Universität Leipzig

Wirtschaftswissenschaftliche Fakultät

Institut für Wirtschaftsinformatik

Professur für Wirtschaftsinformatik, insbesondere Softwareentwicklung für Wirtschaft und Verwaltung

Prof. Dr. Ulrich W. Eisenecker

David Baum, M. Sc.

Thema

Visualisierung von Variabilität in C-Quellcode

Masterarbeit zur Erlangung des akademischen Grades
Master of Science – Wirtschaftsinformatik

vorgelegt von: Christina Sixtus

Email-Adresse: wir13dvx@studserv.uni-leipzig.de

Leipzig, den 25. Juni 2019

Abstract

In C-Quellcode wird der C-Präprozessor häufig verwendet, um ein Softwaresystem für verschiedene Ausführungsumgebungen und Varianten zu konfigurieren. Anweisungen zur bedingten Kompilierung ermöglichen es, dass Quellcodeteile bei der Verarbeitung durch den Präprozessor ein- oder ausgeblendet werden. Dies erzeugt verschiedene Varianten der Software, erschwert jedoch die Lesbarkeit und Wartung des Quellcodes. Insbesondere die Auswirkungen einzelner Makrodefinitionen sind oft nicht einfach zu ermitteln. In dieser Arbeit soll der Frage nachgegangen werden, wie das Verständnis des Quellcodes und der Auswirkungen von Makrodefinitionen mithilfe von Softwarevisualisierung unterstützt werden kann. Dazu wird eine bestehende Visualisierungsmetapher an den Anwendungsfall angepasst. Anschließend folgt der Entwurf eines Verarbeitungsprozesses, um den Quellcode automatisiert darstellen zu können. Mithilfe eines Prototyps wird die Machbarkeit gezeigt.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Abkürzungsverzeichnis

ANSI	American National Standard Institute
API	Application Programming Interface
ASG	Abstract Syntax Graph
AST	Abstract Syntax Tree
CPP	C-Präprozessor
JVM	Java Virtual Machine
LIFO	Last-in-First-out
MPS	Meta Programming System
RD	Recursive Disk
XML	Extensible Markup Language

1 Einleitung

1.1 Motivation

Die Programmiersprache C bietet mit ihrem Präprozessor ein häufig genutztes Instrument, um Software für unterschiedliche Plattformen und mit unterschiedlichen Eigenschaften zu konfigurieren. Dies wird vor allem durch bedingte Kompilierung erreicht, ein Verfahren, um Quellcode vor dem Kompilieren der Software zu konfigurieren. Durch die bedingte Kompilierung werden Elemente des Quellcodes ein- oder ausgeschlossen, abhängig von zuvor definierten Bedingungen. Der C-Präprozessor gesteht dem Entwickler hier große Freiheiten zu. Dieser kann beispielsweise beliebige Elemente unter eine Bedingung setzen und Bedingungen schachteln. Zudem können die Anweisungen der Präprozessorsprache auch parametrisiert werden und komplizierte Quelltextfragmente ersetzen. Dies alles trägt dazu bei, dass C-Quellcode bei umfangreichem Einsatz des Präprozessors nur schwer les- und wartbar ist (vgl. [Spencer/Collyer 1992, 1]). In Listing 1.1 ist ein Quellcodefragment aus der Software cURL[1], welche in C geschrieben ist, zu sehen. Zu erkennen ist der Einsatz des Präprozessors, um verschiedene Konfigurationsmöglichkeiten zu verarbeiten. Besonders ist hier auf die geschachtelte Definition von Makros hinzuweisen, beispielsweise in Zeile 3 und das Struct ab Zeile 11, welches nur deklariert wird, wenn die Bedingung erfüllt ist.

```
1  #if defined(__DragonFly__) || defined(HAVE_WINSOCK_H)
2  /* DragonFlyBSD and Windows use millisecond units */
3  #define KEEPALIVE_FACTOR(x) (x *= 1000)
4  #else
5  #define KEEPALIVE_FACTOR(x)
6  #endif
7
8  #if defined(HAVE_WINSOCK2_H) && !defined(SIO_KEEPALIVE_VALS)
9  #define SIO_KEEPALIVE_VALS    _WSAIOW(IOC_VENDOR,4)
10
11 struct tcp_keepalive {
12 u_long onoff;
13 u_long keepalivetime;
14 u_long keepaliveinterval;
15 };
16 #endif
```

Listing 1.1: Ausschnitt aus der Datei connect.c des cURL-Projektes

[1] https://curl.haxx.se/

Softwarevisualisierung dient dazu, das Verständnis von Softwaresystemen und die Produktivität der Softwareentwicklung zu verbessern. Hierzu bietet sie grafische Repräsentationen für die Struktur, das Verhalten und die Entwicklung von Software (vgl. [Diehl 2007]).

In dieser Arbeit soll untersucht werden, wie Variabilität von C-Quellcode, welche durch bedingte Kompilierung erzeugt wird, als Teil einer Strukturvisualisierung dargestellt und damit das Verständnis des Softwaresystems erhöht werden kann.

1.2 Zielsetzung

Um das zuvor definierte Thema zu bearbeiten, wird die Forschungsmethode des Prototypings (vgl. [Wilde/Hess 2007]) gewählt. Dazu sollen zunächst die theoretischen Grundlagen erarbeitet werden, unter anderem, welche Varianten der Variabilität in C-Quellcode existieren und welche Formen die bedingte Kompilierung annimmt. Anschließend werden verschiedene Werkzeuge geprüft und zu einer Werkzeugkette zusammengesetzt. Dieser Schritt wird davon beeinflusst, dass die vorliegende Arbeit Teil eines größeren Forschungsprojektes zur Softwarevisualisierung ist und der Prototyp in die dort entwickelte Visualisierungsplattform eingebettet werden soll. Die praktische Implementierung dient der Überprüfung der Machbarkeit des beschriebenen Konzepts. Eine ausführliche Evaluation, die Teil der Forschungsmethode ist, kann aufgrund des Umfangs der Arbeit nur theoretisch skizziert werden und muss dann in der Folge durchgeführt werden.

1.3 Vorgehensweise

Die Vorgehensweise soll folgendermaßen aussehen: Der nächste Abschnitt erläutert kurz bestehende Ansätze zur Analyse und Visualisierung von C-Quellcode und bedingter Kompilierung. Dazu wurde eine einleitende Literaturrecherche durchgeführt.

Kapitel 2 beschäftigt sich mit den notwendigen Grundlagen, um das Thema zu bearbeiten. Zunächst werden die Programmiersprache C sowie der C-Präprozessor näher beschrieben. Außerdem werden unterschiedliche Möglichkeiten zur Implementierung von Variabilität, insbesondere die bedingte Kompilierung, betrachtet. Der zweite Schwerpunkt in diesem Kapitel ist die Softwarevisualisierung, vor allem ihre Ziele und die Einordnung des hier beschriebenen Prototyps im Bereich der Softwarevisualisierung. Zudem wird Getaviz, eine Zusammenstellung von Werkzeugen im Bereich der Softwarevisualisierung, beschrieben. Der Visualisierungsgenerator von Getaviz soll verwendet werden, um mithilfe verschiedener Metaphern Quellcode darzustellen. Um Details der Implementierung besser nachvollziehen zu können, folgt eine kurze Einführung in Graphdatenbanken, insbesondere Neo4j.

Im darauffolgenden Kapitel wird die Konzeptionierung des Prototyps beschrieben. Zunächst stellt sich hier die Frage, welche Informationen aus dem Quellcode für die Visualisierung benötigt und welche Arten der Variabilität wie dargestellt werden. Als Nächstes wird untersucht, wie die Variabilität aus dem Quellcode korrekt extrahiert und der Quellcode geparst werden kann. Hier findet sich ein Vergleich zwischen verschiedenen Ansätzen und

Werkzeugen. Da die Informationen über den Quelltext in einem Graphmodell abgelegt werden sollen, muss dieses im nächsten Schritt entworfen werden. Anschließend widmet sich die Arbeit der Visualisierung. Nach einer Beschreibung des Anwendungsfalls, dem die hier entworfene Visualisierung dient, folgt eine Anpassung der bestehenden Visualisierungsmetaphern an den Anwendungsfall und die extrahierten Informationen. Zudem wird eine Erweiterung der Benutzungsoberfläche konzipiert, damit der Nutzer die Variabilität interaktiv erkunden kann.

Nach diesem Entwurf folgt in Kapitel 4 die Beschreibung der Implementierung. Hier wird entsprechend der Verarbeitungsreihenfolge zunächst das Parsen des Quellcodes beschrieben. Anschließend folgt die Abbildung in einer Graphdatenbank durch ein geeignetes Werkzeug. Die Anpassung des bestehenden Visualisierungsgenerators und der Nutzeroberfläche wird im nächsten Abschnitt beschrieben. Der letzte Teil dieses Kapitels beschäftigt sich mit automatisierten Tests und dem Entwurf einer Evaluation des Prototyps.

Das letzte Kapitel fasst die Ergebnisse dieser Arbeit zusammen, würdigt sie kritisch und gibt einen Ausblick, welche Fragen offen geblieben sind und wie das Thema weiter bearbeitet werden kann.

1.4 Stand der Forschung

Im folgenden Abschnitt werden verwandte Forschungsarbeiten betrachtet und mit dem hier vorgestellten Ansatz verglichen. Dabei steht die Visualisierung von C-Quellcode im Vordergrund, insbesondere die Visualisierung von Präprozessoranweisungen. Anschließend sollen als angrenzende Gebiete einige Arbeiten zum Refactoring und Verständnis von C-Quellcode beschrieben werden. Da mit C-Präprozessordirektiven häufig auch Softwareproduktlinien mit Variabilitätspunkten (engl. *features*) beschrieben werden, werden auch in diesem Bereich wichtige Arbeiten zur Visualisierung erläutert.

Die vorliegende Arbeit baut auf Getaviz auf, einer Zusammenstellung von Softwarevisualisierungswerkzeugen, welche unter anderem einen Visualisierungsgenerator für 2D- und 3D-Modelle enthält. Baum et al. [2017, 5] nennen in ihrer Arbeit einige vergleichbare Arbeiten mit ähnlichen Visualisierungen in 2D und 3D. Allerdings konzentrieren sich diese Publikationen nicht primär auf die Darstellung von C-Quellcode und vor allem nicht auf den Einsatz von Präprozessoren.

Im Bereich der Darstellung von Kontrollfluss-, Aktivitäts- und Aufrufdiagrammen für C-Quellcode finden sich zahlreiche Arbeiten, die häufig mehrere Visualisierungen anbieten. Beispielsweise ist Kayrebt ein Werkzeugkasten, der präkompilierten C-Code als Aktivitätsdiagramm darstellt (vgl. [Georget et al. 2015]). Telea et al. [2009] erstellen automatisiert einen Aufrufgraphen aus C- und C++-Quellcode, Bohnet und Dollner [2007] stellen diesen in 2D und 2½D dar und bieten mit ihrem Werkzeug die Möglichkeit, den Graphen schrittweise zu durchwandern und Details zu betrachten. Matsumara et al. [2009] sowie Sundararaman und Back [2008] haben einen Debugger für C-Code erstellt, der während des Debuggens grafisch die Struktur des Programms darstellt. Auch CLAS ist ein Werkzeug, um einen Aufruf-

und Kontrollflussgraphen des Programms zu erzeugen (vgl. [Bhattacharjee et al. 1994]). Linos et al. [1993] bieten verschiedene Visualisierungen für den Datenfluss- und den Kontrollflussgraphen. Mit CANTO (vgl. [Antoniol et al. 1997]) wurde eine Umgebung geschaffen, die das Programmverständnis durch Darstellungen auf verschiedenen Granularitätsebenen unterstützen soll: verschiedene Module erlauben die Quellcodebearbeitung, zahlreiche Analysen von Abhängigkeiten im Code sowie grafische Darstellungen, unter anderem in Form von Aufrufgraphen und Modulabhängigkeiten. Reniers et al. [2014] haben ein Werkzeug erstellt, das Codemetriken visualisiert, vor allem die Komplexität von Programmteilen. Cserép und Krupp [2014] hingegen schlagen verschiedene Grafiken vor, die Beziehungen zwischen Dateien in einem Softwaresystem zeigen. All diese Systeme visualisieren die Struktur und das Verhalten von C-Quellcode, allerdings analysieren sie nur präkompilierten Quellcode. Präprozessoranweisungen und bedingte Kompilierung spielen in den Darstellungen keine Rolle.

Anders gehen die Autoren der folgenden Publikationen vor: Livadas und Small [1994] haben eine Entwicklungsumgebung erstellt, die beim Klick auf ein Makro anzeigt, wo ein Makro definiert wurde und wie die Expansion des Makros aussieht. Auch Kullbach und Riediger [2001] visualisieren Makroexpansionen und bedingte Kompilierung in einer integrierten Entwicklungsumgebung durch sogenanntes *folding*: durch einen Klick auf Präprozessoranweisungen, wird der präkompilierte Quellcode ausgeklappt.

Im Bereich des Refactorings von C-Code unter Berücksichtigung von Präprozessoranweisungen finden sich zahlreiche Arbeiten, unter anderem [Garrido/Johnson 2005], [Spinellis 2010], [M. Vittek 2003], [Waddington/Yao 2005] und [Baxter/Mehlich 2001]. Badros und Notkin [2000] haben ein Werkzeug geschrieben, mit dem nicht präkompilierter C-Quellcode durch einfache Skripte analysiert werden kann.

Kuiter et al. [2018] haben PCLocator entwickelt, ein Werkzeug, welches für eine gegebene Stelle im Quellcode eine valide Konfiguration findet. Dabei nutzen die Autoren TypeChef und zwei weitere bestehende Softwaresysteme, um eine statische Quellcodeanalyse durchzuführen. Im Bereich der Linux-Konfigurationsanalyse gibt es einige Forschungsarbeiten, jedoch haben sie einen engen Fokus auf die Analyse von Konfigurationen. Die vorliegende Arbeit legt den Schwerpunkt jedoch nicht auf die Extraktion der Informationen, sondern auf die metapherbasierte Visualisierung derselben.

Der Bereich Softwareproduktlinien bietet einige Werkzeuge zur Visualisierung und zum besseren Verständnis von Variabilitätspunkten und Varianten. Beispielsweise bieten zwei Eclipse-Plugins Visualisierungen von Featuremodellen, Typprüfung von Präprozessorcode und vieles mehr (vgl. [Medeiros et al. 2013; Thüm et al. 2014]). Durch die Verwendung des Meta Programming System (MPS)[2] können Behringer et al. [2017] Entwicklerinnen verschiedene Ansichten zum Bearbeiten und Verstehen des Quellcodes von Softwareproduktlinien anbieten.

[2] https://www.jetbrains.com/mps/

Andere Produkte unterstützen allgemein die Entwicklung von Produktlinien, ohne speziellen Fokus auf C-Quellcode: Feigenspan et al. [2010] haben ein Eclipse-Plugin erstellt, das Code-Highlighting von Feature-Code ermöglicht. Nestor et al. [2008] haben Visualisierungen für die Konfiguration von Softwareproduktlinien erstellt, sie bieten allerdings keine direkte Verbindung zum Quellcode. Der *Feature Relation Graph* von Martinez et al. [2014] stellt mögliche Merkmalskombinationen in Abhängigkeit von einem ausgewählten Merkmal dar. Auch Illescas et al. [2016] zeigen verschiedene Visualisierungsmodelle für Merkmalskombinationen, aber auch hier findet sich keine Verbindung zum Quellcode, ebenso nicht bei Urli et al. [2015]. Im Bereich der Softwareproduktlinien fokussiert sich die Forschung auf die Darstellung von Variabilitätspunkten und zulässigen Kombinationen. Hier findet aber oft keine Verknüpfung mit dem zugrundeliegenden Quellcode statt.

Insgesamt lässt sich festhalten, dass eine Visualisierung, die mit der in dieser Arbeit vorgestellten Visualisierung vergleichbar ist, bisher nicht existiert. Es gibt zahlreiche metapherbasierte 2D- und 3D-Strukturmodelle von Quellcode. Zudem lassen sich einige Plugins für Entwicklungsumgebungen ausmachen, die häufig Kontroll- und Datenflüsse darstellen oder ein Refactoring von C-Quellcode erleichtern. Allerdings fanden sich nur zwei Visualisierungen, welche bedingte Kompilierung und andere Präprozessoranweisungen explizit darstellen. Die meisten dieser Werkzeuge konzentrieren sich auf eine Variante des Quellcodes nach der Verarbeitung durch den Präprozessor.

Im Bereich der Softwareproduktlinien beschäftigen sich einige Publikationen mit der Visualisierung von Merkmalen und Varianten sowie der ihnen innewohnenden Variabilität. Hier liegt der Fokus auf der Konfiguration und Visualisierung von zulässigen Merkmalskombinationen, wobei nur wenige Arbeiten diese grafischen Darstellungen mit dem Quellcode verknüpfen. Die in dieser Arbeit entwickelte Verbindung von einer Strukturvisualisierung auf Basis einer abstrakten Metapher mit einer Darstellung der bedingten Kompilierung ist aber nicht zu finden.

2 Grundlagen

Dieses Kapitel beschreibt für diese Arbeit notwendige Grundlagen. Zunächst wird die Programmiersprache C beschrieben, wobei für die Visualisierung wichtige Sprachmittel und die Funktionsweise des Präprozessors im Vordergrund stehen. Anschließend folgt eine Einführung in die Softwarevisualisierung, ihre Definition und ihre Ziele. Um die Nutzung des Visualisierungswerkzeugs Getaviz in dieser Arbeit besser zu verstehen, gibt es hierzu eine kurze Erläuterung. Zuletzt sollen Graphdatenbanken beschrieben werden, speziell das Graphdatenbanksystem Neo4j, welches im hier entwickelten Prototypen verwendet wird.

2.1 Die Programmiersprache C

Die Programmiersprache C wurde 1972 von Dennis Ritchie veröffentlicht und war ursprünglich für die Entwicklung des Betriebssystems UNIX gedacht. Der folgende Abschnitt stellt ihre Eigenschaften und Sprachelemente kurz vor, wobei näher auf den C-Präprozessor (CPP) sowie den Ablauf der Kompilierung eingegangen wird. Dies dient zum besseren Verständnis der entwickelten Visualisierung.

2.1.1 Eigenschaften

Henning/Vogelsang [2007, 13] klassifizieren Programmiersprachen nach drei Merkmalen: nach dem Programmierparadigma, nach dem Abstraktionsgrad von der zugrundeliegenden Prozessorarchitektur und nach dem Ausführungsschema. Diese Einordnung soll hier für C erläutert werden.

C ist dem imperativen Programmierparadigma zuzurechnen, da – im Gegensatz zu deklarativen Programmiersprachen – Anweisungen streng sequenziell verarbeitet werden. Außerdem lassen sich Anweisungen zur besseren Wiederverwendbarkeit in Funktionen kapseln. Deshalb ist C auch dem prozeduralen Programmierparadigma zuzurechnen, welches eine Unterkategorie des imperativen Paradigmas darstellt (vgl. [Henning/Vogelsang 2007, 13]). Als höhere Programmiersprache erfolgt die Einordnung von C bezüglich des Abstraktionsgrades von der Prozessorarchitektur über Maschinensprache und Assemblersprache, da sie von Prozessoren nicht ohne Übersetzung verstanden wird und Datentypen wie Strukturen und Arrays sowie Funktionen besitzt. Andererseits abstrahiert C weniger von dem ausführenden Prozessor als objektorientierte oder deklarative Sprachen. Bezüglich des Ausführungsschemas ist die Sprache den kompilierenden Programmiersprachen zuzuordnen, das heißt, der Quelltext wird von einem Compiler vor der Ausführung in Maschinencode übersetzt. Dies führt zu einer hohen Ausführungsgeschwindigkeit der Programme.

Weitere wichtige Eigenschaften, die Henning/Vogelsang [2007, 60] nennen, sind die statische Typisierung, statisch gebundene Funktionen und Unsicherheit. Statische Typisierung bedeutet, dass C Datentypen besitzt, welche zur Übersetzungszeit geprüft werden. Dadurch sollen Laufzeitfehler vermieden werden. Bei statisch gebundenen Funktionen werden aufzu-

rufende Funktionen beim Linken, einer Phase der Kompilierung, ermittelt. Bei dynamisch gebundenen Funktionen hingegen wird die genaue Version einer Methode, abhängig vom dynamischen Typ eines Objekts, erst zur Laufzeit des Programms bestimmt. Die Autoren bezeichnen C als unsicher, da direkte Speicherzugriffe möglich sind, was ungewollte Speichermanipulationen ermöglicht. Aufgrund der Tatsache, dass der Kernel aller wichtigen Betriebssysteme zum Teil in C geschrieben sind, stellt Wolf [2019, 23] fest, dass sich C besonders für die Systemprogrammierung eignet. Zudem sei es sehr verbreitet im Bereich der Embedded Systems und der Mikrocontroller aufgrund der vergleichsweise kleinen Programme.

Der Quelltext von C wird in zwei Arten von Dateien gespeichert: Headerdateien (Dateiendung .h) und Implementierungsdateien (Dateiendung .c). In Headerdateien werden Funktionen, Typen, Konstanten, externe Variablen und Makros deklariert, wodurch sie später zu einer Bibliothek kompiliert und in anderen Programmen wiederverwendet werden können. Eine Implementierungsdatei enthält, wie der Name andeutet, die Implementierung eines Programms und der Elemente, die in der dazugehörigen Headerdatei deklariert wurden. Die Kompilierung dieser Dateien wird in Abschnitt 2.1.4 erklärt.

2.1.2 Wichtige Sprachelemente

Die Sprache C wurde 1989 durch ein Komitee des American National Standard Institute (ANSI) standardisiert. Seitdem sind einige Revisionen veröffentlicht worden, die letzte zum Zeitpunkt der Abfassung dieser Arbeit in 2017. Auf diese Revision stützt sich auch die Analyse der für die Visualisierung benötigten Sprachbestandteile in Kapitel 3. In der folgenden Beschreibung soll nicht der gesamte Standard erläutert werden, sondern nur einige zum Verständnis der Arbeit und für die Visualisierung bedeutende Sprachbestandteile. Die Informationen stammen von Wolf [2019].

