eine Eingabedatei

wird die Datei eingelesen. Darauf folgt in Zeile 4 eine lexikalische Analyse, die in Zeile 5 als ein *TokenStream* abgespeichert wird. Letztendlich folgt in Zeile 6 die syntaktische Analyse, die den AST erzeugt, der in Zeile 7 zurückgegeben wird.

```
1 ParseTree tree = parse(path);
2 ParseTreeWalker.DEFAULT.walk(new VariableListener(), tree);
```

Listing 15: Einen Visitor über den AST traversieren lassen

Listing 15 illustiert die Anwendung eines Visitors auf den AST. In Zeile 1 wird die Variable `tree` mit dem AST, der durch die Methode `parse(Path path)` erzeugt wird, initialisiert. Als nächstes wird in Zeile 2 der Visitor `VariableListener` auf den AST angewendet. Er traviersiert den AST und ruft auf jedem Knoten das entsprechende Methodenpaar auf.

3.2.2 Objektmodellierung der Sprachelemente und Abhängigkeiten

Um bei der Analyse des ASTs nicht direkt den RFG durch die Methoden der Bauhaus API[7] zu erzeugen, werden die gesammelten Informationen erst einmal in eine eigene Zwischendarstellung gespeichert und dann an den *RFGCreator*, der die Informationen dann letztendlich in das RFG-Format übersetzt, weitergegeben. Diese Zwischendarstellung ermöglicht eine leichtere Weiterverarbeitung der Knoten und Kanten und schafft eine Abstraktion zwischen AST Analyse und *RFGCreator*.

Die Objektmodellierung entspricht ungefähr der Form des RFGs. Es gibt zwei abstrakte Superklassen, `GraphNode` und `GraphEdge`, von denen alle Objekte erben. `GraphNode` entspricht den Knoten und `GraphEdge` entspricht den Kanten des RFGs. `GraphNode` bietet die Möglichkeit `identifier` (den Namen der Variable, Funktion, etc.), `line` und `column` (die Stelle (Zeile und Spalte) in der Quelldatei) zu speichern. `GraphEdge` bietet die Möglich-

[7] Programmierschnittstelle (engl. appplication pprogramming interface, kurz API)

keit `line` und `column` zu speichern, jedoch `identifier` nicht, da es sich bei `GraphEdge` um eine Abhängigkeit zweier Komponenten handelt, welche kein Sprachelement ist und deswegen keine Namen besitzt. `line` und `column` entsprechen der Stelle an der die Abhängigkeit auftritt. Listing 16 zeigt zum Beispiel die Verwendung einer Variable in einer Funktion. Dort würde es eine Abhängigkeit zwischen der Variable und der Funktion geben. Sie hat keinen Namen, aber die Stelle, wo diese Abhängigkeit auftritt, kann ermittelt und abgespeichert werden.

```
1 var a;
2
3 function test() {
4   a = 5;
5 }
```

Listing 16: Funktion die eine Variable verwendet

Außerdem hat `GraphEdge` jeweils einen Verweis auf seine inzidenten Knoten. Dies ist über eine generische Implementierung dieser Klasse gelöst. Listing 17 zeigt den Kopf der Klasse. `S` repräsentiert den Typ des Vorgängerknotens und `T` den Typ des Nachfolgerknotens.

```
1 public abstract class GraphEdge<S extends GraphNode, T extends GraphNode> {
2   ...
3 }
```

Listing 17: Generischer Kopf der Klasse `GraphEdge`

3.2.3 AST Analyse

Für jedes in 3.1 definierte Sprachelement und dessen Abhängigkeiten ist ein Visitor definiert, der die Vorkommen auf dem AST analysiert und als die in Abschnitt 3.2.2 beschriebene interne Darstellung zwischenspeichert. Darauf folgt ein Aufruf der entsprechenden Methode im *RFGCreator*, welche das Objekt als Parameter entgegen nimmt und in das RFG-Format übersetzt. Einen solchen Ablauf demonstriert Listing 18.

```
1  public class VariableListener extends JavaScriptParserBaseListener {
2
3      [...]
4
5      @Override
6      public void enterVariableDeclaration(VariableDeclarationContext ctx) {
7          if (isGlobal()) {
8              if (ctx.assignable().Identifier() != null) {
9                  Variable variable = new Variable();
10                 variable.setAttributes(ctx.assignable().Identifier().
       getSymbol());
11                 Application.RFG_CREATOR.addVariable(variable, GraphView.
       Code_Facts, GraphView.Declaration_Facts);
12             }
```

```
13            }
14        }
15    }
```