C ist eine statisch typisierte Sprache. Als Basisdatentypen stehen zur Verfügung: `int`, `long`, `short`, die Gleitpunkttypen `float` und `double`, sowie der Datentyp `char`, um sowohl einzelne Zeichen als auch kleine Zahlen zu verarbeiten. Mit dem Datentyp `wchar_t` können beliebige landesspezifische Zeichensätze dargestellt werden. Der C-99-Standard hat zusätzlich den Datentyp `_Bool` eingeführt, mit welchem Wahrheitswerte gespeichert werden können. Zudem gibt es den Typen `void`, der allerdings kein echter Datentyp ist und zum Beispiel eine Funktion ohne Rückgabewert kennzeichnet (vgl. [Wolf 2019, 67ff]). Datentypen können aus mehreren Schlüsselwörtern, beispielsweise `long long`, gebildet, sowie die Schlüsselwörter `signed` und `unsigned` davorgestellt werden, um vorzeichenbehaftete und vorzeichenlose Zahlen zu kennzeichnen. Wolf listet in einer Tabelle alle Standard-Datentypen in C auf (vgl. [Wolf 2019, 97]).

C bietet drei Arten von Kontrollstrukturen: Verzweigungen, Schleifen und Sprünge. Verzweigungen können mit `if-else`- oder `switch`-Anweisungen abgebildet werden. Dem Programmierer stehen drei Arten von Schleifen zur Verfügung: `while`-Schleifen, `do-while`-Schleifen und `for`-Schleifen. Es gibt vier Arten von kontrollierten Sprüngen,

`continue`, `break`, `exit` und `return`. Außerdem sind `goto`-Anweisungen im Sprachstandard definiert, von deren Verwendung allerdings abgeraten wird (vgl. [Wolf 2019, 137ff]).

Als prozedurale Sprache besitzt C *Funktionen*, um die Wiederverwendbarkeit und Modularisierung von Code zu unterstützen. Die `main()`-Funktion dient als Einstiegspunkt in ein Programm und muss in jedem C-Programm vorhanden sein. Funktionen können jeden beliebigen Datentyp zurückgeben und ihre Parameter können per *call-by-value* oder per *call-by-reference* übergeben werden (vgl. [Wolf 2019, 177ff]).

Zeiger sind Variablen, die eine Speicheradresse als Wert enthalten. Dadurch lassen sich Datenobjekte direkt als Parameter an Funktionen übergeben und dort modifizieren. Dieses Verfahren wird *call-by-reference* genannt. Außerdem können auf diese Weise auch Funktionen als Parameter an andere Funktionen übergeben werden. Der Typ eines Zeigers entspricht dem Typ der Variable, auf die ein Zeiger zeigt. Zeiger vom Datentyp `void` können auf einen beliebigen Typ zeigen. Die Syntax einer Zeigerdeklaration ist in Listing 2.1 zu sehen (vgl. [Wolf 2019, 300ff]).

```
1 Datentyp *zeigervariable;
```

Listing 2.1: Deklaration eines Zeigers in C

Um mehrere zusammengehörige Variablen gemeinsam zu deklarieren und zu nutzen, gibt es in C *Strukturen*. Diese können Variablen verschiedenen Datentyps enthalten. Die Deklaration einer Struktur erfolgt mit Angabe des Schlüsselwortes `struct` und des Typs der Struktur, der Zugriff auf einzelne Elemente der Struktur ist durch den Punktoperator (.) möglich (vgl. [Wolf 2019, 399ff]).

Unions sind Datenstrukturen, die ebenfalls mehrere Variablen kapseln. Allerdings wird hier immer nur eine der Variablen initialisiert. Dadurch können alternative Datentypen dargestellt werden (vgl. [Wolf 2019, 431ff]).

Mit *Enums* lassen sich zusammengehörige Konstanten deklarieren. Diese Funktionalität könnte auch mit Makros des C-Präprozessors abgebildet werden, allerdings bieten Enums einige Vorteile, zum Beispiel eine Debuggerunterstützung (vgl. [Wolf 2019, 436ff]).

Viele C-Programme enthalten Anweisungen, die mit # beginnen. Diese Anweisungen dienen der Steuerung des C-Präprozessors. Im folgenden Abschnitt sollen der Präprozessor und wichtige Anweisungen vorgestellt werden, da sie in dieser Arbeit von besonderer Bedeutung sind.

2.1.3 Der C-Präprozessor

Ein Präprozessor bereitet Eingabedaten vor, um sie anschließend zur Verarbeitung an ein anderes Computerprogramm weiterzureichen. Der CPP wird vor dem Kompilieren ausgeführt, um Teile zum Quellcode hinzuzufügen, zu ersetzen oder zu entfernen. Er ist ein lexikalischer Präprozessor, das heißt, er führt reine Textersetzungen durch, unabhängig

von der Syntax der C-Programme, in denen er verwendet wird (vgl. [Kästner et al. 2011, 1f]). Folgende Arbeiten werden unter anderem von ihm vorgenommen: Er kopiert Header- und Quelltextdateien in den Quelltext, er entfernt Kommentare aus dem Quelltext, er ersetzt Makroanweisungen durch ihren Inhalt und entfernt Teile des Quelltextes, abhängig von Bedingungen (vgl. [Wolf 2019, 221]). Die C-Präprozessoranweisungen bilden eine eigene, nicht Turing-vollständige Sprache (vgl. [Eisenecker 2008, 148]). Auf einige Präprozessoranweisungen (auch *Präprozessordirektiven* genannt) soll hier näher eingegangen werden.

Die `#include`-Anweisung dient dazu, Headerdateien in den Quellcode einzubinden, um dort definierte Funktionen und andere Elemente nutzen zu können. Der Präprozessor ersetzt diese Anweisung durch den Inhalt der Datei.

Mit der Anweisung `#define` werden sogenannte *Flags* und *Makros* definiert. Flags werden auch *symbolische Konstanten* genannt und sind reine Bezeichner, die dem Präprozessor bekannt sind und häufig bei der bedingten Kompilierung zum Einsatz kommen. Makros sind Bezeichner für ein Quelltextfragment und werden bei der sogenannten Makroexpansion, die der Präprozessor durchführt, durch dieses Fragment ersetzt. Das Substitut muss kein vollständiges C-Sprachmittel sein, es kann auch nur einzelne Zeichen oder Teile von Sprachmitteln enthalten. Des Weiteren können Makros auch parametrisiert sein, dann werden sie als *Funktionsmakros* bezeichnet. In Listing 2.2 sind ein einfaches Makros und ein Funktionsmakro dargestellt. Beim Aufruf des Präprozessors wird eine Verwendung des Funktionsmakros durch die Anweisung in Zeile 4 ersetzt, wobei die formalen Parameter `zahl1` und `zahl2` durch die übergebenen Argumente ersetzt werden. Die zusätzliche Klammerung der Parameter ist notwendig, da die übergebenen Argumente auch aus komplexen Ausdrücken bestehen können. Wird dann eine Textersetzung durchgeführt, kann die korrekte Auswertung der Funktion nicht gewährleistet werden (vgl. [Wolf 2019, 233ff; Eisenecker 2008, 129ff]).

```
1 #define EINS 1
2
3 #define SUMME(zahl1, zahl2) \
4  ((zahl1) + (zahl2))
```

Listing 2.2: Zwei Beispiele für Makros in C

Eine wichtige Kategorie von Präprozessoranweisungen sind Bedingungen. Sie dienen dazu, Quelltextbestandteile je nach Konfiguration ein- oder auszublenden. Dadurch entstehen unterschiedliche Varianten des Quellcodes nach dem Präprozessorlauf. Dieses Verfahren nennt man *bedingte Kompilierung*. Ein Beispiel für bedingte Kompilierung ist in Listing 1.1 zu sehen. Hier wird die Technik genutzt, um den Quellcode für verschiedene Betriebssysteme zu konfigurieren. Die Möglichkeit der Konfiguration kann aber in vielen verschiedenen Bereichen eingesetzt werden. Der Linux-Kernel[1] enthält beispielsweise über 6.000 Merkmale

[1] https://www.linux.org/

(engl. *features*) (vgl. [Kästner et al. 2011, 2]), wodurch beim Kompilieren unzählige Varianten der Software erzeugt werden können. Ein weiteres wichtiges Einsatzgebiet ist das Debugging. Dort können bei Definition eines bestimmten Flags zusätzliche Informationen zur Fehlersuche ausgegeben werden (vgl. [Eisenecker 2008, 132ff]).

Verschiedene Präprozessordirektiven stehen zur bedingten Kompilierung zur Verfügung: Mit `#if` werden allgemein Bedingungen eingeleitet, hier kann eine Makrodefinition auch auf einen numerischen Wert oder eine Zeichenfolge geprüft werden. Die Anweisung `#ifdef` oder `#if defined` prüft, ob ein Bezeichner bereits vereinbart wurde. Analog fragen `#ifndef` und `#if !defined`, ob ein Bezeichner nicht definiert ist. Um alternative Fälle abzuprüfen, gibt es die Direktiven `#elif` und `#else`. Die so definierten Bedingungen enden immer mit der Anweisung `#endif` (vgl. [Wolf 2019, 233ff]).

Ein mit `#define` vereinbarter Bezeichner gilt bis zum Ende des Programms, außer die Definition wird mit `#undef` wieder aufgehoben (vgl. [Wolf 2019, 241]). Es gibt noch weitere Direktiven sowie einige im C-Standard definierte Makros, auf die hier nicht näher eingegangen wird.

Eisenecker [2008, 148ff] wägt die Vor- und Nachteile des Präprozessors ab. Er betont zunächst, dass dieser sowohl für die Einbindung von Bibliotheken als auch zur Konfiguration von Software für verschiedene Plattformen und Compiler unerlässlich sei. Auf der anderen Seite habe CPP auch viele Nachteile: Fehler in der Implementierung seien oft auf eine fehlerhafte Klammerung von Makros sowie eine fehlende Typprüfung des Präprozessors zurückzuführen. Zudem fielen Fehler erst bei der Verwendung von Makros auf, nicht bei ihrer Definition. Außerdem führe der häufige Einsatz von großen Makros zu aufgeblähten Programmen, da diese an jedem Verwendungsort ersetzt würden, anders als gewöhnliche Funktionsaufrufe. Als letzten Nachteil nennt Eisenecker die oft mangelhafte Unterstützung von Präprozessoranweisungen beim Debugging. Der Debugger kenne in der Regel nur den Code nach der Verarbeitung durch den Präprozessor. Auch Kästner et al. [2011, 1] weisen auf mehrere Probleme bei der Verwendung des C-Präprozessors hin: vermehrt syntaktische und semantische Fehler im Quellcode, sehr unübersichtlicher Code, vor allem, wenn Bedingungen sehr häufig und tief verschachtelt verwendet werden, sowie eine allgemeine Verschlechterung der Wartbarkeit.

2.1.4 Ablauf der Kompilierung

Zum besseren Verständnis des Präprozessors folgt hier eine kurze Erläuterung des Kompiliervorgangs von C-Programmen. Abbildung 2.1 zeigt den Ablauf der Kompilierung. Die Quelldatei wird zunächst vom Präprozessor verarbeitet, der die im vorangehenden Abschnitt beschriebenen Schritte durchführt. Anschließend übersetzt der Compiler den entstandenen Code zu Assemblercode. Dieser wiederum wird durch einen Assembler in Maschinencode übersetzt. Der Linker ergänzt verwendete Bibliotheken und verbindet am Ende die einzelnen Objektdateien zu einem ausführbaren Programm.

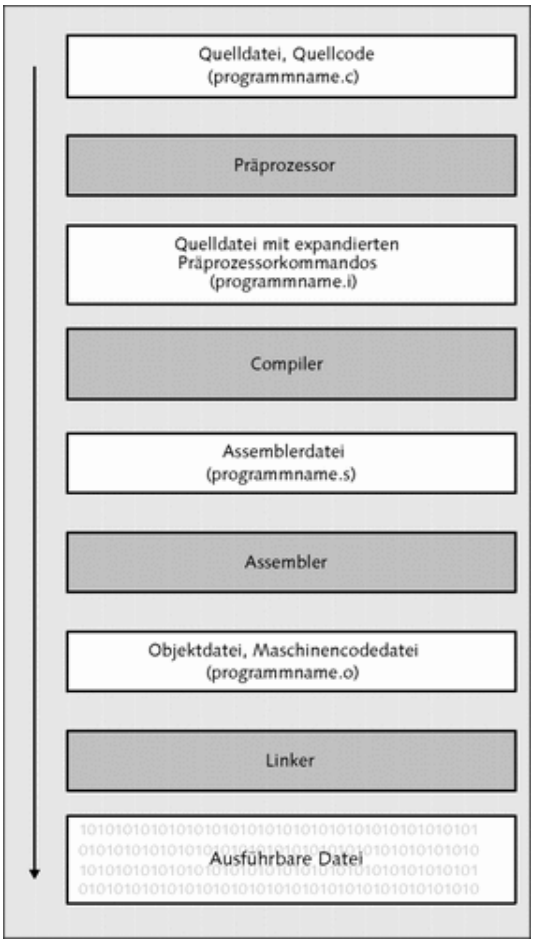

Abbildung 2.1: Ablauf der Kompilierung einer C-Quelldatei [Wolf 2019, 222]

2.2 Softwarevisualisierung

Die Visualisierung beschäftigt sich mit der grafischen Darstellung von Informationen, um ihr Verständnis und ihre Analyse zu erleichtern. Diese Technik spielt in vielen wissenschaftlichen Disziplinen eine große Rolle. Im folgenden Abschnitt soll die Softwarevisualisierung als ein Teilgebiet der Visualisierung definiert und erläutert werden, um die in dieser Arbeit entwickelte Visualisierung einordnen zu können.

2.2.1 Eigenschaften und Ziele

Die Visualisierung hat zwei wichtige Teilgebiete: Die wissenschaftliche Visualisierung (engl. *scientific visualization*) stellt physikalische Daten dar, während Informationsvisualisierung (engl. *information visualization*) abstrakte Daten visuell darstellt. Softwarevisualisierung wird der Informationsvisualisierung zugeordnet, da Algorithmen nicht der physischen Welt zuzurechnen sind, sondern eine Art von Information repräsentieren (vgl. [Diehl 2007, 3]).

Diehl [2007] unterscheidet eine enge und eine weite Definition von Softwarevisualisierung. Die enge Definition sieht Softwarevisualisierung „*as the visualization of algorithms and programs*“ [Diehl 2007, 3]. Hier werden also einige softwarebezogene Artefakte, bei-

spielsweise Entwicklungstickets oder Informationen aus Versionsverwaltungsprogrammen ausgeschlossen. Die weitere Definition umfasst jedoch auch diese Artefakte, indem sie Softwarevisualisierung als „*the visualization of artifacts related to software and its development process*“ [Diehl 2007, 3] definiert. Die vorliegende Arbeit visualisiert ausschließlich Quelltext und wäre somit auch nach der engeren Definition der Softwarevisualisierung zuzurechnen.

Die Aspekte von Software, welche in der Softwarevisualisierung dargestellt werden, lassen sich grob in drei Kategorien einordnen: die Struktur, das Verhalten und die Entwicklung beziehungsweise Historie von Software. Um die Struktur von Software abzubilden, muss ein Programm nicht ausgeführt werden, sondern alle Informationen können durch die statische Analyse des Quellcodes gewonnen werden. Die dargestellten Informationen sind also unabhängig von möglichen Eingabedaten. Beispiele für Informationen dieser Kategorie sind Softwaremodule, Funktionen, aber auch ein statischer Aufrufgraph. Die Verhaltensvisualisierung von Software benötigt hingegen die Ausführung eines Programms, da die Ausprägung der Visualisierung von den Eingabedaten abhängt. Mit Entwicklung oder Historie von Software sind Abfolgen von Versionen gemeint, die im Laufe des Softwareentwicklungsprozesses durch Änderungen entstehen (vgl. [Diehl 2007, 3f]).

Beispiele für alle drei Arten von Softwarevisualisierungen finden sich im folgenden Abschnitt zu Getaviz. Der in dieser Arbeit vorgestellte Prototyp lässt sich als Strukturvisualisierung einordnen, da die Visualisierung auf dem Quellcode aufbaut und keine Eingabedaten benötigt. Allerdings hängt die Struktur, welche das verwendete Werkzeug TypeChef ausgibt, von der Ausführung des Präprozessors ab. Zudem wird die Struktur beeinflusst von Makrodefinitionen im Quellcode und beim Aufruf von Typechef. Insofern lässt sich die hier entwickelte Visualisierung bezüglich des C-Quellcodes der Strukturvisualisierung zuordnen, die Visualisierung der Präprozessordirektiven gehört jedoch der Verhaltensvisualisierung an.

Ziele der Softwarevisualisierung sind nach Diehl die Erleichterung des Verständnisses von Software und daraus resultierend die Erhöhung der Produktivität im Softwareentwicklungsprozess (vgl. [Diehl 2007, 4]). Bassil und Keller stellen fest, dass Softwarevisualisierungen vor allem das Verständnis von Quellcode unterstützen, sodass die Wartung, das Testen, das Implementieren neuer Funktionalitäten und einige weitere Aktivitäten unterstützt werden (vgl. [Bassil/Keller 2001, 7]).

2.2.2 Getaviz

Getaviz ist ein Toolset zur Erstellung und Nutzung von Softwarevisualisierungen und zur Durchführung von empirischen Evaluationen der Modelle. Es wird von der Forschungsgruppe “Visual Software Analytics” des Instituts für Wirtschaftsinformatik an der Universität Leipzig entwickelt. Einen guten Überblick über Getaviz findet man in der Arbeit “GETAVIZ: Generating Structural, Behavioral, and Evolutionary Views of Software Systems for Empirical Evaluation” von Baum et al. [2017]. Im Folgenden sollen die Grundlagen kurz erläutert werden.

Das Toolset besteht aus vier Komponenten:

1. Aus Extraktoren für verschiedene Programmiersprachen und Versionsverwaltungssysteme,
2. einem Softwarevisualisierungsgenerator,
3. einer browserbasierten graphischen Oberfläche, über die man die Visualisierung benutzen kann,
4. einem Evaluationsserver, um empirische Evaluationen durchzuführen

(vgl. [Baum et al. 2017, 1]).

Visualisierungen kann man in Getaviz für die drei Aspekte von Software erstellen, die schon im vorangehenden Abschnitt erläutert wurden:

1. Bei der Strukturbetrachtung werden Pakete, Klassen, Methoden und Attribute dargestellt.
2. Die Verhaltensvisualisierung zeigt Stacktraces von realen Programmausführungen.
3. Der dritte Aspekt, der visualisiert werden kann, ist die Historie, indem Versionsverwaltungssysteme ausgelesen und die verschiedenen Entwicklungsschritte als Versionen eines Modells dargestellt werden.

Alle drei Visualisierungsarten in Getaviz sind in den Abbildungen 2.2, 2.3 und 2.4 beispielhaft zu sehen.

Getaviz kann 2D- und 3D-Modelle erzeugen und unterstützt vier verschiedene Visualisierungsmetaphern, von welchen es teilweise noch Varianten gibt. In dieser Arbeit sollen die City- und die Recursive Disk (RD)-Metapher betrachtet werden, da diese am häufigsten genutzt und am weitesten ausgebaut sind.

Bislang ist die Visualisierung von vier Programmiersprachen möglich: Java, Ruby, ABAP und C#, allerdings können Verhaltensmodelle bisher nur für Java und Ruby erstellt werden.

Der Generator basiert auf dem Ansatz der generativen und modellbasierten Softwareentwicklung. In mehreren Schritten wird ein aus dem Quellcode gewonnenes Modell modifiziert. Dabei werden Sprachmittel in verschiedene Visualisierungselemente der Metapher übersetzt und weitere Eigenschaften der Visualisierung berechnet und ergänzt. Durch den modellbasierten Ansatz ist der Generator für neue Eingabedaten und Visualisierungsmetaphern einfach erweiterbar.

Bisher wurde in Getaviz das Eclipse Modeling Framework, insbesondere Xtext[2] und Xtend[3] eingesetzt. Allerdings hat diese Implementierung den Nachteil, dass die Modelle

[2]https://www.eclipse.org/Xtext/
[3]https://www.eclipse.org/xtend/

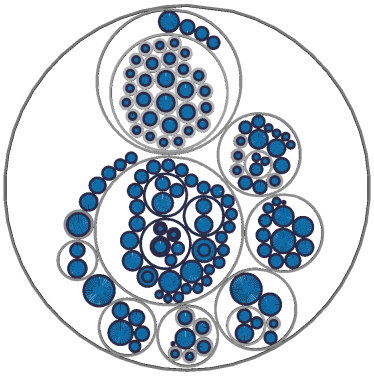

Abbildung 2.2: Beispiel für die Visualisierung der Struktur eines Softwaresystems in Getaviz [Baum 2017, 2]

Abbildung 2.3: Verhaltensvisualisierung in Getaviz: Dauer eines Aufrufs als Höhe einer Methode [Baum 2017, 3]

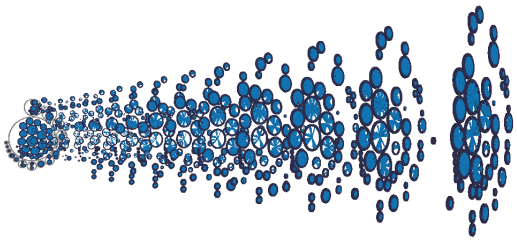

Abbildung 2.4: Visualisierung der Historie in Getaviz: Darstellung von verschiedenen Versionen eines Softwaresystems [Baum 2017, 3]

vollständig in den Hauptspeicher geladen werden, was die Größe der visualisierten Software beschränkt.

Aus diesem Grund wurde eine neue Version des Generators erstellt, welche die Modelle in Neo4j[4], einer Graphdatenbank, speichert. Auf Neo4j wird in Abschnitt 2.3 näher eingegangen. Durch den Einsatz der Graphdatenbank können das Eclipse Modeling Framework und die FAMIX-Modelle in Zukunft vollständig abgelöst werden. Bislang gibt es allerdings noch keine Neuimplementierung für die Abbildung von Verhalten und Historie.

2.3 Graphdatenbanken und Neo4j

In diesem Abschnitt sollen Eigenschaften von Graphdatenbanksystemen allgemein und von Neo4j im Besonderen beschrieben werden, um das Verständnis der folgenden Kapitel zu erleichtern.

[4] https://neo4j.com/

Graphdatenbanksysteme speichern Daten nicht in Form von Relationen oder Objekten, sondern in Form von Graphen. Graphen sind eine Menge von Knoten und Kanten, wobei die Kanten Beziehungen zwischen den Knoten darstellen. Dadurch lassen sich reale Objekte, die durch Beziehungen untereinander verbunden sind, einfach abbilden und analysieren (vgl. [Robinson et al. 2015, 1ff]).

Das verbreitetste Datenmodell für Graphdaten ist der *labeled property graph*. Dieser hat folgende Eigenschaften: Der Graph enthält Knoten und Kanten. Knoten können Eigenschaften (engl. *properties*) haben, welche als Schlüssel-Wert-Paare ausgedrückt sind. Außerdem können Knoten einen oder mehrere Bezeichner (engl. *labels*) haben. Beziehungen sind ebenfalls benannt und haben eine Richtung, sie können auch Eigenschaften haben (vgl. [Robinson et al. 2015, 4]). Um den Aufbau zu verdeutlichen, zeigt Abbildung 2.5 ein einfaches Beispiel eines *labeled property graphs*. Die Kreise stehen für Knoten, die Beschriftungen in den Knoten sind Labels. In an den Knoten liegenden Kästen befinden sich Properties der Knoten. Verbunden werden die Knoten durch Kanten, welche ebenfalls Labels haben.

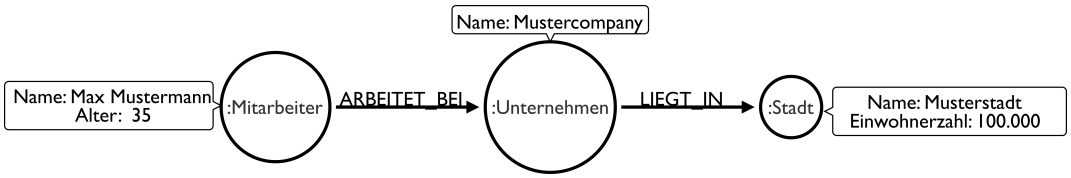

Abbildung 2.5: Einfaches Beispiel eines *labeled property graph*

Wie die Grafik verdeutlicht, stehen in einem Graphdatenmodell Beziehungen gleichwertig neben Entitäten. Sie werden explizit formuliert und müssen im Gegensatz zu relationalen Datenbanken nicht aus Fremdschlüsselbeziehungen abgeleitet werden (vgl. [Robinson et al. 2015, 4]). Aus diesem Grund eignen sich Graphdatenbanken besonders für Anwendungsgebiete, in denen Beziehungen zwischen Objekten eine große Rolle spielen. Beispiele für Anwendungsfälle sind: Beziehungen zwischen Nutzern in sozialen Netzwerken, Produktempfehlungen, welche aus Käufen anderer Nutzer abgeleitet werden, und Geoinformationen, die zur Routenberechnung genutzt werden (vgl. [Robinson et al. 2015, 107ff]).

Bei relationalen Datenstrukturen sinkt die Performanz bei Join-Abfragen, je größer die Datenmenge ist. Bei Graphdatenbanken hingegen bleibt die Abfragegeschwindigkeit in der Regel annäherungsweise konstant, da hier nur der Teil des Graphs durchlaufen werden muss, der für die Abfrage relevant ist. Zudem bieten sie eine große Flexibilität, bezüglich des Hinzufügen von Entitäten und Beziehungen, da dies bestehende Strukturen nicht beeinflusst. Dadurch sind sie besonders in schnelllebigen Domänen von Vorteil, weil keine aufwendigen Schemamigrationen und Anpassungen der Anwendungen notwendig werden, wie es bei relationalen Datenbanken oft der Fall ist. Graphdatenbanken besitzen kein Schema, was agile Entwicklungsmethoden unterstützt, gleichzeitig aber auch einen sorgfältigen Umgang mit den Daten notwendig macht. Ein weiterer Vorteil ist, dass Graphdatenbanken oftmals automatisches Testen durch ihre API unterstützen (vgl. [Robinson et al. 2015, 8f]). Robinson et al. betonen, dass das Abbilden von realen Problemen auf Graphdatenmodelle intuitiv

ist und dass diese Modelle einfach sind im Vergleich zu relationalen Datenmodellen (vgl. [Robinson et al. 2015, 4, 6]).

Die genannten Vorteile machen Graphdatenbanken interessant für den Einsatz in Getaviz. Dort sollen auch große Softwareprojekte gespeichert und verwendet werden. Zudem sind Beziehungen von Quellcodebestandteilen untereinander und zu den Komponenten der Visualisierungsmetapher wichtig. Durch den Einsatz einer Graphdatenbank können außerdem während des Generierungsprozesses weitere Elemente flexibel zu der Datenbank hinzugefügt werden, beispielsweise bei der Erzeugung der Visualisierungskomponenten.

Das konkrete Graphdatenbankensystem, das in Getaviz eingesetzt wird, heißt Neo4j. Es wurde 2007 als Open-Source-Graphdatenbank veröffentlicht und bietet neben einer performanten und stabilen Datenbank einige weitere Funktionalitäten, wie die Abfragesprache Cypher, eine Bibliothek für performante Graph-Algorithmen und eine Webanwendung zur Abfrage und Visualisierung von Graphdaten (vgl. [o.V. 2019a]).

Um die Graphdatenbank zu befüllen, wird in der schon implementierten Visualisierung von Java-Quellcode jQAssistant[5] eingesetzt. Dieses Werkzeug dient eigentlich der regelbasierten Analyse von Quellcode. Dazu bietet es eine API, um Objekte in einer Neo4j-Datenbank anzulegen. Mithilfe von Plugins kann jQAssistant einfach erweitert werden, um bestimmte Dateitypen zu verarbeiten (vgl. [Mahler 2018]). Das Java-Quellcode-Plugin ist beispielsweise folgendermaßen aufgebaut: Ein Parser wird für die Dateien aufgerufen. Für die zurückgegebenen Elemente des Quellcodes werden daraufhin Knoten, Beziehungen und Eigenschaften in der Graphdatenbank angelegt, gleichzeitig werden Typen aufgelöst und einige Quellcodemetriken gespeichert. Das Graphmodell kann anschließend für die Visualisierung weiterverwendet werden.

[5] https://jqassistant.org/

3 Konzeption

In diesem Kapitel erfolgt ein strukturierter Entwurf des Prototyps. Dazu wird zunächst der Anwendungsfall für die zu entwerfende Softwarevisualisierung definiert, um diese zweckmäßig zu gestalten. Daraus ergeben sich die benötigten Informationen für die Visualisierung, welche durch Parsen gewonnen werden. Anschließend sollen sie zur Weiterverarbeitung in einem Graphmodell abgelegt werden. Diese Schritte finden sich in Abschnitt 3.2.

Im Abschnitt 3.3 wird aus dem Anwendungsfall eine Softwarevisualisierung abgeleitet. Da der entwickelte Prototyp auf Getaviz aufbaut, werden die bestehenden Visualisierungsmetaphern auf ihre Eignung für den Prototypen untersucht und eine Übertragung für die Programmiersprache C wird beschrieben. Zuletzt erfolgt eine Konzeption der notwendigen Anpassungen der Benutzungsoberfläche an den Anwendungsfall.

3.1 Anwendungsfall

Das Ziel von Softwarevisualisierung ist, wie in Abschnitt 2.2.1 beschrieben, die Verbesserung des Verständnisses eines Softwareartefaktes und daraus folgend eine Produktivitätssteigerung in der Softwareentwicklung beziehungsweise -wartung. Um dies zu erreichen, muss eine Softwarevisualisierung für den Nutzer nützlich sein und überhaupt eingesetzt werden. Dies bemängelt Hundhausen [1996] bei vielen bestehenden Softwarevisualisierungen: Problematisch sei, dass in der Forschung viele Systeme entworfen und prototypisch umgesetzt, aber dann nicht weiterentwickelt und demzufolge auch nie im Produktivbetrieb eingesetzt würden. Als Ursache sieht der Autor die Intention, mit der viele Systeme entworfen würden: „*technical challenges, as well as a desire for technological innovation, are the principal forces behind system design*“ [Hundhausen 1996, 2]. Dies führe dazu, dass die Visualisierungen nicht zu den Bedürfnissen ihrer angedachten Nutzer passten oder bei der Problembewältigung schlichtweg keine Hilfe seien (vgl. [Hundhausen 1996, 2]).

Um die genannten Probleme zu beseitigen, muss nach Hundhausen [1996] der Ansatz bei der Entwicklung von Softwarevisualisierungen grundlegend verändert werden. Nicht mehr die Ausdrucksfähigkeit (engl. *expressiveness*) solle ausschlaggebend sein, sondern die Zweckmäßigkeit (engl. *effectiveness*). Ausdrucksfähigkeit wird hier als die Fähigkeit eines Systems gesehen, verschiedene Arten von Visualisierungen zu erzeugen, um die gewünschten Informationen darzustellen, während Zweckmäßigkeit beschreibt, zu welchem Grad ein System die Bedürfnisse von Nutzern erfüllt und wie einfach es von diesen benutzt werden kann. Hundhausen [1996, 3] folgert daraus, dass Ausdrucksfähigkeit notwendig, aber nicht hinreichend für die Zweckmäßigkeit sei. Zudem beziehe sich Zweckmäßigkeit immer auf die Nutzung einer Softwarevisualisierung im Kontext einer Aufgabe.

Daraus folgt, dass eine Softwarevisualisierung im Kontext einer Aufgabe entworfen werden muss, um die Zweckmäßigkeit für die Erfüllung dieser Aufgabe gewährleisten zu kön-

nen. Deshalb soll hier zunächst die Aufgabe betrachtet werden, die mit der hier entwickelten Softwarevisualisierung gelöst werden kann, der Anwendungsfall der Softwarevisualisierung.

Zur Beschreibung der Aufgabe dienen vier Aspekte, die Hundhausen im Zusammenhang einer Aufgabe nennt (vgl. [Hundhausen 1996, 4]):

- Ziel der Aufgabe: Allgemein gefasst ist das Ziel der Aufgabe, das Softwaresystem besser zu verstehen. Genauer definiert, soll ein Überblick über wichtige Elemente des Quellcodes gewonnen werden. Dabei ist besonders die Ausprägung derselben im Kontext von bedingter Kompilierung von Bedeutung. Konkret sollen folgende Fragen beantwortet werden: Welche Auswirkung hat die Definition einer symbolischen Konstante oder einer Gruppe von symbolischen Konstanten auf die Struktur des Programms? Welche Elemente sind im Quellcode enthalten, wenn eine bestimmte Gruppe von symbolischen Konstanten definiert ist?
- Gruppe von Menschen, die dieses Ziel verfolgt: Das genannte Ziel wollen vor allem Softwareentwicklerinnen erreichen, die mit der Wartung und Erweiterung der Software beschäftigt sind. Einen derart detaillierten Einblick in den Quellcode benötigen nur Personen, die auch direkt mit diesem arbeiten wollen. Für die Visualisierung der Kombinationen von symbolischen Konstanten wären auch Personen als Zielgruppe denkbar, die mit der Einrichtung und Konfiguration des Programms beschäftigt sind, jedoch ist für diese eine Quellcodeansicht nicht vonnöten.
- Softwarevisualisierungsartefakt, das genutzt wird: Das Softwarevisualisierungsartefakt, das für diese Aufgabe genutzt werden soll, ist der hier entwickelte Prototyp. Er ist eingebettet in die Visualisierungsoberfläche von Getaviz. Diese bietet zum jetzigen Zeitpunkt metapherbasierte Softwarevisualisierungen, welche einen Überblick über die Struktur des untersuchten Softwareprodukts geben. Zudem enthält die Darstellung Tooltips mit Details der Elemente, eine Quellcodeansicht, Möglichkeiten zum Filtern, Zoomen und Auswählen von Elementen und einiges mehr.
- Programm, das visualisiert wird: Visualisiert werden soll der Quellcode eines C-Programms. Hier ist jedoch zu beachten, dass Teile des Quellcodes vor der Visualisierung vorverarbeitet werden sollen, beispielsweise `#include`-Anweisungen.

Zusammengefasst besteht die Aufgabe darin, dass ein Softwareentwickler die Auswirkung der Definition und von möglichen Kombinationen symbolischer Konstanten auf die Struktur des Quellcodes verstehen möchte. Auch wenn Probleme in einer Variante des Quellcodes auftreten, können durch Definition der entsprechenden symbolischen Konstanten die Struktur des Quellcodes erforscht und die möglichen Fehlerquellen eingegrenzt werden. Dies wird vor allem im Zusammenhang mit der Wartung und Weiterentwicklung der den Entwicklern weitgehend unbekannten Softwaresysteme notwendig. Eine freie und interaktive Exploration der Kombinationen und ihrer Auswirkung wäre wünschenswert. Jedoch sollten vorhandene symbolische Konstanten für den Nutzer angezeigt werden, um die Arbeit mit sehr vielen symbolischen Konstanten zu erleichtern.

Unter diesen Voraussetzungen ist eine Auswahlkomponente für symbolische Konstanten vorstellbar. Nach Auswahl beziehungsweise Definition eines oder mehrerer Elemente werden diese ausgewertet und die Strukturdarstellung des Quellcodes wird entsprechend angepasst.

Um ein nützliches Softwarevisualisierungssystem zu entwickeln, ist eine Evaluation seiner Zweckmäßigkeit unerlässlich. Anschließend müssen weitere Zyklen aus Entwicklung und Evaluation folgen. Das Vorgehen bezüglich des im Rahmen dieser Arbeit entwickelten Prototypen wird in Abschnitt 4.5 erläutert.

3.2 Extraktion der benötigten Informationen

Dieser Abschnitt konzipiert die Extraktion der benötigten Informationen für die Visualisierung. Dazu müssen zunächst die Strukturelemente, die für die Visualisierung von Bedeutung sind, anhand des C-Standards identifiziert werden. Anschließend folgt eine Erläuterung und Abwägung von verschiedenen Möglichkeiten zur Extraktion aus dem Quellcode. Der letzte Teil beschreibt den Entwurf eines Graphmodells zur Ablage der gewonnenen Informationen.

3.2.1 Sprachmittel im C-Standard

Der in dieser Arbeit entworfene Prototyp visualisiert die Struktur von C-Quellcode. Das heißt, dass die Informationen durch eine statische Codeanalyse gewonnen werden können. Da das Ziel des Prototyps ist, einen schnellen Überblick über die Struktur des Quellcodes zu gewinnen, sollen nur wichtige Strukturelemente visualisiert werden. Einzelne Anweisungen oder Algorithmen sind nicht von Bedeutung. Unter diesen Voraussetzungen werden im Folgenden die Sprachmittel des C-Standards von 2017 (vgl. [ISO/IEC 2017]) untersucht und auf ihre Relevanz hin bewertet. Eine vollständige Darstellung der Ergebnisse findet sich in Anhang A. Die Tabelle listet alle identifizierten Sprachmittel, gruppiert nach ihrem Abschnitt im Standard, auf. Eine Bewertung der Relevanz der Sprachmittel liefert die zweite Spalte. Für alle gekennzeichneten Elemente müssen Informationen aus dem Quelltext extrahiert werden. In der dritten Spalte findet sich die Angabe, ob das entsprechende Element schon in der Implementierung umgesetzt wurde. Da es sich hierbei nur um einen Prototypen handelt, konnten nicht alle relevanten Sprachmittel Eingang in die Implementierung finden. Diese Auswahl wird in Kapitel 4 textuell beschrieben.

Vom gesamten Standard wurde nur Kapitel Sechs betrachtet, da sich hier die Beschreibung der Sprachbestandteile findet. Die anderen Kapitel enthalten beispielsweise eine Schilderung der Ausführungsumgebung und die Bibliotheken, welche im Standard enthalten sind. Abschnitte in Kapitel Sechs, die keine Sprachmittel definieren, sondern beispielsweise zulässige Konversionen von Datentypen, werden in der Tabelle nicht aufgeführt.

Im Folgenden wird die Tabelle zusammenfassend erläutert. Zunächst werden die Elemente aufgezählt, welche direkt in der Visualisierung abgebildet werden sollen. Anschließend folgt

eine Beschreibung der notwendigen Metainformationen, die ebenfalls aus dem Quellcode extrahiert werden müssen. Zuletzt sollen auch die Sprachmittel genannt werden, die nicht Eingang in die Visualisierung gefunden haben.

Zur Strukturierung und einfacheren Wiederverwendung von Quellcode besitzt C Funktionen. Diese kapseln Algorithmen und sind wichtig zum Verständnis eines Programms. Deshalb sollen sie als eigenständiges Element in der Visualisierung abgebildet werden. Man unterscheidet zwischen Funktionsdeklaration und -definition. In C werden Funktionsdeklarationen häufig in Headerdateien vorgenommen. Fallen Deklaration und Definition auseinander, befinden sich aber in der gleichen Übersetzungseinheit, sollen sie nichtsdestotrotz als ein einzelnes Element dargestellt werden. Der Körper einer Funktion wird aufgrund des zu großen Detailgrades nicht betrachtet. Statische Funktionsaufrufe werden visualisiert, da sie eine wichtige Strukturinformation darstellen.

Wichtig für die Struktur von Quellcode sind auch Variablen. Da nur ein Überblick über den Quellcode gegeben werden soll und nicht die Implementierung der Algorithmen von Bedeutung ist, wird nur die Deklaration von globalen Variablen visualisiert. Als wichtige Information für das Verständnis wird auch das Lesen und Schreiben von globalen Variablen gesehen, da so Abhängigkeiten zwischen verschiedenen Dateien und Übersetzungseinheiten sichtbar werden.

Zur Kapselung mehrerer Variablen bietet C Structs und Unions. Innerhalb dieser Elemente werden wiederum mehrere Variablen deklariert, sodass ein komplexer Datentyp entsteht. Auch die Deklaration dieser beiden Elemente soll in der Visualisierung abgebildet werden. Um sie besser zu verstehen, werden nicht nur ein Struct oder Union an sich abgebildet, sondern auch ihre enthaltenen Variablen.

Enums definieren zusammengehörige Konstanten. So bilden sie ebenfalls größere Strukturelemente. Sie werden analog zu Structs und Unions abgebildet, das heißt, die Visualisierung enthält Enums und diese wiederum enthalten die darin deklarierten Konstanten.

Ebenfalls mit einem “V” sind in der Tabelle die Präprozessordirektiven gekennzeichnet, welche die bedingte Kompilierung erzeugen. Auf sie wird im nächsten Abschnitt eingegangen.

Da C keine objektorientierte Programmiersprache ist, finden sich keine Objekte als übergreifendes Strukturelement. Jedoch enthält C sogenannte Übersetzungseinheiten (engl. *translation units*). Diese finden keine explizite Darstellung im Quellcode, sondern eine Quelltextdatei mitsamt allen eingebundenen Header- und Quellcodedateien bildet eine sogenannte *preprocessing translation unit*. Nach der Ausführung des Präprozessors wird daraus eine Übersetzungseinheit. Ein C-Programm kann aus einer oder mehreren Übersetzungseinheiten bestehen. Teile des Programms können auch separat kompiliert und bei der Ausführung beispielsweise über Funktionsaufrufe verbunden sein. Übersetzungseinheiten sollen als gliederndes und andere Sprachmittel verbindendes Element Eingang in die Visualisierung finden.

Die Gruppe der Sprachmittel, welche als Glyphen in der Visualisierung abgebildet werden, umfasst also Übersetzungseinheiten, Funktionen, globale Variablen, Structs und Unions samt ihrer deklarierten Variablen sowie Enums und die von ihnen deklarierten Konstanten.

Zahlreiche Elemente des Standards sind wichtig für die Metainformationen der Visualisierung. Dort sollen für Funktionen der Funktionskopf, für Variablen der Datentyp und der Bezeichner und für Structs, Unions und Enums ebenfalls der Bezeichner, sowie entsprechende Metainformationen für die enthaltenen Elemente, extrahiert werden. Dazu sind alle Datentypen aus dem Standard wichtig, wie sie in der Tabelle unter 6.2.5 sowie unter 6.7.2 nochmals als *Type Specifier* aufgelistet werden. Die in der Tabelle genannten *Identifiers* spielen als Bezeichner für die visualisierten Strukturelemente eine Rolle. Ebenfalls für die Metainformationen von Bedeutung sind die *Type Qualifiers* und *Storage Class Specifiers*, welche zusätzlich die Verwendung und Modifikation von Typen und die Speicherung der Elemente spezifizieren. *Function specifiers* beschreiben das Verhalten von Funktionen näher und sind als Teil des Funktionskopfes ebenfalls wichtig für die Metainformationen. *Punctuators* und *Keywords* definieren ganz allgemein Zeichen und Zeichenfolgen mit einer speziellen Bedeutung für die Syntax und sind nur insofern relevant, dass sie andere relevante Sprachmittel identifizieren.

Konstanten und Stringliterale sind nicht bedeutsam für die Visualisierung, da sie zur Initialisierung von Variablen dienen oder zum Vergleich in Bedingungen, jedoch werden diese Elemente nicht visualisiert. Initialisierungen und Zuweisungen sind nur relevant, wenn sie globale Variablen betreffen. Die unter dem Abschnitt *Ausdrücke* (engl. *expressions*) zusammengefassten Sprachmittel sind zu detailliert für die überblicksartige Darstellung, welche hier entstehen soll. Eine Ausnahme bilden, wie schon beschrieben, Funktionsaufrufe. Auch Anweisungen (engl. *statements*), worunter beispielsweise Schleifen und Bedingungen fallen, passen nicht zum Detailgrad der Visualisierung. Die Beschreibung der Präprozessoranweisungen erfolgt in ausführlicher Form in Abschnitt 3.2.2. Hier werden nur die nicht relevanten Elemente genannt: Das Einbinden von Dateien bestimmt den Umfang der Übersetzungseinheit, wird jedoch nicht explizit visualisiert. Die Expansion von Makros soll durchgeführt werden, sodass nur der expandierte Code visualisiert wird. Vordefinierte Makros, wie `#line` oder `#pragma`, spielen keine Rolle.