Listing 18: Implementierung der `VariableListener` Klasse

In Abb. 16 ist der AST von Listing 16 abgebildet. Dort ist zu erkennen, dass im linken Teilbaum das Nichtterminalsymbol *variableDeclaration* den Bezeichner der deklarierten Variable beinhaltet. Dieser Bezeichner wird für die Analyse benötigt und deshalb verwendet der `VariableListener` die `enter`-Methode des Knotens, um den Bezeichner und die Stelle (Zeile und Spalte) im Quellcode zu erlangen.

In Zeile 9 wird ein neues `Variable` Objekt instanziiert, welches in Zeile 10 die Informationen des Bezeichnersymbols übergeben bekommt und darauf in Zeile 11 dem *RFGCreator* übermittelt wird.

Der rechte Teilbaum des ASTs aus Abb. 16 bildet die Deklaration der Funktion `test()` ab. Das Vorkommen solcher Funktionsdeklarationen analysiert der `FunctionListener`. Weiter unten im Teilbaum ist die Zuweisung der Variable `a` zu sehen. Darum kümmert sich der `VariableSetListener`.

3.2.4 RFGCreator

Der *RFGCreator* ist die höchste Abstraktionsebene. Er übersetzt die interne Darstellung in das RFG-Format, wozu er die Bauhaus API benutzt. Er bietet die Möglichkeit Views anzulegen, Knoten und Kanten den verschiedenen Views hinzuzufügen und neue Knoten- und Kantentypen zu erstellen. Listing 19 demonstriert an dem Beispiel einer Variable wie ein Knoten dem RFG hinzugefügt wird.

```
1  private IGraph graph = GraphFactory.createGraph();
2
3  public void addVariable(Variable variable) {
4    INode node = graph.createNode(graph.getNodeTypeByName(variable.
       getNodeType()));
5    node.setLinkageName(variable.getLinkageName());
6    node.setName(variable.getIdentifier());
7    node.putInt(Schema.ATTR_SOURCE_LINE, variable.getLine());
8    node.putInt(Schema.ATTR_SOURCE_COLUMN, variable.getColumn());
9    graph.getViewByName("Code Facts").addNode(node);
10 }
```

Listing 19: Beispiel einer Übersetzung in das RFG-Format

Zeile 1 erstellt den Graphen, dem dann Views, Knoten oder Kanten hinzugefügt werden können. Der Knoten wird über die Methode in Zeile 3 bis 10 dem Graphen hinzugefügt, indem Zeile 4 einen leeren Knoten vom Typ *Variable* erstellt. Diesem Knoten können Attribute wie Name, Zeile, Spalte, etc. hinzugefügt werden. Dies passiert in Zeile 7-9.