3.2.2 Variabilität in C-Quellcode

Variabel, das Adjektiv zu Variabilität, wird folgendermaßen definiert: „*nicht auf nur eine Möglichkeit beschränkt; veränderbar, [ab]wandelbar*“ [Bibliographisches Institut GmbH 2018]. Variabilität findet sich in C-Quellcode auf zwei Ebenen: Zum einen durch unterschiedliche Eingabedaten bei der Ausführung des Programms, zum anderen durch Präprozessoranweisungen, indem sie bei der Kompilierung Varianten eines Programms erzeugen.

Dieser Abschnitt erläutert beide Arten der Variabilität und erklärt, welche Art bedeutend für den hier vorgestellten Prototypen ist.

Variabilität durch unterschiedliche Eingabedaten wird im Quellcode durch einige Sprachmittel unterstützt: Variablen nehmen Werte ihres Datentyps auf und dienen zur Weitergabe und Verarbeitung derselben. Parametrisierte Funktionen verändern ihr Verhalten durch die Werte der übergebenen Parameter. Durch Anweisungen, welche Bedingungen prüfen, wie `if`-Anweisungen oder `switch-case`-Anweisungen, kann, abhängig vom Wert der geprüften Bedingung, der Ablauf des Programms beeinflusst werden. Die genannten Beispiele sind keine vollständige Auflistung, verdeutlichen jedoch die Bedeutung von Variabilität bei der Ausführung von Software. Listing 3.1 zeigt einen Ausschnitt einer Funktion aus der Datei `cookie.c` des cURL-Projekts. In Zeile 6 wird der Wert des Parameters `domain` geprüft und wenn dieser falsch beziehungsweise 0 ist, wird die Funktion mit dem Rückgabewert `NULL` verlassen.

```
1 static const char *get_top_domain(const char * const domain,
     size_t *outlen)
2 {
3   size_t len;
4   const char *first = NULL, *last;
5
6   if(!domain)
7     return NULL;
```

Listing 3.1: Beeinflussung des Programmablaufs durch einen Parameterwert

Diese Art der Variabilität kommt jedoch nur bei der Ausführung eines Programms zum Tragen und ist somit der Verhaltensvisualisierung zuzurechnen. Die Sprachmittel, die diese Variabilität unterstützen, werden zum Teil in dem hier beschriebenen Prototypen dargestellt (zum Beispiel globale Variablen und Funktionsparameter), zum Teil aber auch nicht, da nur bestimmte Strukturelemente abgebildet werden.

Czarnecki [2013] beschreibt Variabilität als „*[...]the ability to create system variants for different market segments or contexts of use*“ [Czarnecki 2013, 1]. Er bezieht sich hierbei unter anderem auf die Variabilität, die für Softwareproduktlinien notwendig ist. Nach Liebig et al. [2010, 1] kann der C-Präprozessor zur Implementierung der Variabilität in Softwareproduktlinien genutzt werden. Durch den Einsatz von bedingter Kompilierung, Makroexpansion et cetera entstehen sogenannte Varianten. Je nach Konfiguration werden durch den Präprozessor manche Quellcodebestandteile ein- beziehungsweise ausgeblendet, wodurch bei der Kompilierung eine Variante der Software erzeugt wird. Ein Vorteil dieser Technik, besonders im Kontext von Softwareproduktlinien, ist die einfache Wiederverwendung von gleichen Codebestandteilen. An der Stelle, an der bedingte Kompilierung im

Quellcode verwendet wird, entsteht ein sogenannter Variationspunkt (engl. *variation point*) (vgl. [Czarnecki 2013, 2]). Listing 3.2 zeigt einen solchen Variationspunkt in der Datei `connect.c` des cURL-Projektes. Hier wird eine Variable, abhängig von der Definition eines Makros, unterschiedlich initialisiert.

```
1 #ifdef ENABLE_IPV6
2   is_tcp = (addr.family == AF_INET || addr.family == AF_INET6
       ) &&
3     addr.socktype == SOCK_STREAM;
4 #else
5   is_tcp = (addr.family == AF_INET) &&
6     addr.socktype == SOCK_STREAM;
7 #endif
```

Listing 3.2: Beispiel eines Variationspunktes in der Datei connect.c des cURL-Projektes

In Kapitel 2.1.3 wurde die Funktionsweise des C-Präprozessors bereits erläutert. Nun folgt eine Beschreibung, wie CPP Variabilität von Quellcode unterstützt. Dazu sollen einzelne Sprachmittel detailliert untersucht werden:

Kommentare dienen nicht der Variabilität, da sie keinen Einfluss auf den Programmablauf haben. Auch das Einbinden von Dateien unterstützt Variabilität nicht direkt. Zwar kann durch unterschiedliche Dateien eine Anpassung des Programms an die Umgebung vorgenommen werden. Allerdings muss die Auswahl dieser Dateien durch einen zusätzlichen Variabilitätsmechanismus geschehen, in welchem Bedingungen geprüft werden. Durch Funktionsmakros kann bei der Makroexpansion Variabilität im Quellcode erzeugt werden. Jedoch kommt diese Variabilität wieder nur zum Zeitpunkt der Programmausführung zum Tragen, wie bei der vorher beschriebenen Art der Variabilität durch den C-Quellcode. Die genannten Sprachmittel des C-Präprozessors sind also für die Betrachtung der Variabilität auf Ebene des CPP nicht interessant.

Liebig et al. [2010, 2] beschreiben, wie durch den Präprozessor induzierte Variabilität in C-Quellcode entsteht: Zunächst muss eine *Merkmalkonstante* (engl. *feature constant*) definiert werden, die ein bestimmtes *Merkmal* repräsentiert. Ein Merkmal stellt im Bereich der Softwareproduktlinien „*an optional or incremental unit of functionality*“ [Liebig et al. 2010, 2] dar. Das heißt, durch die Auswahl eines oder mehrerer Merkmale entsteht eine Variante eines Softwaresystems. Im zweiten Schritt muss dieses Merkmal im Präprozessorcode abgefragt werden, beispielsweise mit der Anweisung `#ifdef`. So wird die Konfiguration bestimmt und variable Quellcodebestandteile werden dementsprechend ausgewählt.

Die Definition einer Merkmalkonstante geschieht mit der Anweisung `#define`. Listing 3.3 zeigt drei Arten von sogenannten symbolischen Konstanten, welche bei Einsatz des CPP als Merkmalkonstanten dienen. Im ersten Fall wird nur ein Bezeichner definiert, der

dem Präprozessor anschließend bekannt ist. Der zweite Fall definiert eine Ersetzung für den Bezeichner `FEATURE2`. Im dritten Fall ist der Ersetzungstext parametrisiert, dies nennt man ein Funktionsmakro.

```
1 #define FEATURE1
2
3 #define FEATURE2 featurewert
4
5 #define FEATURE3(x) ((x) < 100)
```

Listing 3.3: Verschiedene Arten von symbolischen Konstanten im CPP

Die erste Merkmalkonstante in Listing 3.3 kann nur zwei Werte annehmen, wie eine boolesche Variable. Entweder sie ist definiert oder nicht.

`FEATURE2` hingegen kann als Wert einen beliebigen Inhalt haben. Beim Abfragen ist es zulässig, die Konstante dahingehend zu prüfen, ob sie definiert ist. Es können jedoch auch mehr als zwei Fälle aus dem Wert der Konstante konstruiert werden.

Durch die Parametrisierung von `FEATURE3`, hängt der Wert der Bedingung von einem oder mehreren Eingabewerten ab. Auch hier kann wieder die Definition des Makros oder ein bestimmter Wert abgefragt werden.

Mit der Anweisung `#undef` wird die Definition von symbolischen Konstanten wieder aufgehoben. Dadurch sind sie nur in einem Teil des Quellcodes definiert.

Auf diese Weise können also symbolische Konstanten und Makros für die Variabilisierung des Quellcodes erzeugt werden. Wie diese im zweiten Schritt abgefragt werden, um optionale Codebestandteile zu erstellen, wird nun beschrieben.

Die Anweisungen `#ifdef` und `#if defined` prüfen, ob eine symbolische Konstante oder ein Makro definiert sind. Analog dazu prüfen `#ifndef` und `#if !defined`, ob dies nicht der Fall ist. Bei dieser Art der Bedingung sind nur zwei Fälle möglich: definiert oder nicht definiert. Mit `#if` hingegen kann abgeprüft werden, ob eine symbolische Konstante einen bestimmten Wert enthält oder ob ein Funktionsmakro nach Auswertung des Inhalts einen bestimmten Wert ergibt, der Wertebereich ist jedoch nicht eingeschränkt. `#else` und `#elif` prüfen alternative Fälle. Die eben genannten Anweisungen sind aufgrund ihrer Bedeutung für die bedingte Kompilierung wichtig für die Visualisierung und sollen als Information aus dem Quellcode extrahiert werden.

Problematisch für das Verständnis von Präprozessoranweisungen im Quellcode ist die große Freiheit bei der Verwendung: Es können beliebige Codefragmente unter eine Bedingung gestellt (vgl. [Kästner et al. 2011, 1]) und Bedingungen können verschachtelt werden. Außerdem ist es möglich, auch die Definition von symbolischen Konstanten nur

unter bestimmten Bedingungen zu vollziehen oder diese in eingebundenen Headerdateien zu definieren.

Viele Werkzeuge zum Verständnis und Refactoring von C-Quellcode arbeiten mit einer präprozessierten Variante des Programms (vgl. [Ernst et al. 2002, 1147]). Dies bildet jedoch nicht den tatsächlichen Quellcode ab, mit dem eine Entwicklerin arbeitet und erleichtert nicht das Arbeiten mit zahlreichen Varianten. Deshalb soll die durch den C-Präprozessor eingeführte Variabilität im Quellcode bei der hier vorgestellten Visualisierung berücksichtigt und explizit visualisiert werden.

3.2.3 Extraktion von Variabilität

In diesem Kapitel wurden bisher die Sprachmittel von C definiert, die visualisiert werden sollen. Anschließend folgte eine Erläuterung der Arten von Variabilität in C-Quellcode. Es wurde dargelegt, warum bedingte Kompilierung die Verständlichkeit von Quellcode verschlechtert und dass die hier entworfene Visualisierung deshalb bedingte Kompilierung berücksichtigen und darstellen soll. Nun stellt sich die Frage, wie die benötigten Informationen aus dem Quellcode gewonnen werden können, um sie dann visuell darzustellen.

Anhang A stellt die zu extrahierenden Sprachmittel aus dem Standard dar. Diese müssen durch Parsen des Quellcodes gewonnen werden. Ob ein passendes Werkzeug hierfür bereits existiert, muss recherchiert werden. Dazu folgt eine Auflistung der Anforderungen an einen solchen Parser:

1. Das Ergebnis des Parsens muss alle für die Visualisierung notwendigen Sprachmittel aus dem C-Standard enthalten. Dazu zählen sowohl Strukturelemente, wie Structs oder Funktionen, als auch Typen sowie Informationen über Funktionsaufrufe und das Lesen und Schreiben von globalen Variablen.
2. Eingebundene Dateien soll der Parser eigenständig berücksichtigen und die Struktur entsprechend darstellen. Dies ist wichtig, um Deklarationen korrekt zu visualisieren.
3. Makroexpansionen soll der Parser vor dem Parsen durchführen, da die Expansion nicht Teil der Visualisierung ist und der Inhalt der Makros Einfluss auf die Visualisierung haben kann.
4. Das Ergebnis des Parsens soll Informationen zur bedingten Kompilierung enthalten. Wünschenswert wäre, wenn die Präprozessordirektiven aus dem Quellcode erhalten blieben. Noch nützlicher wäre eine Auswertung von verschachtelten Bedingungen und eine explizite Darstellung von verschiedenen Makrokombinationen als alternative Zweige im Ergebnis.

Für das Parsen ergeben sich prinzipiell zwei Möglichkeiten: Die erste besteht in der Eigenentwicklung eines solchen Parsers beziehungsweise in der Erweiterung eines bestehenden Parsers für C-Quellcode um die Möglichkeit, Präprozessoranweisungen zu verarbeiten. Die zweite Möglichkeit besteht in der Nutzung eines existierenden Werkzeugs, welches alle genannten Anforderungen erfüllt.

Kästner et al. [2011, 1] stellen fest, dass das Parsen von nicht präprozessiertem Code eine bedeutende Herausforderung darstellt. Unter anderem nennen sie das Problem, dass lexikalische Präprozessoren wie CPP unabhängig von der zugrundeliegenden Struktur verwendet werden können. Das bedeutet, dass die bedingten Quellcodebestandteile keine vollständigen Sprachmittel sein müssen, es können auch nur einzelne Zeichen oder Fragmente enthalten sein. Zudem merken Some/Lethbridge [1998, 1] an, dass durch ein einfaches Parsen ohne alternative Zweige Inkonsistenzen und Syntaxfehler im Code entstehen können. Dies kommt daher, dass sich manche Featurekombinationen gegenseitig ausschließen und nicht dazu gedacht sind, gemeinsam im Quellcode zu erscheinen.

Eine weitere Herausforderung des Parsens von nicht präprozessiertem Quellcode stellt das Einbinden von Dateien dar. Dieses Verhalten muss implementiert werden, allerdings enthalten eingebundene Dateien oftmals wiederum Dateiinklusionen, was zu einer großen Komplexität führt. Zudem können Dateien bedingt eingebunden werden. Des Weiteren ist auch das Expandieren von Makros, insbesondere von Funktionsmakros, nicht trivial.

Padioleau [2009] hat einen Parser, Yacfe, für C- und C++-Quellcode geschrieben, der auch Präprozessoranweisungen verarbeiten kann. Allerdings nutzt der Autor zur Vereinfachung des Parsens einige Annahmen darüber, wie Präprozessoranweisungen häufig verwendet werden. Beispielsweise nimmt er an, dass `#include`-Anweisungen immer am Anfang einer Datei stehen und dass symbolische Konstanten groß geschrieben werden. Zudem kann Yacfe nur einen kleinen Teil der bedingten Kompilierung verarbeiten. Alternative Zweige werden beim Parsen auskommentiert und der Entwickler ist aufgefordert, diese Anweisungen zu entfernen. Dadurch sind manuelle Überarbeitungen des Codes notwendig, wodurch eine automatische Generierung der Visualisierung nicht möglich ist.

An diesem Beispiel zeigt sich, dass das Parsen von Präprozessorcode sehr komplex ist und im Rahmen dieser Arbeit nicht umgesetzt werden kann. Daher liegt der Fokus auf der Recherche von bestehenden Werkzeugen.

Eine umfangreiche Suche in Datenbanken nach entsprechenden Parsern lieferte als Ergebnis drei mögliche Kandidaten: MAPR (vgl. [Platoff et al. 1991]), TypeChef (vgl. [Kästner et al. 2011]) und SuperC (vgl. [Gazzillo/Grimm 2012]). Alle drei Werkzeuge dienen der Analyse und dem Refactoring von C-Quellcode. Dazu parsen sie C-Quellcode mit Präprozessoranweisungen und erstellen daraus einen Abstract Syntax Tree (AST) beziehungsweise einen Abstract Syntax Graph (ASG), welche alternative Zweige enthalten. Das Parsen der alternativen Programmteile erfolgt in den Programmen durch Aufteilen des Parsers in mehrere Unterparser, um die alternativen Pfade zu durchlaufen. Anschließend werden sie wieder zusammengeführt. Dabei erzeugen die Parser alternative Unterbäume, die am AST beziehungsweise ASG ergänzt werden. Abbildung 3.1 stellt einem Quellcodefragment den entsprechenden AST gegenüber. Dieser enthält Anweisungen zur bedingten Kompilierung, welche im rechten Unterbaum als alternative Zweige abgebildet sind. Zur

Kennzeichnung der Bedingung wird das Rautensymbol mit dem Namen der symbolischen Konstanten verwendet. Abhängig vom Wert dieser Bedingung ist entweder der rechte oder der linke Unterbaum Teil des präprozessierten Quellcodes.

```
1 3 * 7 +
2 #ifdef X
3 1
4 #else
5 0
6 #endif
```

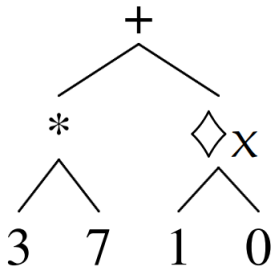

Abbildung 3.1: Beispiel eines AST mit dem entsprechenden Quellcode [Kästner et al. 2011, 3]

MAPR nutzt einen ASG, um gemeinsame Statements von Unterbäumen nicht replizieren zu müssen. Zum Parsen verwenden die Entwickler einen Parser-Generator. `#include`- und `#define`-Anweisungen verarbeitet MAPR wie C-Anweisungen. Allerdings ist ihre Verwendung Restriktionen unterworfen, um die Verarbeitung zu vereinfachen, sodass nicht alle Konstruktionen verarbeitet werden können (vgl. [Platoff et al. 1991]). Jedoch bemängeln Gazzillo/Grimm [2012, 1], dass MAPR einen eher naiven Algorithmus zur Erstellung der Unterparser nutzt, sodass unnötig viele Unterparser erzeugt werden. Außerdem versagt es beim Parsen der meisten Dateien des Linux-Kernels (vgl. [Gazzillo/Grimm 2012, 9]) und ist unzureichend dokumentiert (vgl. [Gazzillo/Grimm 2012, 1]).

SuperC ist nach eigenen Angaben die performanteste und vollständigste Lösung. Der Parser-Algorithmus baut auf dem von MAPR auf, fügt aber einige Optimierungen hinzu. In einem direkten Vergleich parst SuperC den Linux-Kern mehr als dreimal so schnell wie TypeChef (vgl. [Gazzillo/Grimm 2012, 11]). SuperC ist in Java geschrieben.

TypeChef nutzt statt eines GLR-Parsers einen LL-Parser, der auch zu der geringeren Performanz beiträgt. Das Ziel für die Entwickler war es, einen soliden Parser zu bauen, der C-Quellcode vollständig und ohne manuelle Vorverarbeitung parsen kann, um ihn anschließend zu analysieren (vgl. [Kästner et al. 2011]). Ihnen ist es gelungen, den Linux-Kernel vollständig zu parsen. Als Programmiersprache haben die Entwickler Scala verwendet.

Bei der Entscheidung zwischen den drei Werkzeugen kann MAPR aufgrund des fehlenden Vermögens, zahlreiche Dateien zu parsen, schnell ausgeschlossen werden. Die Entscheidung fiel in dieser Arbeit aus mehreren Gründen darauf, TypeChef zu nutzen. Zum einen erleichtert eine umfangreiche Dokumentation, das Programm aufzusetzen und zu bearbeiten. Zum anderen wird das Projekt immer noch gepflegt, während SuperC seit mehreren Jahren nicht mehr angepasst wurde und noch einen veralteten Java-Standard einsetzt.

TypeChef erfüllt alle genannten Anforderungen:

1. TypeChef unterstützt alle geforderten Sprachmittel von C. Das Projekt konzentriert sich zwar auf die Umsetzung des GNU C-Standards, dieser enthält aber alle vorher definierten Sprachmittel, welche für die Visualisierung benötigt werden.
2. TypeChef bindet Dateien entsprechend der Funktionsweise des Präprozessors ein und berücksichtigt dabei auch bedingte Dateiinklusionen.
3. Makros werden von TypeChef vor dem Parsen expandiert und dabei auch bedingte oder alternative Definitionen korrekt verarbeitet.
4. TypeChef wertet Anweisungen zur bedingten Kompilierung aus und stellt diese als boolesche Ausdrücke im Ergebnis des Parsers dar.

TypeChef nutzt einen AST mit Variabilität, welcher für die hier vorliegende Visualisierung genutzt werden soll. Für den Zugriff auf diesen gibt es zwei Möglichkeiten: entweder es wird während der Verarbeitung über eine API auf die Struktur zugegriffen oder der AST wird in einer strukturierten Form in eine Datei ausgegeben. Die Option, den AST in eine Datei auszugeben, bestand schon, jedoch handelt es sich bei der Ausgabe um keine Textdatei, sondern um eine Objektserialisierung in einem binären Format. Um die Entwicklung des Prototypen zu vereinfachen, soll eine Option zur Ausgabe des AST in Form einer Extensible Markup Language (XML)-Datei ergänzt werden. Hierdurch ist die Struktur des AST, besonders der Bedingungen, einfach zu erfassen. Eine nähere Beschreibung dieser Implementierung und des Aufbaus der XML-Datei folgt in Kapitel 4.

Durch die Verwendung von TypeChef entsteht ein wichtiger Vorteil für die Auswertung der bedingten Kompilierung: Die entsprechenden Präprozessoranweisungen werden nicht unverarbeitet in den AST übernommen, sondern es findet eine Berechnung statt, welche Kombinationen der Definitionen symbolischer Konstanten zu einer Inklusion der eingeschlossenen Quellcodebestandteile führt. Der AST enthält alle möglichen Kombinationen der Definitionen und auch alle alternativen Quellcodebestandteile, welche zu Varianten des Programms führen. TypeChef wertet die Zweige strukturiert aus und gibt die Bedingung für die Inklusion als booleschen Ausdruck an. Auch verschachtelte Bedingungen und `#undef`-Anweisungen werden berücksichtigt. Dies erleichtert die Auswertung der bedingten Kompilierung für die Visualisierung bedeutend.

Eine Einschränkung ergibt sich durch die Verwendung von TypeChef: Es kann bezüglich der bedingten Kompilierung nur Merkmale mit booleschen Werten korrekt auswerten. Das heißt, nur wenn eine symbolische Konstante darauf getestet wird, ob sie definiert ist oder nicht, wird diese Bedingung berücksichtigt. Bei der Prüfung auf einen anderen Wert, wird die Bedingung ausgewertet, aber nicht mit seiner Variabilität dargestellt. Diese Einschränkung begründen die Entwickler von TypeChef mit der Tatsache, dass ein SAT-Solver Bedingungen dieser Art nur mit hohen Performanzverlusten auswerten kann (vgl. [Kästner et al. 2011, 16]). Sie verweisen jedoch auch darauf, dass die meisten Merkmale in Linux boolesche Werte enthalten (vgl. [Kästner et al. 2011, 13]).

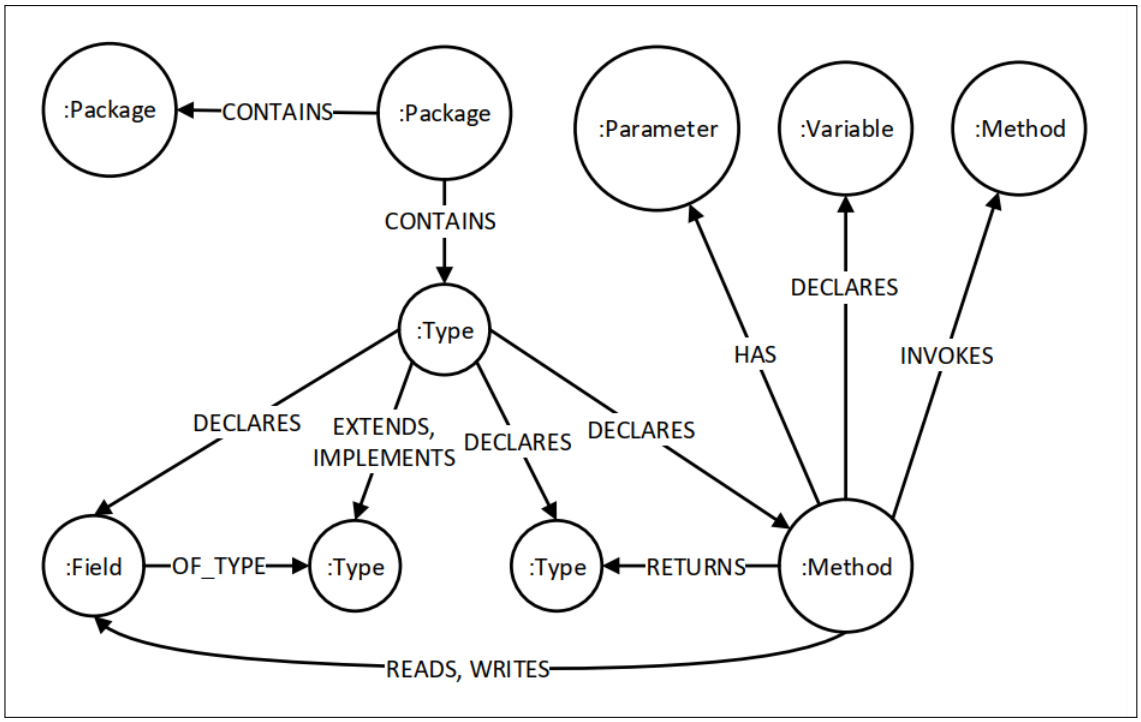

Abbildung 3.2: Graphmodell für die Speicherung von Java-Quellcode [Vogelsberg 2018]

Aufgrund dieser Einschränkung wird auch in dem hier beschriebenen Prototypen nur die Definition oder fehlende Definition einer symbolischen Konstante visualisiert. Andere Arten von Bedingungen finden in dieser Arbeit keine Berücksichtigung.

3.2.4 Entwurf eines Graphmodells für C-Quellcode

Der geparste Quellcode wird zur Visualisierung in einer Graphdatenbank abgelegt und später durch die Elemente der Visualisierungsmetapher ergänzt. Dafür muss ein Graphmodell entworfen werden, welches die extrahierten Sprachmittel in einem *labeled property graph* abbildet. Für die Programmiersprache Java existiert bereits ein solches Modell. Es findet sich in Abbildung 3.2.

Um den Generierungsprozess der Visualisierung so wenig wie möglich anpassen zu müssen, orientiert sich das Graphmodell für C-Quellcode an dem Modell für Java-Quellcode. Aufgrund der unterschiedlichen zugrundeliegenden Paradigmen – Java ist eine objektorientierte, C eine prozedurale Programmiersprache – ergeben sich aber zwangsläufig einige Unterschiede. Zudem soll das hier entwickelte Graphmodell auch die extrahierten Informationen zur bedingten Kompilierung enthalten, welche für die Visualisierung von Bedeutung sind.

Im nun folgenden Abschnitt wird die Ableitung und Anpassung des Graphmodells für C-Quellcode beschrieben. Abbildung 3.3 enthält die grafische Darstellung.

Das *Package* ist im Java-Graphmodell ein zentrales Element. Dieses kann andere Pakete und Typen enhalten. Letztere wiederum deklarieren alle anderen Elemente. Analog dazu wird im C-Graphmodell die Übersetzungseinheit (engl. *translation unit*) als zentrales

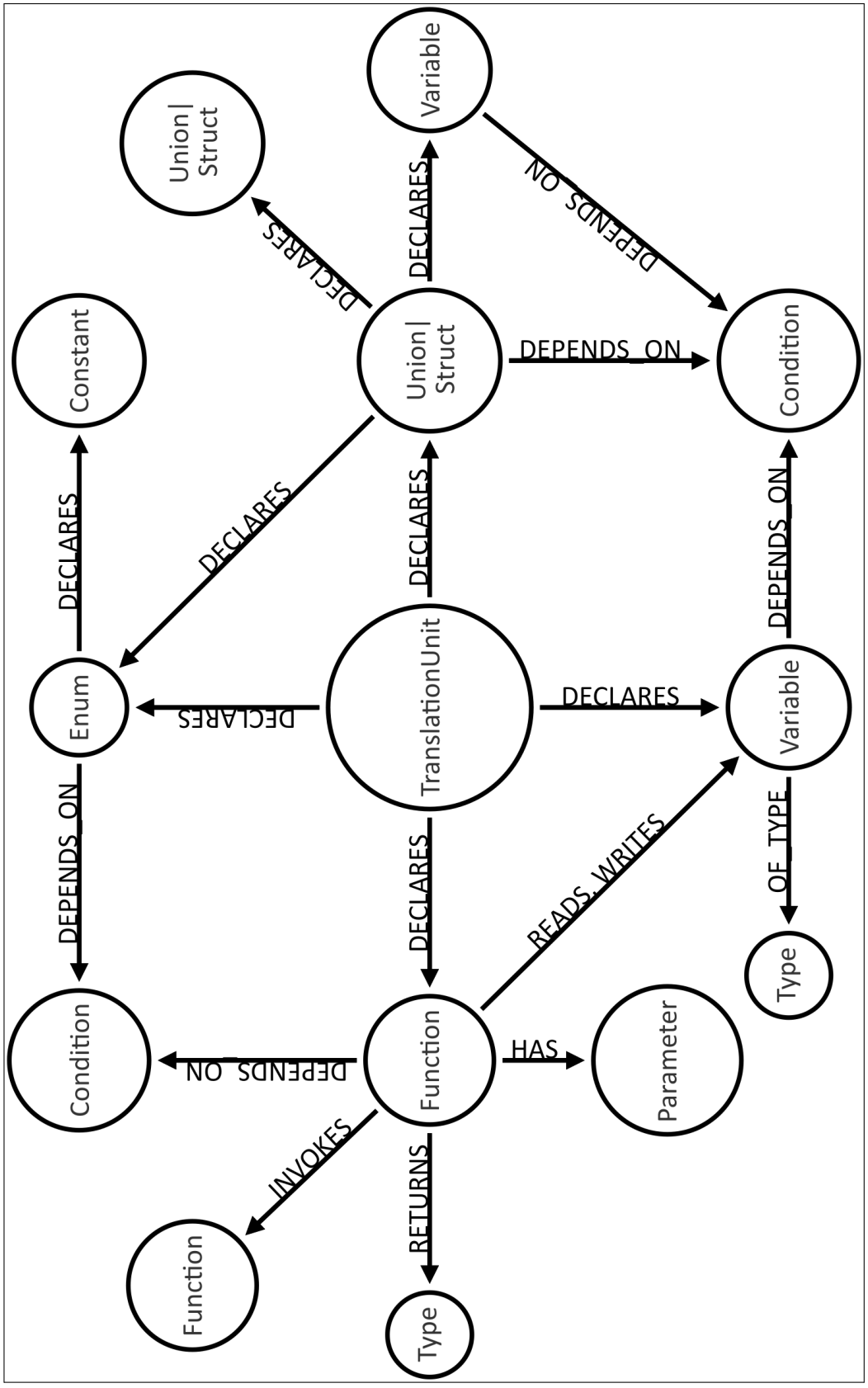

Abbildung 3.3: Graphmodell für die Speicherung von C-Quellcode

Element abgebildet, welches alle anderen Elemente deklariert, wenn auch teilweise indirekt. Eine Übersetzungseinheit stellt zwar kein explizites Sprachmittel dar, wird jedoch im C-Standard als eine Quellcodedatei mit den eingebundenen Header- und Quellcodedateien nach der Verarbeitung durch den Präprozessor beschrieben (vgl. [ISO/IEC 2017, 9]). Zudem enthält auch der von TypeChef erzeugte AST die Übersetzungseinheit. Da C keine objektorientierte Programmiersprache ist, fallen das übergeordnete Element *Type*, welches im Modell einer Klasse entspricht, sowie seine Vererbungsbeziehungen aus dem Java-Graphmodell weg. Es werden jedoch einige andere Elemente deklariert: Funktionen, Variablen, Enums, Structs und Unions.

Funktionen sind durch eine *RETURNS*-Beziehung mit dem Rückgabetyp verbunden, durch *HAS* mit ihren Parametern und die *INVOKES*-Beziehung beschreibt Funktionsaufrufe von anderen Funktionen. In der Visualisierung sollen nur globale Variablendeklarationen enthalten sein, sowie die Variablen, welche zu einer Struct oder Union gehören. Das Lesen und Schreiben dieser Variablen innerhalb von Funktionen wird durch die *READS*- beziehungsweise die *WRITES*-Beziehung abgebildet, genauso wie im Java-Graphmodell das Lesen und Schreiben von Attributen. Variablen sind von einem bestimmten Typ, welcher wie der Rückgabewert einer Funktion in einem Knoten mit dem Label *Type* gespeichert wird. Für Enums werden auch die deklarierten Konstanten in dem Graph erfasst analog zu den in Structs und Unions deklarierten Variablen. Da sich Structs und Unions von ihrem strukturellen Aufbau und ihrer Verwendung sehr ähneln, werden sie in der graphischen Darstellung in einem Knoten zusammengefasst, um das Modell übersichtlicher zu gestalten. Innerhalb von Structs und Unions können auch andere Structs und Unions sowie Enums deklariert werden.

Über die *DEPENDS-ON*-Beziehung können Bedingungen zu den Knoten *Function*, *Variable*, *Enum*, *Struct* und *Union* zugeordnet werden. Der *Condition*-Knoten dient der Speicherung von Anweisungen der bedingten Kompilierung. Er ist aus Gründen der Übersichtlichkeit zweimal in dem Modell abgebildet. Um die Bedingungen jedoch feingranularer zu speichern, kann eine Bedingung wiederum aus mehreren Unterelementen bestehen. Diese sind in einer separaten Grafik abgebildet.

Abbildung 3.4 zeigt das Graphmodell für Bedingungen im Detail. Zum besseren Verständnis enthält Listing 3.4 einen Auszug eines etwas längeren Bedingungsausdrucks (von den TypeChef-Entwicklern *feature expression* genannt). Dieser entsteht durch die Auswertung aller verschachtelten Definitionen von symbolischen Konstanten, durch Dateiinklusionen und bedingte Kompilierung, welche ebenfalls verschachtelt sein kann. TypeChef wertet diese aus und bildet daraus einen booleschen Ausdruck.

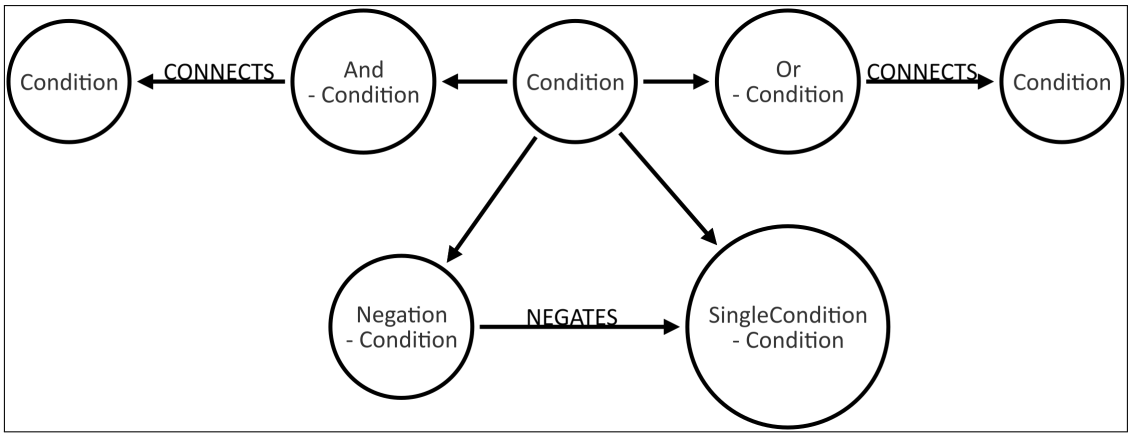

Abbildung 3.4: Graphmodell für die Speicherung der von TypeChef ausgegebenen Bedingungen

```
((definedEx(HEADER_CURL_FILEINFO_H)
  || !definedEx(__VMS)
  || definedEx(HEADER_CURL_SETUP_H)
  || definedEx(HEADER_CURL_LLIST_H)
  || definedEx(HEADER_CURL_SETUP_VMS_H)
  || !definedEx(__VAX))
  &amp;&amp; (definedEx(HEADER_CURL_FILEINFO_H)
  || !definedEx(__VMS)
  || definedEx(HEADER_CURL_SETUP_H)
  || definedEx(HEADER_CURL_LLIST_H)
  || definedEx(HEADER_CURL_SETUP_VMS_H))
```

Listing 3.4: Auszug eines von TypeChef erzeugten Bedingungsausdrucks

Das Schlüsselwort `definedEx` steht dabei für die Bedingung, wenn der in Klammern enthaltene Bezeichner definiert ist. Ein Ausrufezeichen vor dem Ausdruck verneint die Bedingung. `||` symbolisiert ein exklusives Oder und `&&` ein logisches Und. In dem Graphmodell gibt es vier Arten von Bedingungsknoten: eine *SingleCondition*, eine *Negation*, ein *And* und ein *Or*. *SingleCondition* steht für den Ausdruck `definedEx(<Bezeichner>)`. Eine Negation wird durch den *Negation*-Knoten abgebildet, eine Konjunktion durch den *And*-Knoten und ein exklusives Oder durch den *Or*-Knoten. *Or*- und *And*-Knoten verbinden mit der *CONNECTS*-Beziehung mehrere Bedingungsknoten. Diese können wiederum alle Arten von Bedingungsknoten sein. TypeChef negiert nur *SingleCondition*-Ausdrücke, dies wird durch die *NEGATES*-Beziehung abgebildet. Die im C-Graphmodell dargestellten Elemente, die über eine *DEPENDS_ON*-Beziehung verfügen, können mit einem der *Condition*-Knoten verbunden sein. Das heißt, dass die Bedingung nur eine einfache oder negierte *SingleCondition* sein kann oder eine sehr komplexe Konjunktion oder Disjunktion.

Um die Graphdatenbank mit den eben beschriebenen Modellen zu befüllen, müssen die von TypeChef erzeugten XML-Dateien durchlaufen und entsprechende Knoten und Kanten erzeugt werden. Neo4j bietet eine HTTP-API (vgl. [o.V. 2019c]) und Treiber für die Programmiersprachen Java, Python, JavaScript, C# und Go (vgl. [o.V. 2019b]). Jedoch müssen hier alle Knoten und Kanten über Cypher-Anweisungen und Prozeduren angelegt und manipuliert werden.

Eine ausgereifte, objektorientierte Schnittstelle bietet jQAssistant. Wie schon in Abschnitt 2.3 beschrieben, dient jQAssistant eigentlich der regelbasierten Analyse von Quellcode. Die Informationen für diese Analyse legt es in einer eingebetteten Neo4j-Datenbank ab. Zu diesem Zweck gibt es Scanner-Plugins für jQAssistant, welche bestimmte Strukturen, unter anderem aus Dateien, auslesen und extrahierte Informationen in Neo4j ablegen. Wird ein solches Plugin zu einem jar-Archiv gepackt und in der jQAssistant-Auslieferung eingefügt, kann dieses Plugin automatisch passende Eingabedaten erkennen und verarbeiten.

Ein großer Vorteil bei der Verwendung von jQAssistant besteht in der einfachen Abbildung des Graphmodells. Zum Zugriff auf Neo4j verwenden die Entwickler eine selbst erstellte Datenbankschnittstelle namens *eXtended Objects*. Kerngedanke dieser Technik ist, Datenobjekte als Interfaces zu definieren, um multiple Vererbung zu ermöglichen. Eigenschaften und Relationen dieser Objekte können über Annotationen sowie Getter- und Setter-Methoden definiert werden. Über ein Application Programming Interface (API) instanziiert man die Entitäten und legt damit gleichzeitig Knoten und Kanten in der Datenbank an (vgl. [Mahler 2017]).

Das in diesem Abschnitt beschriebene Graphmodell kann mit jQAssistant als Interfaces mit entsprechenden Labels, Properties und Relationen abgebildet werden. Dazu muss ein neues Plugin erstellt werden, welches die von TypeChef erzeugten XML-Dateien einliest und die Datenbankobjekte initialisiert. Auf diese Weise ist es möglich, die Struktur des Quellcodes in Neo4j abzulegen und so zur weiteren Verwendung in der Visualisierung zur Verfügung zu stellen.

3.3 Visualisierung

Nachdem in den vorhergehenden Abschnitten der Anwendungsfall entworfen und TypeChef als Werkzeug für die Verarbeitung des Quellcodes identifiziert wurde, folgt in diesem Teil der Entwurf der Visualisierung. Dazu werden bestehende Metaphern, die in Getaviz implementiert sind, auf ihre Übertragbarkeit überprüft. Wenn diese als tauglich erachtet werden, soll eine Übertragung auf die Programmiersprache C erfolgen. Die Konzipierung der Anpassung der Benutzungsoberfläche an die Darstellung der bedingten Kompilierung vervollständigt dieses Kapitel.

3.3.1 Analyse bestehender Metaphern

In Getaviz existieren vier Metaphern: die RD-Metapher, die City-Metapher, die Plant- und die Multisphere-Metapher (vgl. [Baum et al. 2017, 3]). Da jedoch in dem neu entwickelten Generator, der hier genutzt wird, bisher nur die RD- und die City-Metapher implementiert sind und diese insgesamt am weitesten ausgereift sind, sollen hier nur diese beiden Metaphern berücksichtigt werden.

Die City-Metapher stammt ursprünglich von Wettel/Lanza [2007]. Sie stellt Java-Quellcode als eine Stadt dar. Pakete werden als Bezirke – in der Abbildung Bodenplatten – abgebildet und können wiederum in anderen Paketen enthalten sein, also auf größeren Bodenplatten liegen. Die Gebäude stehen für Klassen und Interfaces. Ihre Grundfläche stellt die Anzahl der Attribute und ihre Höhe die Anzahl der Methoden dar. Abbildung 3.5 zeigt die originale City-Metapher.

Es existieren mehrere Varianten der City-Metapher in Getaviz. Neben der originalen Version von Wettel/Lanza gibt es drei weitere wichtige Varianten: die City-Bricks-Metapher (siehe Abbildung 3.6), die City-Floors-Metapher (siehe Abbildung 3.7) und die City-Panels-Metapher (siehe Abbildung 3.8).

Bei der City-Bricks-Metapher sind die Gebäude wiederum in kleine Backsteine (engl. *bricks*) eingeteilt. Diese stellen die Member einer Klasse dar, beispielsweise stehen rote Backsteine für Konstruktoren, hellgrüne Backsteine für Getter-Methoden und pinke Backsteine für Attribute mit primitivem Typ.

Die City-Floors-Metapher unterscheidet nur zwischen Methoden und Attributen: Methoden bilden sogenannte Stockwerke (engl. *floors*) in den Gebäuden und Attribute sind als kleine gelbe Schornsteine auf dem Dach der Gebäude abgebildet.

In der City-Panels-Metapher werden Methoden als eckige Stockwerke und Attribute als runde Stockwerke der Gebäude visualisiert (vgl. [Baum et al. 2017, 4; Müller 2019]).

Allen City-Metapher-Varianten gemein ist die Visualisierung von Paketen als Bezirke und Klassen als Gebäude. Hier tritt ein grundsätzliches Problem bei der Übertragung der City-Metapher zutage: sie ist für objektorientierte Sprachen entwickelt worden und enthält die Klasse und das Modul (im Fall der Programmiersprache Java das Paket) als verbindende Elemente. C bietet als vergleichbares Sprachmittel zur Modularisierung Übersetzungseinheiten. Structs und Unions bündeln, ähnlich wie Klassen, Variablen, jedoch keine Funktionen. Auch werden hiervon globale Variablen nicht erfasst. Somit würden viele freie Elemente ohne eine zusammenfassende Einheit, in der City-Metapher dem Gebäude entsprechend, übrig bleiben. Diese sind in der City-Metapher bisher nicht vorgesehen. Deshalb wird eine Verwendung dieser Metapher für den hier entwickelten Prototypen ausgeschlossen. Eventuell ist eine Anpassung der Metapher an prozedurale Programmiersprachen möglich, dies soll jedoch kein Gegenstand dieser Arbeit sein.