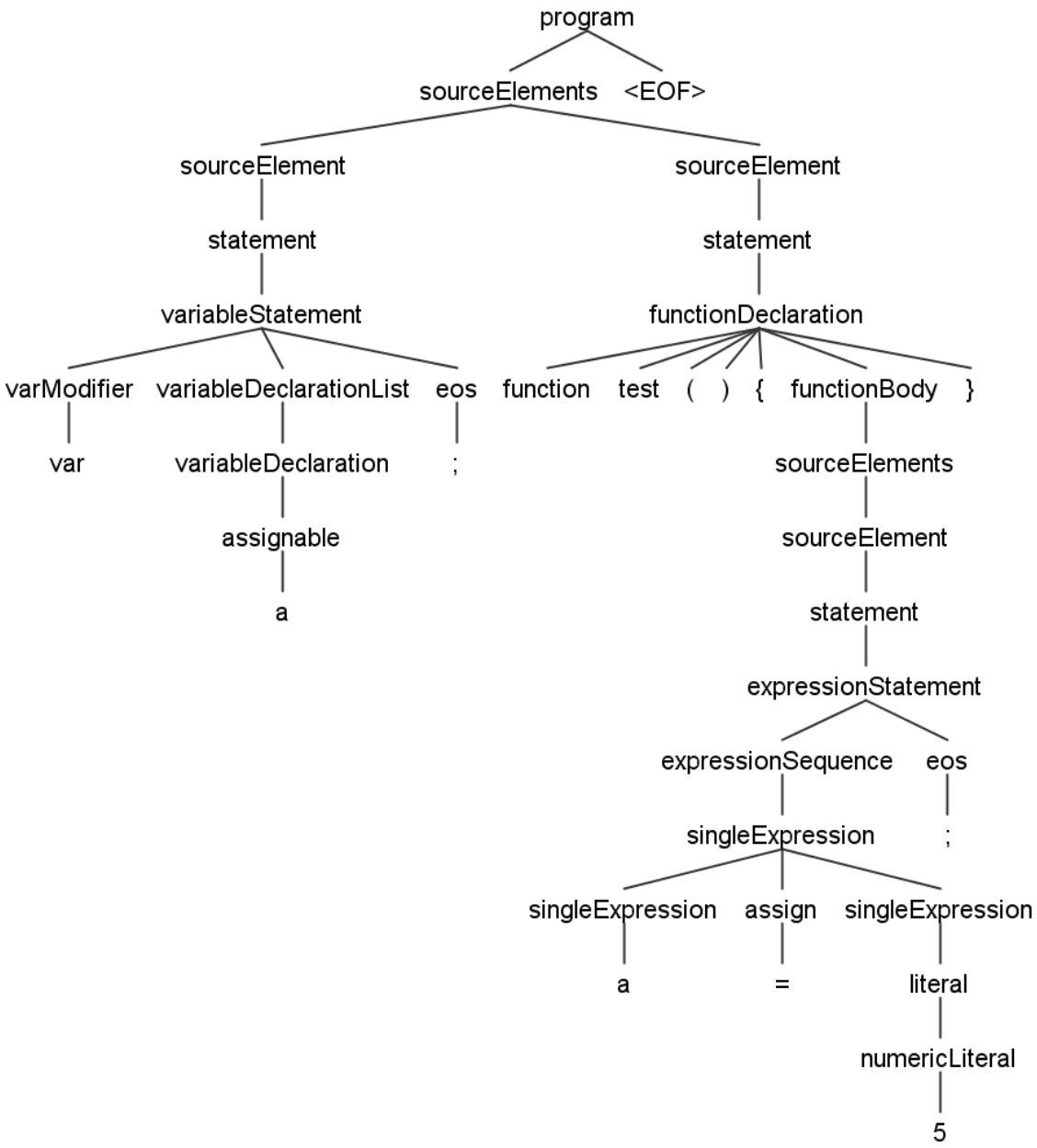

Abbildung 16: Abstrakter Syntaxbaum von Listing 16

Außerdem braucht der Knoten noch einen eindeutigen Namen (*LinkageName* genannt). Dieser wird in Zeile 6 festggelegt. Letztendlich wird der modifizierte Knoten in Zeile 10 in dem entsprechenden *View* gespeichert. Natürlich können dem Knoten weitere Attribute hinzugefügt werden. Diese werden in dieser Bachelorarbeit aber nicht weiter behandelt.

4 Ergebnisse

4.1 Funktionalität

Das Programm *js2rfg* bietet die Übersetzung aller in Abschnitt 3.1 vorgestellten Sprachelemente und Abhängigkeiten in das RFG-Format. *js2rfg* wurde so entwickelt, dass eine weiterführende Entwicklung des Programms einfach und mit wenig Aufwand verbunden ist. Es wurde viel Wert auf eine Abtrennung in Abstraktionsebenen gelegt, sodass das Programm überschaubar bleibt. Mit dem generierten RFG sind Analysen, wie Erkennung von totem Code, Zyklenerkennung und Architekturanalysen möglich. Jede dieser Analysen wird in Abschnitt 4.2 anhand eines Beispiels vorgestellt.

4.2 Analysen

4.2.1 Erkennung von totem Code

Das Erkennen von totem Code im RFG liegt dem Erreichbarkeitsproblem in der Graphentheorie[KN09] zu Grunde. Dieses Erreichbarkeitsproblem beschäftigt sich mit der Frage, ob in einem Graphen ein Weg von einem Startknoten zu jedem anderen Knoten des Graphs existiert. Existiert kein solcher Weg, gilt der Knoten als nicht erreichbar und entspricht totem Code. Der Knoten *.entry*, den es in jedem JavaScript RFG gibt, stellt den Startknoten dar. Dieser Knoten dient als Startknoten und die Analyse findet alle von ihm aus nicht erreichbaren Knoten.

```
1  greet();
2
3  function greet() {
4    return "Hello " + firstname();
5  }
6
7  function firstname() {
8    ...
9  }
10
11 function lastname() {
12   ...
13 }
```