Von Müller/Zeckzer [2015] wird die zweite wichtige Metapher in Getaviz beschrieben: die RD-Metapher. Abbildung 3.9 zeigt ein Beispiel. In der RD-Metapher werden wich-

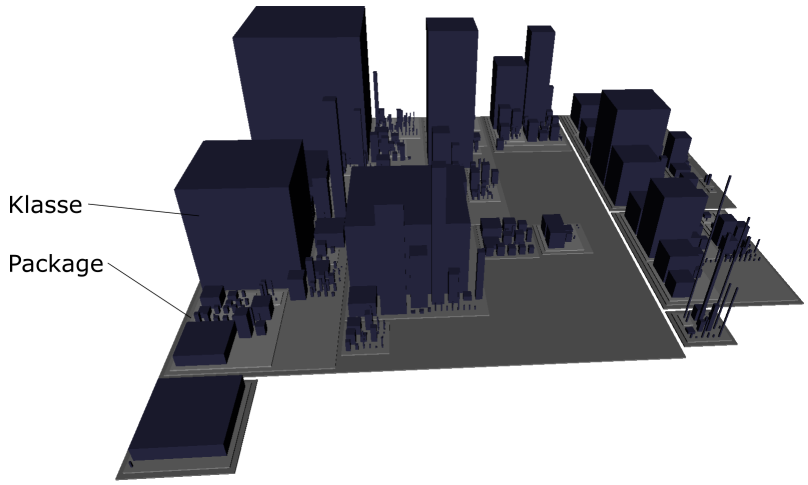

Abbildung 3.5: Die originale City-Metapher in Getaviz (in Anlehnung an [Baum et al. 2017, 3])

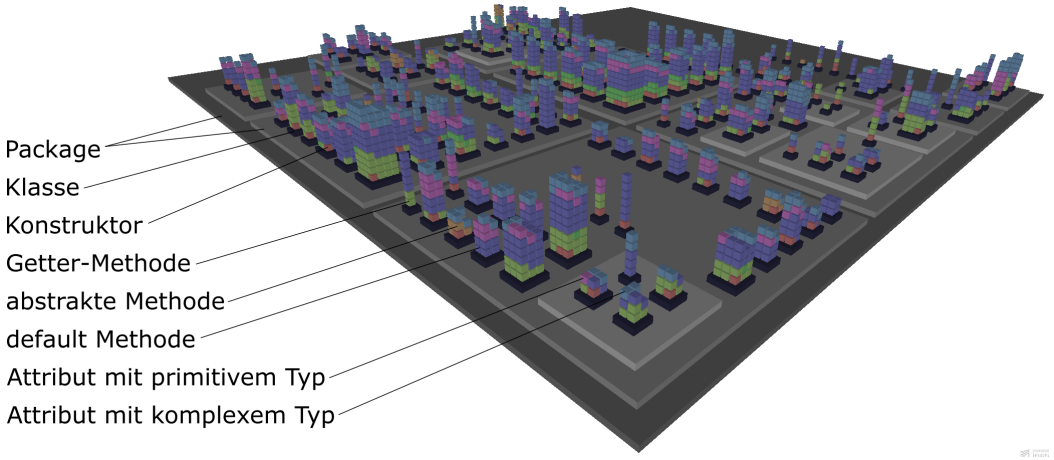

Abbildung 3.6: Die City-Bricks-Metapher von Getaviz (in Anlehnung an [Baum et al. 2017, 4])

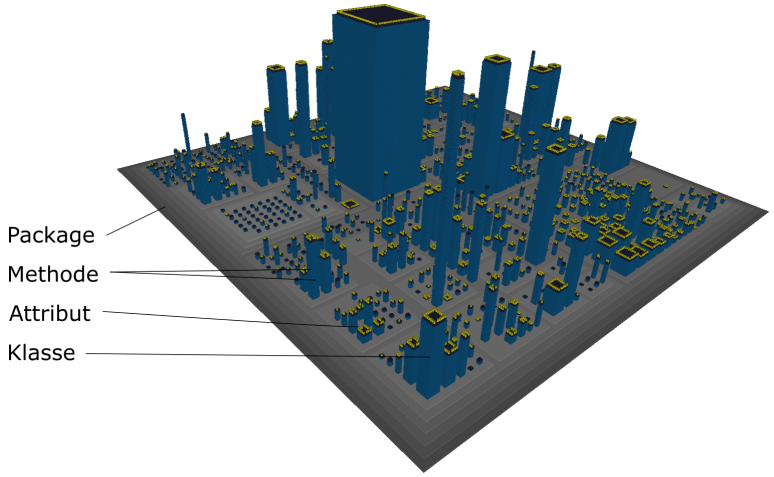

Abbildung 3.7: Die City-Floors-Metapher von Getaviz (in Anlehnung an [Baum et al. 2017, 4])

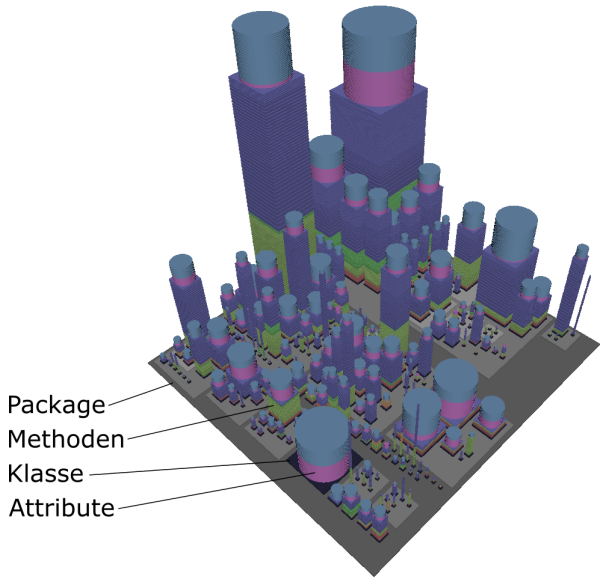

Abbildung 3.8: Die City-Panels-Metapher von Getaviz (in Anlehnung an [Baum et al. 2017, 4])

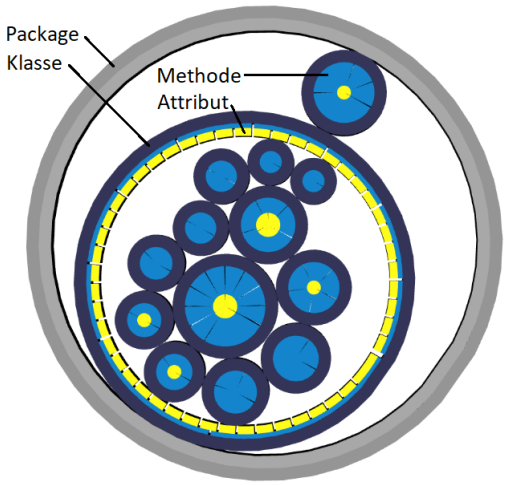

Abbildung 3.9: Die RD-Metapher, erzeugt mit Getaviz

tige Sprachelemente von Programmiersprachen als kreisförmige Glyphen (engl. *glyphs*), im folgenden auch Scheiben genannt, abgebildet. Sie sind entsprechend der Struktur der Software ineinander verschachtelt, beispielsweise enthalten Paket-Scheiben andere Paket-Scheiben oder Klassen-Scheiben. Paket-Scheiben sind grau gefärbt, mehrere einander enthaltende Pakete in verschiedenen Grautönen. Violette Glyphen stellen Klassen dar, welche wiederum hellblaue Methoden-Scheiben und gelbe Attribut-Scheiben enthalten. Die Methoden- und Attribut-Glyphen werden standardmäßig als Scheibensegmente abgebildet, welche die Klassen-Scheiben gleichmäßig unterteilen. Durch Konfigurationsmöglichkeiten können Klassen und Attribute jedoch auch als in der Klasse liegende Scheiben visualisiert werden. Es gibt auch eine dreidimensionale Variante der RD-Metapher. Hier wird die dritte Dimension beispielsweise dazu genutzt, um die zyklomatische Komplexität oder die Länge eines Methodenaufrufs als Höhe einer Methode darzustellen.

Sprachmittel	Darstellung in der RD-Metapher
Übersetzungseinheit	graue Scheibe
Struct	violette Scheibe
Variable eines Structs	gelbes Scheibensegment
Union	violette Scheibe
Variable eines Union	gelbes Scheibensegment
Enum	violette Scheibe
Enum-Konstante	gelbes Scheibensegment
Funktion	hellblaues Scheibensegment
globale Variable	gelbes Scheibensegment

Tabelle 3.1: Übertragung der RD-Metapher auf die Sprachmittel des C-Quellcodes

Bei der RD-Metapher fällt, wie bei der zuvor besprochenen City-Metapher, auf, dass Pakete und Klassen zentrale Strukturelemente in der Visualisierung bilden. Da die Disks in der Abbildung jedoch beliebig verschachtelt und angeordnet werden können, können Pakete und Klassen leicht durch die Übersetzungseinheit im C-Quellcode ersetzt werden. Hierdurch wird die Visualisierung nur insoweit verändert, dass die Glyphen für Funktionen und globale Variablen direkt in die Übersetzungseinheits-Glyphe eingefasst werden und es keine weitere zusammenfassende Glyphe gibt. Structs, Unions und Enums werden analog zu Klassen in Java-Quellcode als violette Scheibe visualisiert. Tabelle 3.1 stellt allen visualisierten Sprachmittel ihre Darstellung in der RD-Metapher gegenüber.

3.3.2 Darstellung der Variabilität in der Benutzungsoberfläche

In vorhergehenden Abschnitt wurde diskutiert, welche in Getaviz implementierten Visualisierungsmetaphern für die Übertragung auf C-Quellcode geeignet sind. Die City-Metapher stellte sich aufgrund der zentralen Rolle von Paketen und Klassen als ungeeignet heraus, die RD-Metapher wurde jedoch als übertragbar bewertet und eine Übertragung vorgenommen. Diese umfasst aber bisher nur die Sprachmittel der Programmiersprache C, nicht die Sprachmittel des C-Präprozessors. Die von diesen eingeführte Variabilität soll in dieser Arbeit aber ebenfalls visualisiert werden. In diesem Abschnitt soll die Darstellung dieser Variabilität in der Visualisierung und damit einhergehend die Benutzungsoberfläche entworfen werden.

Die bedingte Kompilierung, welche durch den Einsatz des C-Präprozessors entsteht, führt dazu, dass je nach Zustand der abgefragten symbolischen Konstanten bestimmte Elemente des Quelltextes in das Programm eingefügt oder ausgeschlossen werden. Der hier konstruierte Prototyp soll deshalb ausgehend von dem Zustand symbolischer Konstanten die Bedingungen für die Einbindung von Sprachmitteln in das fertige Programm prüfen

und die Visualisierung entsprechend anpassen. Bei Änderung des Zustandes werden dann gegebenenfalls Elemente ein- oder ausgeblendet.

Wie in Abschnitt 3.2.3 beschrieben, stellt TypeChef nur Bedingungen dar, die die Definition oder Nichtdefinition von symbolischen Konstanten prüfen. Aus diesem Grund wird auch die Benutzungsoberfläche in dieser Arbeit auf die Definition oder Nichtdefinition von symbolischen Konstanten beschränkt. Elemente, welche zwei Werte annehmen können, können in grafischen Oberflächen mithilfe von Checkboxen dargestellt werden. Hierdurch ist der Nutzer in der Lage, eine gewünschte Kombination symbolischer Konstanten zu definieren und die Auswirkungen auf die Struktur des Quellcodes zu betrachten.

Für die Darstellung des Zustands der symbolischen Konstanten ergeben sich grundsätzlich zwei Lösungen: Entweder werden von TypeChef ausgewertete Bedingungen angezeigt, welche als Ganzes als erfüllt definiert werden, oder es werden nur einzelne Makros angezeigt, welche beliebig kombiniert werden können. Bei der ersten Lösung besteht der Vorteil darin, dass der Benutzer der Visualisierung sicher sein kann, dass eine Definition eine Veränderung der Visualisierung zur Folge hat. Es muss nicht erst die richtige Kombination für das Ein- oder Ausblenden eines Elements gefunden werden. Allerdings lässt sich als großer Nachteil anmerken, dass bei vielen Merkmalen in einem Programm sehr viele Kombinationen entstehen können, wodurch eine lange und unübersichtliche Liste entsteht. Hier hat die zweite Variante ihre Stärke, indem die Länge der Liste auf die Anzahl der Merkmale beschränkt ist. Auf der anderen Seite besteht für den Benutzer der Nachteil, dass er möglicherweise Auswirkungen von Zustandsänderungen nicht direkt in der Visualisierung sieht, was für Verwirrung sorgen kann.

Nach gründlicher Abwägung fiel die Entscheidung zugunsten der Anzeige einzelner symbolischer Konstanten. Die Checkboxen werden in einem eigenen Fenster am Rand der Oberfläche angezeigt. Dort gibt es bei Getaviz bereits einige zusätzliche Ansichten, beispielsweise einen *Package Explorer* und eine Quellcodeansicht. Abbildung 3.10 und Abbildung 3.11 zeigen den Entwurf mit derselben Visualisierung, aber in unterschiedlichen Zuständen. In der ersten Abbildung ist eine ausgewählte Struct zu sehen, hervorgehoben durch die rote Farbe. Auf der rechten Seite findet sich in der Quellcodeanzeige der dazugehörige Quellcode. Oben links ist der sogenannte *Macro Explorer* platziert, in welchem die im Quellcode verwendeten symbolischen Konstanten durch Anhaken einer Checkbox definiert werden können. Dadurch wird die Visualisierung in der Mitte der Abbildung entsprechend modifiziert. Den Zustand, wenn das hier vorhandene Makro nicht definiert ist, stellt die zweite Abbildung dar. Einige Elemente der Visualisierung sind transparent. Das bedeutet, dass sie bei der im Macro Explorer gewählten Konfiguration nach der Verarbeitung durch den Präprozessor nicht mehr im Quellcode enthalten wären.

In Abschnitt 3.1 wurden zwei Fragen definiert, welche der Nutzer mithilfe der Visualisierung beantworten können soll:

1. Welche Auswirkung hat die Definition einer symbolischen Konstante oder einer Gruppe von symbolischen Konstanten auf die Struktur des Programms?

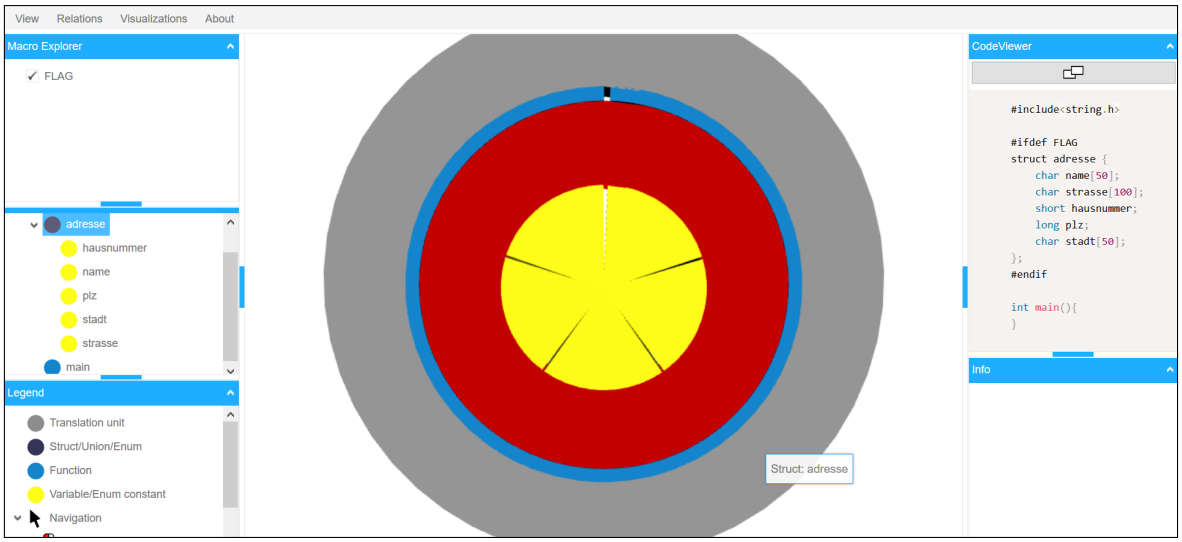

Abbildung 3.10: Entwurf der Benutzungsoberfläche mit ausgewählter Glyphe

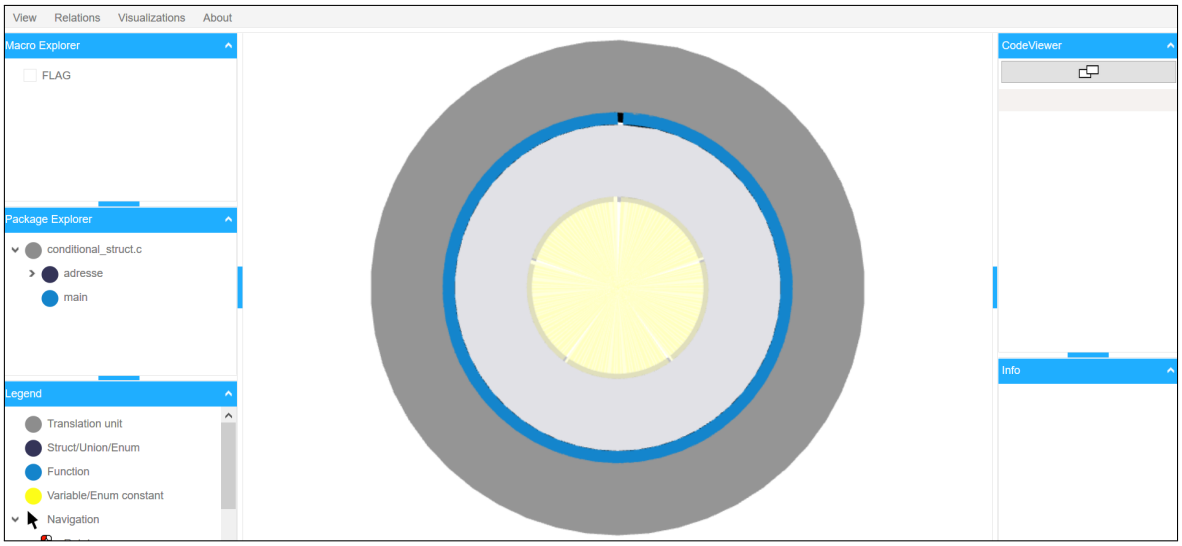

Abbildung 3.11: Entwurf der Benutzungsoberfläche mit nicht definierter symbolischer Konstante

2. Welche Elemente sind im Quellcode enthalten, wenn eine bestimmte Gruppe von symbolischen Konstanten definiert ist?

Diese können mithilfe der hier beschriebenen Visualisierung beantwortet werden. Auch das Ziel der Aufgabe, einen Überblick über das Softwaresystem zu bekommen, kann mit dem entworfenen Artefakt prinzipiell erreicht werden.

3.4 Überblick über den Generierungsprozess

Zuletzt soll der geplante Prozess der Erzeugung einer Softwarevisualisierung für C-Quellcode überblicksartig dargestellt werden. Dazu findet sich in Abbildung 3.12 eine Darstellung der notwendigen Verarbeitungsschritte. Das blaue Element enthält die Struktur und den Inhalt der Daten in diesem Schritt. Der weiß hinterlegte Kasten zeigt das

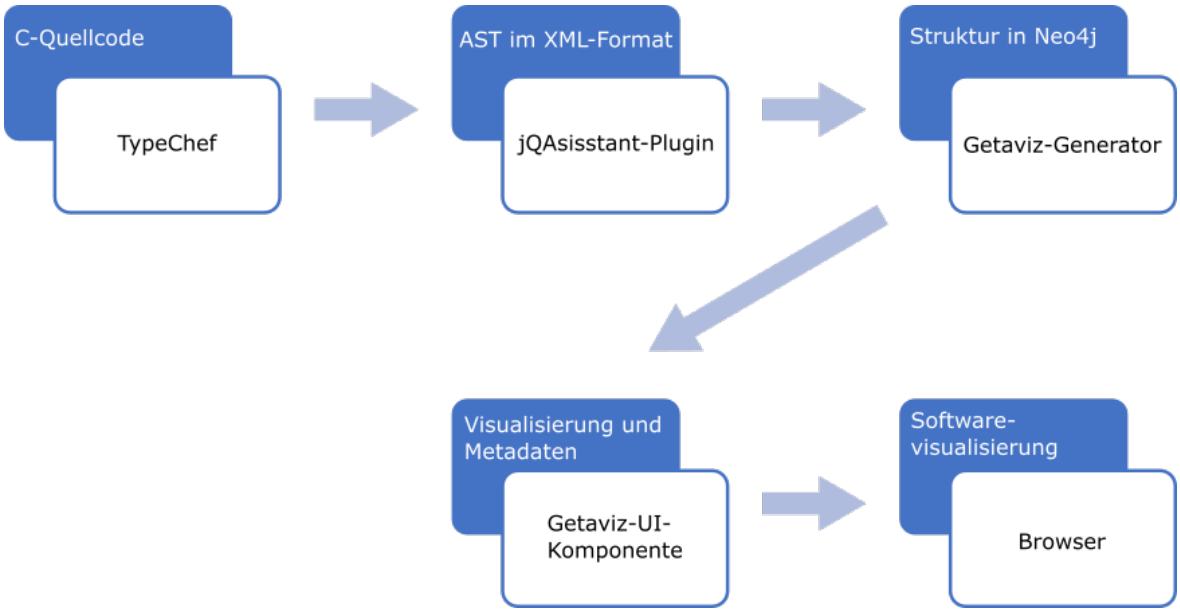

Abbildung 3.12: Gesamtprozess der Visualisierungserzeugung für C-Quellcode

Werkzeug, welches in diesem Schritt zum Einsatz kommt. Dieses verarbeitet die in dem jeweiligen Schritt vorhandenen Daten und erzeugt die Eingabe-Daten für den nächsten Verarbeitungsschritt.