Listing 20: Beispiel von totem Code

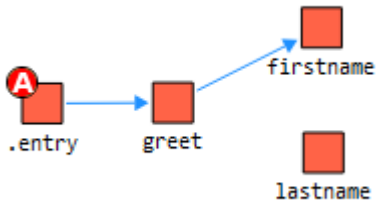

Abbildung 17: Der RFG von Listing 20

Listing 20 und Abb. 17 zeigen ein Beispiel, in dem *.entry* die Funktion `greet()` und `greet()` die Funktion `firstname()` aufruft. `lastname()` wird nicht aufgerufen. Im RFG (Abb. 17) ist zu erkennen, dass es vom *.entry* Knoten zu `greet()` und `firstname()` einen Weg gibt, aber nicht zu `lastname()`. Die Analyse würde diesen Knoten erkennen und ausgeben.

4.2.2 Zyklenerkennung

Die Zyklenerkennung lässt sich ebenfalls auf ein graphentheoretisches Problem[KN09] zurückführen, welches auf den RFG angewandt werden kann. In der Graphentheorie wird ein Graph nach einem Kreis abgesucht. Ein Kreis ist ein Weg der den selben Start- und Endknoten besitzt.

```
1  a();
2
3  function a() {
4    b();
5  }
6
7  function b() {
8    c();
9  }
10
11 function c() {
12   a();
13 }
```

Listing 21: Beispiel eines Zyklus

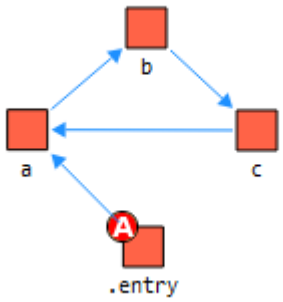

Abbildung 18: Der RFG von Listing 21

Listing 21 zeigt ein Beispiel eines Zyklus. Funktion `a` ruft Funktion `b` auf, Funktion `b` ruft Funktion `c` auf und Funktion `c` ruft erneut Funktion `a` auf. Der Weg `(a, b, c, a)` ist ein Kreis, wie Abb. 18 visualisiert. Die Zyklenanalyse würde solche Kreise finden und anzeigen.

4.2.3 Architekturanalyse

In der Architekturanalyse werden zwei Graphen, die Soll- und die Ist-Architektur, miteinander verglichen. Die Soll-Architektur kann in Gravis modelliert werden und dann mit dem erzeugten RFG eines JavaScript Programms verglichen werden.

Abbildung 19: Die Soll-Architektur

```
1  function login() {
2    password();
3    encode();
4  }
5
6  function password() {}
7
8  function encode() {}
```

Listing 22: Der Programmcode der Ist-Architektur

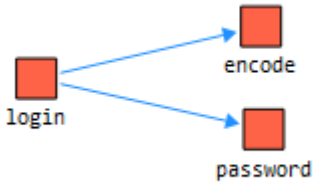

Abbildung 20: Der RFG der Ist-Architektur

In Abb. 19 ist die Soll-Architektur abgebildet und in Listing 22 und Abb. 20 ist die Ist-Architektur abgebildet. Bei der Architekturanalyse werden Abb. 19 und Abb. 20 miteinander verglichen und es wird geprüft, ob die Richtlinien der Architektur eingehalten werden. Bei diesem Beispiel ist dies nicht der Fall, da die Soll-Architektur festlegt, dass `login()` nur auf `password()` zugreifen darf. In der Ist-Architektur greift `login()` allerdings auf `password()` und `encode()` zu. Die Architekturanalyse würde in so einem Fall die Stelle des Architekturbruchs angeben.

5 Fazit

5.1 Ausblick

Obwohl *js2rfg* die wichtigsten Programmabläufe von JavaScript in den RFG übersetzt, unterstützt es keine JavaScript Programme, die aus mehreren Dateien aufgebaut sind, und keine Objekteigenschaften und -methoden.