Zunächst werden die C-Quelldateien in Form von Quell- und Headerdateien an TypeChef übergeben. Das Werkzeug inkludiert automatisch die durch die Präprozessoranweisungen eingebundenen Dateien und erzeugt einen AST mit der Struktur des Quellcodes. Durch Angabe der entsprechenden Kommandozeilenoption gibt TypeChef diesen AST in Form einer XML-Datei aus.

Diese Datei dient wiederum als Eingabe für das zu entwickelnde jQAssistant-Plugin. Das Plugin ist speziell für die Verarbeitung dieser Struktur konzipiert und erzeugt beim Durchlaufen der Datei entsprechende Knoten und Relationen in einer Neo4j-Datenbank. Die Struktur des resultierenden Graphmodells entspricht dem in Abschnitt 3.2.4 entworfenen Graphmodell.

Die Graph-Datenbank wird von dem Getaviz-Generator eingelesen und anhand der gespeicherten Sprachmittel wird eine Visualisierung entsprechend der gewählten Metapher erzeugt. Dabei werden in einem ersten Schritt alle Visualisierungselemente generiert und in der Datenbank ergänzt. Anschließend wird das Layout berechnet und bei Verwendung von X3DOM[1] wird am Ende des Prozesses eine .x3d-Datei generiert, welche die notwendigen Informationen für die Darstellung des Modells im Browser enthält. Zusätzlich erzeugt der Generator eine .json-Datei, welche Metadaten zu den Elementen gespeichert hat, beispielsweise den Namen und den Typ der Elemente.

Der vierte Schritt besteht im Einlesen von Modell und Metadaten und der Erzeugung der Komponenten für die Browserdarstellung. Dies wird von der Getaviz-UI-Komponente

[1] https://www.x3dom.org/

durchgeführt. Sie initialisiert alle notwendigen Controller, liest die Metadaten der Elemente und füllt die Komponenten der Benutzungsoberfläche.

Zuletzt stellt der Browser die erzeugte Softwarevisualisierung in der Benutzungsoberfläche dar. Das Modell kann dabei durch verschiedene Komponenten, wie Filter- oder Auswahlmöglichkeiten, manipuliert werden und der Nutzer hat die Möglichkeit, auf verschiedene Arten mit der Visualisierung zu interagieren. Diese Möglichkeiten stellt die Getaviz-UI-Komponente durch JavaScript-Code zur Verfügung.

Für den hier entwickelten Prototypen ist ein manueller Aufruf der verschiedenen Werkzeuge im Erzeugungsprozess vorgesehen. Denkbar ist jedoch auch eine Verknüpfung durch ein einfaches Skript oder die Entwicklung eines Java-Programms, welches den Gesamtprozess steuert. Dadurch würde die Erzeugung der Softwarevisualisierung beschleunigt und die Nutzung durch einen Endanwender vereinfacht. Diese und ähnliche Weiterentwicklungen erläutert Abschnitt 5.3 kurz.

4 Implementierung

Nach der Konzeption des Prototypen im letzten Kapitel folgt nun eine Beschreibung der Implementierung. Diese gliedert sich nach der Verarbeitungsreihenfolge des Quellcodes. Zunächst wird die Vorverarbeitung des Quellcodes durch TypeChef geschildert. Anschließend folgt eine Beschreibung des jQAsisstant-Plugins zum Parsen der Ausgabe von TypeChef. Der nächste Schritt in der Verarbeitung ist die Generierung der Visualisierung durch Getaviz. Auch hier mussten einige Anpassungen vorgenommen werden, welche in Abschnitt 4.3 beschrieben werden. Im letzten Teil des Kapitels stehen das Testen und die Evaluation des Prototypen im Mittelpunkt.

4.1 Vorverarbeitung des Quellcodes

Für die hier beschriebene Softwarevisualisierung ist eine Auswertung der Anweisungen zur bedingten Kompilierung notwendig. Deshalb fiel die Entscheidung darauf, das Werkzeug TypeChef zu verwenden, um den Quellcode vorzuverarbeiten. TypeChef wertet die Definition von symbolischen Konstanten aus und analysiert für alle C-Sprachmittel im Quellcode die Bedingungen für die Inklusion durch den Präprozessor. Der erzeugte AST soll als Basis für die Visualisierung dienen. Um die Entwicklung des Parsers zu erleichtern, soll der AST in Form einer XML-Datei ausgegeben und in dieser Form weiterverarbeitet werden. Dies wird im folgenden Abschnitt geschildert.

4.1.1 Anpassung von TypeChef

TypeChef baut für die Analyse von Quellcode intern einen AST auf. Dieser kann jedoch eigentlich nur als Objektserialisierung in einem binären Format ausgegeben werden. Das Verhalten von TypeChef kann über Kommandozeilenargumente gesteuert werden. Es bestehen bereits einige Optionen, beispielsweise die ausschließliche Ausführung des Lexers ohne Parsen des Quellcodes oder die Erzeugung von Feature Models (vgl. [Kästner/Liebig 2015]).

Für die Ausgabe der XML-Datei wird eine Option namens `--serializeToXML` ergänzt. Diese muss zunächst in der Datei `Frontend/src/main/scala/de/fosd/typechef/options/FrontendOptions.java` eingefügt werden. Anschließend wird diese Option in der Datei `Frontend/src/main/scala/de/fosd/typechef/Frontend.scala` bei der Verarbeitung der Aufrufargumente abgefragt. Wenn die Option übergeben wurde, wird eine Funktion namens `serializeToXML()` aufgerufen, welche die Ausgabe des ASTs als XML-Datei auslöst.

Die Datei `Frontend.scala` ist in der Programmiersprache Scala geschrieben. Dadurch, dass sowohl Scala als auch Java zu Bytecode kompiliert und auf der Java Virtual Machine (JVM) ausgeführt werden, können in Scala-Quellcode auch Java-Bibliotheken verwendet werden. Diese Eigenschaft wird genutzt, um die XML-Datei zu erzeugen. Hierfür

wird die Open-Source-Java-Bibliothek XStream verwendet. XStream hat den Vorteil, dass es keine Java Beans zur Serialisierung benötigt, was beispielsweise bei Nutzung der Klasse Java Persistance Beans der Fall ist. Dadurch müssen die zu serialisierenden Klassen nicht modifiziert werden (vgl. [Ullenboom 2014, 661ff]).

Da XStream keine XML-Deklaration zu Beginn des Dokumentes einfügt, muss diese zunächst erstellt werden. Sie wird einem zuvor erstellten `FileOutputStream` übergeben, in welchen anschließend auch der serialisierte AST eingefügt wird. Die resultierende XML-Datei ist nach der ursprünglich übergebenen C-Quelldatei benannt und trägt die Dateiendung `.ast`.

4.1.2 Aufbau des Abstract Syntax Tree

```
1 <de.fosd.typechef.conditional.Opt>
2   <condition class="de.fosd.typechef.featureexpr.sat.And"
        resolves-to="de.fosd.typechef.featureexpr.
        FeatureExprSerializationProxy">
3     <fexpr>(definedEx(FLAG2) &amp;&amp; !definedEx(FLAG))</fexpr>
4   </condition>
5 <entry class="de.fosd.typechef.parser.c.Declaration">
```

Listing 4.1: Ausschnitt aus dem AST im XML-Format

Zum besseren Verständnis des Parsers wird hier der Aufbau des von TypeChef erzeugten ASTs im XML-Format kurz erläutert. Ein Ausschnitt dieses ASTs ist in Listing 4.1 zu sehen. Das Wurzel-Element der XML-Datei ist die Übersetzungseinheit. Hierin sind alle Elemente enthalten. Jedes Sprachmittel ist von einem XML-Element mit dem Bezeichner `de.fosd.typechef.conditional.Opt` umgeben. Dieses hat als erstes Kind-Element immer eine `condition`, welche wiederum oftmals das Element `fexpr` enthält. Durch das Konstrukt wird beschrieben, ob ein Sprachmittel in eine Anweisung zur bedingten Kompilierung eingeschlossen ist. Die von TypeChef ermittelte Bedingung steht als Text zwischen den `fexpr`-Tags. Wenn dort nur eine 1 zu finden ist, bedeutet dies, dass das nachfolgende Sprachmittel keiner Bedingung unterliegt, also in jeder Variante des Programms enthalten ist. Andernfalls ist hier eine Bedingung enthalten, ein Beispiel einer solchen Bedingung ist in Listing 3.4 zu sehen.

Viele Sprachmittel werden in dem XML-Dokument mit dem Element `entry` ausgezeichnet, das Attribut `class` bestimmt dann die Art des Sprachmittels näher. Unter anderem werden so Deklarationen, Funktionsdefinitionen und Typbezeichner eingeleitet. Problematisch ist in diesem Zusammenhang, dass alle Deklarationen durch die Klasse `de.fosd.typechef.parser.c.Declaration` gekennzeichnet sind. Structs, Unions und Enums werden anschließend speziell als solche gekennzeichnet, aber Variablendeklarationen

und Funktionsdeklarationen ohne Parameter lassen sich nicht unterscheiden. Hier findet sich eine potenzielle Fehlerquelle bei der Visualisierung.

Zu einigen Sprachmitteln enthält der AST auch die Angabe, aus welcher Quelldatei es stammt und in welcher Zeile und Spalte es sich dort befindet.

Durch die Angabe des vollqualifizierten Typs der Klassen werden die XML-Dateien oft sehr groß. Abhilfe könnten hier Aliasdefinitionen bei der Erzeugung der XML-Datei verschaffen. Dies ist mit der Funktion `public void alias(java.lang.String name, java.lang.Class type)` möglich und wird als mögliche Verbesserung für eine künftige Weiterentwicklung vorgeschlagen.

4.2 Das jQAssistant-Plugin

Nach der Ausgabe der XML-Datei durch TypeChef folgt als nächster Schritt in der Erzeugung der Visualisierung das Parsen mithilfe von jQAssistant. Dafür muss ein Plugin erstellt werden, welches die Dateien durchläuft und dabei Knoten und Kanten in der Graphdatenbank erzeugt. Der Aufbau dieses Plugins wird in den folgenden Abschnitten beschrieben.

4.2.1 Aufbau und grundsätzliche Funktionsweise

Das hier entwickelte jQAssistant-Plugin ist in Java geschrieben und als Maven[1]-Projekt angelegt. Maven ist ein Werkzeug zur Verwaltung von Java-Projekten. Es kümmert sich um die Abhängigkeiten des Plugins und wird auch eingesetzt, um die jar-Datei des Plugins zu erstellen. Ein mit Maven erzeugtes Projekt hat einen standardisierten Aufbau, diesem folgt auch das C-Scanner-Plugin. Zusätzlich besitzt es den Ordner `src/main/antlr4`, welcher die ANTLR-Quelldateien enthält. Auf den Einsatz von ANTLR wird in Abschnitt 4.2.4 eingegangen.

Um das Modell von der Anwendungslogik des Scanners zu trennen, sind diese in zwei unterschiedlichen Packages untergebracht: das Package `org.jqassistant.contrib.plugin.c.api.model` enthält die *Descriptors* zur Beschreibung des Modells, das Package `org.jqassistant.contrib.plugin.c.impl.scanner` kapselt die Logik des Scanners inklusive der Einstiegsklasse.

Die Einstiegsklasse eines jQAssistant-Plugins muss die abstrakte Klasse `AbstractScannerPlugin<I, D extends Descriptor>` erweitern. In dem hier beschriebenen Plugin heißt diese Einstiegsklasse `CAstFileScannerPlugin`. Die beiden generischen Typ-Parameter der Klasse sind hier vom Typ `FileResource` und `CAstFileDescriptor`. Daraus ergibt sich, dass das Plugin Eingaben in Form von Dateien verarbeiten kann und einen *Descriptor* vom Typ `CAstFileDescriptor` erzeugt.

[1] https://maven.apache.org/

Die Klasse `CAstFileScannerPlugin` enthält die Logik des Plugins, also Anweisungen zur Verarbeitung der XML-Datei und zur Erzeugung der Daten in der Datenbank.

Für jQAssistant gibt es bereits ein Plugin zur Verarbeitung von XML-Dateien[2], jedoch generiert dieses lediglich allgemeine Datenstrukturen in der Datenbank, welche die XML-Datei abbilden. Auch bei Erweiterung dieses Plugins würde die gesamte Struktur der XML-Datei in der Datenbank gespeichert, was für den Anwendungsfall nicht notwendig ist. Aus diesem Grund fiel die Entscheidung gegen die Erweiterung dieser Klasse und für die Erweiterung der allgemeineren Klasse `AbstractScannerPlugin`.

Um das Graphmodell abzubilden, nutzt jQAssistant ein Konzept namens `eXtended Objects`. Dieses stellt eine Schnittstelle zwischen Java-Anwendungen und Datenbanken zur Verfügung, zunächst unabhängig von einer konkreten Datenbank. Es existieren Implementierungen für verschiedene Datenbanken, bei jQAssistant kommt eine Implementierung für Neo4j zum Einsatz. Mithilfe von sogenannten Descriptors werden Knoten oder Kanten abgebildet. Die Descriptors sind Interfaces, welche automatisiert in Klassen übersetzt und instanziiert werden. Der Vorteil von Interfaces gegenüber Klassen ist, dass hierdurch multiple Vererbung umgesetzt werden kann, was Java für Klassen nicht bietet. Durch die Verwendung der Descriptors kann der Graph in der Neo4-Datenbank durch Java-Code manipuliert werden (vgl. [Mahler 2017]). Auf einige Descriptors wird in Abschnitt 4.2.2 näher eingegangen.

Damit ein Plugin in jQAssistant integriert werden kann, muss es einen sogenannten *plugin descriptor* besitzen. Dies ist eine XML-Datei, welche den Namen und eine Beschreibung des Plugins enthält, sowie alle verwendeten Descriptors.

Nach dem Erzeugen der `jar`-Datei des Plugins wird diese in den `plugin`-Ordner der Auslieferung von jQAssistant kopiert. Mithilfe von Kommandozeilenbefehlen kann jQAssistant aufgerufen und eine zu analysierende Ressource übergeben werden. Die `accept()`-Methode des jQAssistant-Plugins beschreibt, welche Art von Ressourcen von diesem Plugin verarbeitet werden kann. Das C-Scanner-Plugin schränkt diese auf Dateien mit der Dateienendung `.ast` ein. Dadurch wird die Zuständigkeit dieses Plugins automatisiert an der Art der übergebenen Datei erkannt und das Plugin aufgerufen.

JQAssistant unterstützt durch die Klasse `AbstractPluginIT` das Schreiben von Unittests. Sie bietet die Infrastruktur, um in einer Neo4j-Datenbank Objekte anzulegen und zu manipulieren. Durch den Aufruf des Scanner-Plugins können Test-Dateien geparst und mithilfe des Testframeworks JUnit[3] automatische Unittests durchgeführt werden. Für das C-Scanner-Plugin wurden stets vor der Entwicklung einer neuen Komponente oder Funktionalität Tests geschrieben und der entwickelte Code auf deren erfolgreiche Durchführung getestet.

[2] http://buschmais.github.io/jqassistant/doc/1.6.0/#_xml_plugin
[3] https://junit.org/junit5/

4.2.2 Wichtige Descriptors und Relations

Die Descriptors des C-Scanner-Plugins leiten sich aus dem Graphmodell in Abbildung 3.3 und Abbildung 3.4 her. Für alle dort dargestellten Knoten existiert ein Descriptor, beispielsweise ein `TranslationUnitDescriptor` oder ein `StructDescriptor`. Selbstverständlich gibt es nur einen `VariableDescriptor`, auch wenn ein Knoten mit dem Label `Variable` mehrfach in Abbildung 3.3 existiert. Alle Descriptors, welche ein Sprachmittel der Programmiersprache C abbilden, erweitern den `CDescriptor`, welcher diesen Knoten ein Label mit der Aufschrift *C* hinzufügt. Dadurch können diese Knoten von Präprozessoranweisungen, anderen Programmiersprache und von Knoten der Visualisierung unterschieden werden. Auch Relationen können mithilfe von Descriptors abgebildet werden, dies geschieht beispielsweise durch den `DependsOnDescriptor`, welcher die *DEPENDS_ON*-Beziehung zwischen den Knoten enthält. Durch Hinzufügen des `NamedDescriptors` erhalten einige Knoten die Property *name*. In dieser werden beispielsweise Variablennamen, Namen von Structs, Parametern und so weiter gespeichert.

Auch die Knoten der Abbildung 3.4 sind durch Descriptors abgebildet. Die Interfaces `NegationDescriptor`, `SingleConditionDescriptor`, `OrDescriptor` und `AndDescriptor` erweitern den `ConditionDescriptor`, was bei der Speicherung der Bedingungen hilft, da hierdurch alle diese Knoten durch eine `DEPENDS_ON`-Beziehung verbunden werden können.

Das Package `org.jqassistant.contrib.plugin.c.api.model` enthält auch Java-Klassen, beispielsweise die Klassen `Condition` oder `FunctionCall`. Diese dienen als Hilfsklassen zur vorübergehenden Speicherung von Informationen. Sie sind notwendig, wenn beim Durchlaufen der XML-Datei das Anlegen von Knoten und Kanten aufgrund von fehlenden Informationen noch nicht möglich ist, aber die extrahierten Informationen in einer aufbereiteten Form gespeichert werden sollen. Die genaue Funktionsweise des Plugins wird im folgenden Abschnitt beschrieben.

4.2.3 Verarbeitung der XML-Datei

Die Klasse `CAstFileScannerPlugin` enthält die Logik zum Parsen der von TypeChef erzeugten XML-Dateien. Ullenboom [2014, 677ff] unterscheidet vier Verarbeitungstypen für XML-basierte Daten in Java:

- DOM-orientierte APIs lesen zu Beginn der Verarbeitung das gesamte Dokument ein, wodurch beispielsweise Beziehungen zwischen Elementen einfach ermittelt werden können oder eine Sortierung der Elemente erleichtert wird.
- Bei Pull-APIs wie StAX wird das XML-Dokument sukzessiv durchlaufen. Hierbei muss das nächste Element aktiv angefordert werden.

```
1  try {
2    streamReader = inputFactory.createXMLStreamReader(source);
3    while (streamReader.hasNext()) {
4      int eventType = streamReader.next();
5      switch (eventType) {
6        case XMLStreamConstants.START_ELEMENT:
7          handleStartElement();
8          break;
9        case XMLStreamConstants.END_ELEMENT:
10         handleEndElement();
11         break;
12       default:
13         break;
14     }
15   }
16 } catch (Exception e){
17   logger.error(e.getMessage());
18 }
```

Listing 4.2: Ausschnitt aus der `scan()`-Methode der Klasse `CAstFileScannerPlugin`

- Push-APIs wie SAX durchlaufen das XML-Dokument ebenfalls sukzessiv, jedoch übernimmt dies der Parser und ruft beim Auftreten von bestimmten Ereignissen Methoden auf.
- Mapping-APIs bilden die XML-Datei auf ein Java-Objekt ab, mit welchem der Nutzer anschließend arbeiten kann.

Zur Verarbeitung der von TypeChef erzeugten XML-Datei muss die Auswahl zwischen diesen getroffen werden.

Mapping-APIs fallen insofern aus der Betrachtung, da hier Java-Klassen mit einer passenden Struktur vorliegen müssen, im Falle von JAXB Java Beans. Auch DOM-orientierte APIs wurden für den Prototypen ausgeschlossen, da hier das gesamte XML-Dokument in den Arbeitsspeicher geladen wird, was bei sehr großen XML-Dokumenten problematisch ist. Bei den verwendeten Testbeispielen wurde im ungünstigsten Fall eine 30 Zeilen lange C-Quelldatei als XML-Datei über 2100 Zeilen lang. Dies lässt bei realen Programmen sehr große XML-Dateien erwarten.

Aus diesem Grund wird ein Stream-Ansatz bevorzugt, wie er bei Pull- und Push-APIs zum Einsatz kommt. Aufgrund der hohen Performanz und der einfachen Verwendung wurde die Pull-API StaX gewählt und hiervon der `XMLStreamReader`, da er etwas effizienter arbeitet als der `XMLEventReader` (vgl. [Ullenboom 2014, 677ff]). Listing 4.2 zeigt einen Ausschnitt aus der `scan()`-Methode des Plugins, welche den Einstiegspunkt zum Parsen einer Datei darstellt. Das Objekt `streamReader`, eine Instanz der Klasse `XMLStreamReader`, gibt beim Aufruf der Funktion `next()` das nächste Ereignis (engl.

event) beim Durchlaufen des XML-Dokumentes zurück. Die Ereignisse vom Typ `START_ELEMENT` und `END_ELEMENT` werden weiter verarbeitet.

In der beim Auftreten eines Startelementes aufgerufenen Methode `handleStartElement()` werden zunächst die Namen aller Startelemente in einem `ArrayDeque<Object>` abgelegt. Beim Auftreten des entsprechenden Endelementes werden diese wieder entfernt. Das Verhalten, welches von der speichernden Datenstruktur benötigt wird, ist das einer Last-in-First-out (LIFO)-Datenstruktur, weil hierdurch die Schachtelung der XML-Elemente korrekt erfasst werden kann. Dadurch ist es wiederum möglich, Eltern-Kind-Beziehungen zwischen den Elementen zu erkennen, was häufig für das korrekte Verständnis des Quellcodes notwendig ist. In Java bildet die Klasse `Stack` eine solche Datenstruktur ab, jedoch kann auch eine `Deque` genutzt werden, was die Dokumentation zum Java-Standard 8 ausdrücklich empfiehlt (vgl. [o.V. 2018]).

Bei der Verarbeitung der Start-Elemente werden entsprechende Descriptor-Objekte erzeugt, wodurch jQAssistant automatisch Knoten in der Graphdatenbank anlegt. Diese werden beim weiteren Durchlaufen des XML-Dokumentes ergänzt, beispielsweise um Namen, Typen oder Beziehungen zu anderen Elementen. Um zu wissen, wann ein Sprachmittel begonnen hat und wo es endet, wird hierfür der entsprechende Descriptor in der Deque abgelegt. Sobald dieser oben auf der Deque liegt und ein End-Element auftaucht, kann davon ausgegangen werden, dass die Definition des Sprachmittels beendet ist. Dann werden noch einige grundsätzliche Arbeiten ausgeführt. Diese variieren leicht je nach Element: es werden die Bedingungen für ein Element geprüft, also ob dieses Teil der bedingten Kompilierung ist. Bei einigen Sprachmitteln wird geprüft, ob diese schon in der entsprechenden Übersetzungseinheit deklariert wurden und ob hier vielleicht nur die anschließende Definition vorliegt. Am Ende wird das Element zu dem deklarierenden Element hinzugefügt. Häufig ist dieses eine Übersetzungseinheit.