Da JavaScript eine dynamisch typisierte Programmiersprache ist und daher eine statische Analyse problematischer ist als bei statisch typisierten Sprachen, wäre noch eine RFG-Erzeugung von TypeScript interessant. TypeScript[BAT14] ist eine Programmiersprache, die auf JavaScript aufbaut. Sie erweitert JavaScript durch Klassen, Schnittstellen (engl. Interface) und einem statisch typisierten System.

5.1.1 JavaScript Programm aus mehreren Dateien

Größere JavaScript Anwendungen bestehen mit hoher Wahrscheinlichkeit aus mehreren Dateien bzw. Modulen. Der jetzige Stand von *js2rfg* bietet allerdings nur die Möglichkeit eine einzige Datei einzulesen und zu analysieren.

5.1.2 Objekteigenschaften und -methoden

Objekte werden benutzt, um komplexere Informationen in eine Einheit zusammenzufassen. Sie enthalten Variablen und Funktionen, die zu den Eigenschaften des Objekts gehören. Zur Zeit werden diese Eigenschaften noch nicht im RFG abgebildet. Da einem Objekt zur Laufzeit Variablen oder Funktionen hinzugefügt oder entfernt werden können, wurden die Eigenschaften eines Objekts im RFG erst einmal unberücksichtigt gelassen.

5.2 Schlusswort

Die Entwicklung eines *JavaScript-zu-RFG* Übersetzers ist eine anspruchsvolle Aufgabe, da sich eine dynamische Programmiersprache schlecht auf eine statische Abbildung projezieren lässt. Viele Stärken von JavaScript, z. B. die dynamische Typisierung, lassen sich nicht auf eine statische Darstellung abbilden und es mussten viele gängigen Elemente eines RFG weggelassen werden. Dazu zählt zum Beispiel der Typ von *Return Values*. Bei der Darstellung der Typisierung einer Variablen (3.1.2.4) musste vorausgesetzt werden, dass diese Variable nicht für andere Zwecke weiter verwendet wird. Dennoch werden die wichtigsten Analysen für ein JavaScript Programm unterstützt.

Abbildungsverzeichnis

Literatur

[Axia] *Axivion – wir stoppen Software-Erosion*. `https://www.axivion.com/de`. `https://www.axivion.com/de`. – Eingesehen am 26.01.2020

[Axib] AXIVION GMBH: *Axivion Bauhaus Suite Documentation*. 7.0.0

[BAT14] BIERMAN, Gavin ; ABADI, Martín ; TORGERSEN, Mads: Understanding TypeScript. In: JONES, Richard (Hrsg.): *ECOOP 2014 – Object-Oriented Programming*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2014. – ISBN 978–3–662–44202–9, S. 257–281

[Bay13] BAYBULATOV, Emil: *Generierung eines globalen Abhängigkeitsgraphen für Java in Eclipse*. 2013

[Ber07] BERGER, Bernhard J.: *Klonmanagement: Klonerkennung für eingebettete Systeme*, Universität Bremen, Diploma Thesis, 2007

[BYM+98] BAXTER, I.D. ; YAHIN, A. ; MOURA, L. ; SANT'ANNA, M. ; BIER, L.: Clone detection using abstract syntax trees. In: *Software Maintenance, 1998. Proceedings., International Conference on*, 1998. – ISSN 1063–6773, 368-377

[Ecm19] ECMA INTERNATIONAL: *Standard ECMA-262 - ECMAScript 2019 Language Specification*. 10. 2019 `https://www.ecma-international.org/publications/files/ECMA-ST/ECMA-262.pdf`

[EKP+99] EISENBARTH, Thomas ; KOSCHKE, Rainer ; PLOEDEREDER, Erhard ; GIRARD, Jean-Francois ; WÜRTHNER, Martin: Projekt Bauhaus: Interaktive und inkrementelle Wiedergewinnung von SW-Architekturen. (1999), 05

[GMNR10] GOODMAN, Danny ; MORRISON, Michael ; NOVITSKI, Paul ; RAYL, Tia G.: *JavaScript Bible*. 7th. Wiley Publishing, 2010. – ISBN 0470526912

[KN09] KRUMKE, Sven O. ; NOLTEMEIER, Hartmut: *Graphentheoretische Konzepte und Algorithmen*. Springer DE, 2009

[Par13] PARR, Terence: *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2013

[RVP06] RAZA, Aoun ; VOGEL, Gunther ; PLOEDEREDER, Erhard: Bauhaus - A Tool Suite for Program Analysis and Reverse Engineering, 2006, S. 71–82