Teilweise ist der Aufbau des XML-Dokumentes problematisch. Häufig kann beim Beginn eines Sprachmittels noch nicht das genaue Sprachmittel bestimmt werden. Zum Beispiel werden Deklarationen zunächst allgemein mit dem Attributwert `de.fosd.typechef.parser.c.Declaration` gekennzeichnet. Erst durch einige Kindelemente kann die Art der Deklaration bestimmt werden, also ob es sich um eine Variablen-, Funktions- oder eine andere Deklaration handelt. Um zu vermeiden, dass hier unnötige Knoten in der Datenbank angelegt werden, welche später entfernt und durch einen anderen Knoten ersetzt werden müssen, kommen hier einige Hilfsklassen zum Einsatz. Im Fall der Deklaration ist dies die Klasse `Declaration`, welche zunächst in der Deque als Platzhalter dient und bei näherer Bestimmung des Deklarationstyps durch den entsprechenden Descriptor ersetzt wird. Ähnlich verhält es sich bei einer Bedingung und bei einem Funktionsaufruf.

Ebenfalls problematisch stellt sich die Ermittlung der Zeilenanzahl einer Funktion heraus. Diese bestimmt die Größe des Elementes in der Visualisierung. Die Zeilenangabe in

der TypeChef-Ausgabe ist nicht in allen Fällen korrekt, sodass dann die Minimalgröße angenommen wird.

Wie der Tabelle in Anhang A zu entnehmen ist, sind nicht geforderten Sprachmittel für die Metainformationen implementiert worden. Die Extraktion der *Function Specifiers* sowie der *Storage Class Specifiers* wurde nicht umgesetzt. Auch bei den *Type Specifiers* und den *Type Qualifiers* fehlen einige Elemente. Dies beeinträchtigt jedoch nur den Tooltip beim Überfahren der Visualisierung und wurde deshalb nicht als kritisch angesehen.

Das Lesen und Schreiben von globalen Variablen wird ebenfalls nicht in allen Fällen aus dem AST extrahiert, da hier zahlreiche Verwendungsfälle vorliegen können. Der Prototyp konzentriert sich zunächst auf Zuweisungen. Zudem könnten gleichnamige lokale Variablen bei der Verwendung fälschlicherweise als globale Variablen erkannt werden, dies sollte jedoch nicht allzu häufig vorkommen.

Insgesamt ist aber die Extraktion aller für die Metapher notwendigen Elemente implementiert.

4.2.4 Parsen der Bedingungen

Wie in Abschnitt 4.1.2 beschrieben, ist der ausgewertete Ausdruck zur bedingten Kompilierung eines Elements als einfacher Text zwischen zwei `fexpr`-Tags eingefasst. Um diesen in der Graphdatenbank ablegen zu können, muss er geparst werden. Da diese Ausdrücke je nach Komplexität der Bedingung sehr komplex sein können, wurde hierfür ein separater Parser geschrieben. Um den Aufwand für die Entwicklung gering zu halten, bietet sich die Nutzung eines Parsergenerators an.Dieser generiert aus einer formal beschriebenen Grammatik einen Parser mit entsprechenden Zugriffsklassen.

Im vorliegenden Prototypen wird ANTLR[4] eingesetzt, ein Parsergenerator für die Programmiersprache Java. Mithilfe einer Grammatik wird die Syntax der Bedingungen niedergeschrieben. Listing 4.3 zeigt diese Grammatik.

```
1  grammar Conditions;
2
3  completeCondition : singleCondition | negativeCondition |
       andExpression | orExpression;
4
5  andExpression : '(' expression (AND expression)+ ')';
6
7  orExpression : '(' expression (OR expression)+ ')';
8
9  expression : singleCondition | negativeCondition |
       andExpression | orExpression;
10
```

[4] https://www.antlr.org/

```
11 negativeCondition : '!'singleCondition;
12
13 singleCondition : DEFINEDEX MACRONAME ')' | DEFINED MACRONAME
       ')';
14
15 DEFINED : 'defined(';
16 DEFINEDEX : 'definedEx(';
17 AND : '&amp;&amp;' | '&&';
18 OR : '||';
19 MACRONAME : [a-zA-Z0-9_]+ ;
20 WS : [ \r\t\n]+ -> skip;
```

Listing 4.3: ANTLR-Grammatik zum Parsen der Bedingungen

Eine Grammatik besteht aus einer Menge an Regeln. Lexerregeln, erkennbar an einem Großbuchstaben am Wortanfang, definieren Zeichenfolgen. In der abgebildeten Grammatik werden die Schlüsselwörter `defined` und `definedEx` beschrieben, sowie Und- und Oder-Symbole, Bezeichner und Leerzeichen. Im ersten Schritt wird der Text in Tokens zerlegen und Lexerregeln erkannt. Anschließend kommen Parserregeln zum Einsatz, welche größere Ausdrücke erkennen. Eine hier als `completeCondition` bezeichnete Bedingung kann eine einfache Bedingung, eine negierte Bedingung oder eine mit Und oder Oder verknüpfte Bedingung sein. Die in Hochkommas eingeschlossenen Zeichen markieren Tokens im Text. Die anderen Ausdrücke sind Parser- und Lexerregeln oder Teil der ANTLR-Syntax (vgl. [Parr 2007]).

Während eines Maven-Builds des jQAssistant-Plugins generiert ANTLR aus der Grammatik Java-Zugriffsklassen für die einzelnen Elemente der Grammatik. Die Klasse `ConditionsBaseListener` wird durch die Klasse `CConditionsListener` erweitert. In dieser Klasse werden, abhängig von der Art der Bedingung, die entsprechenden Descriptors erzeugt und ein `ConditionDescriptor` zurückgegeben, welcher den erzeugten Subgraphen enthält. Dieser wird mithilfe einer *DEPENDS_ON*-Beziehung mit den entsprechenden C-Sprachmitteln im bestehenden Graphen verknüpft.

4.3 Anpassung von Getaviz

Nachdem mithilfe von jQAssistant und ANTLR eine Neo4j-Datenbank mit der Struktur einer C-Quellcodedatei befüllt wurde, muss daraus eine Visualisierung erzeugt werden. Dafür soll der Visualisierungsgenerator von Getaviz zum Einsatz kommen. Es besteht bereits eine Implementierung für die Visualisierung von Java-Quellcode. Die Sprachmittel und ihre Abbildung in der Graphdatenbank unterscheiden sich jedoch so stark von der des C-Quellcodes, dass viele Schritte für C separat implementiert werden müssen. Abbildung 4.1 zeigt den Ablauf der Verarbeitung durch den Getaviz-Generator. Jeder Schritt zeigt in weißer Schrift den

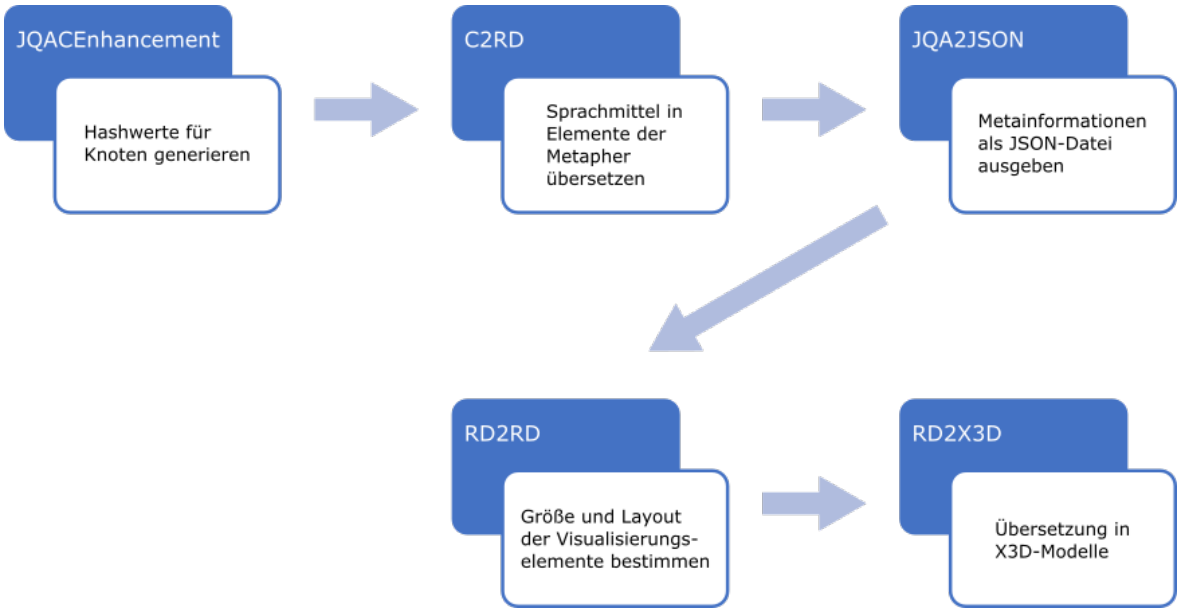

Abbildung 4.1: Klassen und Verarbeitungsschritte für C-Quellcode im Getaviz-Generator

Klassennamen und darunter eine kurze Zusammenfassung der Funktionalität. Im Folgenden werden die einzelnen Schritte des Generierungsprozesses kurz erläutert: Die erste aufgerufene Klasse, `JQACEnhancement`, generiert für jedes Sprachmittel, welches in der Visualisierung als eigenes Element abgebildet wird, einen Hashwert. Dieser dient als eindeutiger Bezeichner des Elements, um beispielsweise andere Elemente zu referenzieren. Er setzt sich zusammen aus dem Dateinamen, eventuell vorhandenen Elternelementen, wie im Fall von Struct-Variablen, und dem Namen des Elements selbst.

Anschließend wird die Klasse `C2RD` aufgerufen. Diese dient speziell der Transformation in die RD-Metapher. Wie der Name andeutet, werden in dieser Klasse aus den Elementen des C-Quellcodes entsprechende Elemente der RD-Metapher, also Scheiben und Scheibensegmente, generiert. Dabei wird der Graph ausgehend von den Übersetzungseinheiten durchlaufen und bei Erreichen eines abzubildenden Sprachmittels wird ein Element der Visualisierungsmetapher erzeugt.

Der nächste Schritt besteht im Aufruf der Klasse `JQA2JSON`. In dieser erfolgt die Erzeugung einer Datei namens `metadata.json`. Die Datei enthält Metadaten zu allen Visualisierungselementen, beispielsweise ihre ID, das Elternelement oder die Signatur einer Methode. Diese Informationen werden später ebenfalls für die Visualisierung benötigt, unter anderem für die Anzeige von Tooltips oder das Filtern von Elementen.

Die Klasse `RD2RD` ist nicht spezifisch für C-Quellcode implementiert und musste nicht angepasst werden. In dieser wird die Visualisierung weiter spezifiziert. Nachdem in einem vorhergehenden Schritt alle Elemente angelegt wurden, können nun die Größe der Elemente und das allgemeine Layout festgelegt werden, welche sich auch aus der Gesamtgröße und dem Verhältnis der Elemente bestimmen.

Auch die Klasse für den letzten Generierungsschritt, `RD2X3D`, musste nicht für C-Quellcode adaptiert werden. Diese erzeugt aus dem generierten Modell eine X3D-Datei,

welche die Visualisierung für den Browser beschreibt. Alternativ wird bei entsprechender Konfiguration die Klasse `RD2AFrame` aufgerufen, welche die Visualisierung für die Verarbeitung durch das Framework A-Frame[5] erzeugt.

4.4 Erweiterung der graphischen Oberfläche

Die grafische Oberfläche der Visualisierung wird mithilfe von JavaScript-Code erzeugt. Wenn eine Visualisierung für die Anzeige im Browser initialisiert wird, wird zunächst aus dem Inhalt der Datei `metadata.json` ein Modell erzeugt. Außerdem werden einige spezielle Datenstrukturen initialisiert, um die weitere Verarbeitung und Interaktion mit der Visualisierung zu vereinfachen. Die Erzeugung des Modells erfolgt in der Datei `Model.js`. In dieser Datei dient die Methode `initialize()` als Einstiegsmethode. Um das Modell zu erzeugen, wird über alle Elemente des Modells iteriert und währenddessen werden JavaScript-Objekte erzeugt. Ein zweiter Iterationsschritt vervollständigt anschließend das Modell, indem er beispielsweise Beziehungen zwischen Elementen auflöst.

In der Datei `Model.js` mussten einige Erweiterungen vorgenommen werden. Alle Elemente des C-Graphmodells, zum Beispiel die `TranslationUnit` oder die `Struct`, werden in die Erzeugung des Modells eingefügt und notwendige Verarbeitungsschritte vorgenommen. Elemente vom Typ `Macro` werden zu einer Map hinzugefügt. Diese Map dient später dazu, die Checkboxen im *Macro Explorer* zu erstellen. Bei Sprachmitteln, welche von einer Makrodefinition abhängen können, wird diese Bedingung ebenfalls vorverarbeitet. Über eine rekursive Funktion erfolgt das Durchlaufen der Bedingungsknoten. Hierbei wird ebenfalls in einer Map gespeichert, welches Modell-Element von welcher symbolischen Konstante abhängig ist. Dies vereinfacht später die Auswertung einer Änderung der Makrodefinitionen.

Für den in Abschnitt 3.3.2 definierten *Macro Explorer* wird analog zur bisherigen Struktur ein eigener Controller erstellt. Dieser Controller ähnelt im Aufbau dem `PackageExplorerController`, da dieser ebenfalls eine Gruppe von Checkboxen enthält und dem Filtern von Modellelementen dient. Der neu angelegte Controller heißt `MacroExplorerController`. Beim Aktivieren des Controllers werden die zuvor ermittelten symbolischen Konstanten abgefragt. Für jede Konstante wird eine Checkbox mit dem Namen der symbolischen Konstante als Bezeichner erstellt. Als Callback-Methode für ein Check-Event wird eine Methode namens `macroOnCheck()` erstellt. Diese ermittelt alle veränderten Makrodefinition, erzeugt ein Ereignis und veröffentlicht dieses. Das Ereignis enthält noch einige andere notwendige Informationen.

Die Verarbeitung des Ereignisses geschieht in der Klasse `CanvasFilterController`. Diese fängt einige Events des Systems und wertet diese aus. Zunächst werden nach dem Empfangen eines Ereignisses vom Typ `macroChanged` alle Modellelemente ermittelt, deren Anzeige von dem geänderten Makro abhängt. Damit soll der Aufwand bei der

[5] https://aframe.io/

Auswertung der Zustandsänderung verringert werden. Dies spielt vor allem bei großen Modellen eine Rolle. Anschließend muss für jedes Modellelement anhand des Zustands aller Makrodefinitionen die Bedingung für die Anzeige ausgewertet werden. Dies geschieht wieder rekursiv, indem alle Teile der Bedingung ausgewertet und verknüpft werden, sodass ein Gesamt-Wahrheitswert resultiert. Mithilfe desselben werden die Modellelemente in Listen sortiert: eine Liste mit den anzuzeigenden und eine mit den auszublendenden Elementen. Diese werden an schon bestehende Funktionen im `CanvasManipulator` übergeben, woraufhin dieser die Visualisierung anpasst. Dabei kann in einer Konfigurationsdatei definiert werden, ob die nicht im Quellcode enthaltenen Elemente nur transparent gemacht oder vollständig ausgeblendet werden sollen.

Dadurch, dass sowohl der Package Explorer als auch der Macro Explorer Modellelemente herausfiltern, können inkonsistente Zustände entstehen. Deshalb wurde die Filtermöglichkeit im Package Explorer in einem ersten Schritt deaktiviert. Bei einer Erweiterung des Prototyps wären die Reaktivierung und ein gegenseitiger Abgleich der Filterzustände wünschenswert. Dadurch könnten beide Funktionalitäten genutzt und trotzdem ein korrekter Zustand der Visualisierung garantiert werden.

4.5 Test und Evaluation

Zu einer vollständigen Umsetzung der Forschungsmethode des Prototypings gehört das ausgiebige Testen und eine umfangreiche Evaluation des Prototyps. In dieser Arbeit lag der Fokus auf der Implementierung des gesamten Verarbeitungsprozesses. Für das implementierte jQAssistant-Plugin wurden zahlreiche Integrationstests mithilfe von JUnit[6] geschrieben. JQAssistant unterstützt Integrationstests durch die abstrakte Klasse `AbstractPluginIT`. Diese soll für Integrationstests erweitert werden und stellt so eine Testumgebung mit einer Instanz der Graphdatenbank Neo4j zur Verfügung. Da der im Plugin enthaltene Quellcode vor allem den Zugriff auf die Graphdatenbank implementiert und keine komplexe Verarbeitungslogik enthält, lag der Schwerpunkt auf den Integrationstests im Zusammenspiel mit der Graphdatenbank und dem jQAssistant-Framework.

Der Quellcode im Getaviz-Generator wurde durch Sichtkontrollen der erzeugten Visualisierungen geprüft, hier fehlen bisher umfangreiche Unit- und Integrationstests.

Um das Plugin an einem realen Softwaresystem zu testen, wurde BusyBox 1.18.5 verwendet und mit TypeChef ausgewertet. In der Verarbeitung wurden nur Makros berücksichtigt, die mit CONFIG_ beginnen, da BusyBox über mehrere hundert derart deklarierte Merkmale verfügt. Für alle 522 Übersetzungseinheiten wurde ein eigener AST erstellt. Anschließend wurden diese Dateien mit jQA geparst, dieser Vorgang nahm ungefähr 24 Stunden in Anspruch. Während des Parsens traten bei einigen ASTs Fehler auf, die auf falsch erkannte Deklarationen zurückzuführen sind, im Rahmen dieser Arbeit jedoch nicht behoben werden konnten. Der überwiegende Teil der Anweisungen wurde jedoch korrekt

[6] https://junit.org/junit4/

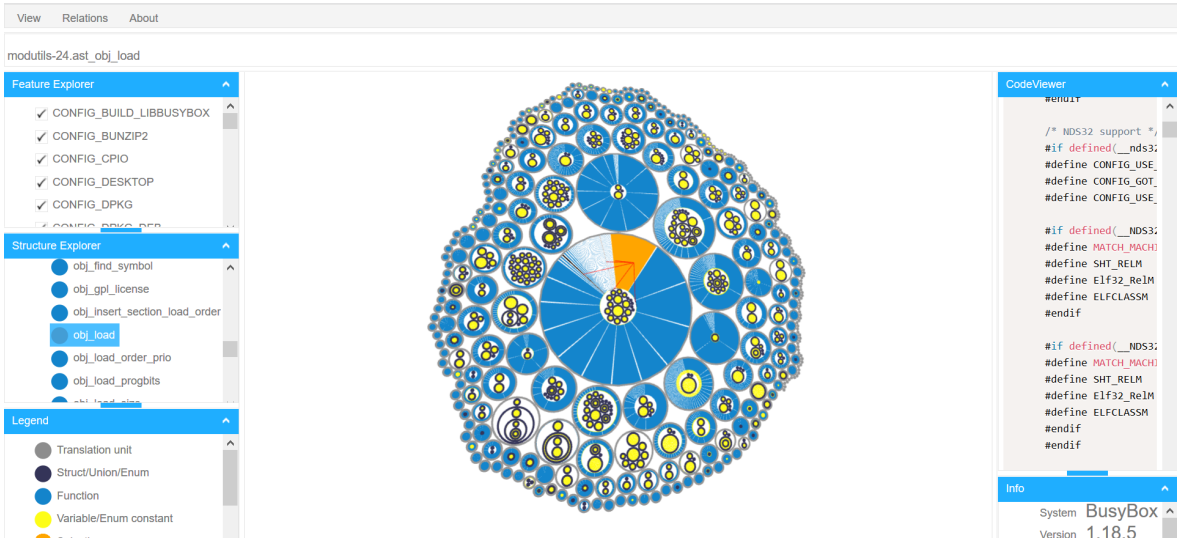

Abbildung 4.2: Screenshot der Visualisierung von BusyBox

verarbeitet und in der Graphdatenbank gespeichert. Abbildung 4.2 zeigt einen Screenshot der erzeugten Visualisierung im Browser. In der Abbildung ist die Unübersichtlichkeit der Visualisierung zu erkennen. Dies liegt zum Einen an der Größe des Softwaresystems, zum Anderen aber auch an der fehlenden Strukturierung der einzelnen Übersetzungseinheiten. Dies stellt ein grundsätzliches Problem von prozeduralen Programmiersprachen dar.

Ein weiterer notwendiger Schritt im Rahmen der Forschungsmethode ist eine Nutzerevaluation. Beispielsweise kann ein Vergleich mit einem ähnlichen Werkzeug vorgenommen werden. Hierbei müssen die Probanden einige Aufgaben lösen und eine statistische Auswertung zeigt die Effizienz und Effektivität der Werkzeuge bei der Bearbeitung. Außerdem können Interviews Aufschluss über Verbesserungsmöglichkeiten und Mängel in der Implementierung geben. Dazu müssen die Aufgaben und Auswertungen sowie eine Einführung der Nutzer in die Benutzungsoberfläche sorgfältig vorbereitet werden. Eine solche Evaluation ist der nächste Schritt nach Abschluss der Implementierung, um anhand der Ergebnisse den Prototypen auszuwerten und die Darstellung zu überarbeiten. Auf diese Weise wird die Zweckmäßigkeit der Visualisierung, wie in Abschnitt 3.1 gefordert, sichergestellt.

5 Fazit

5.1 Zusammenfassung

In der vorliegenden Arbeit wurde die Visualisierung von Variabilität in C-Quellcode untersucht. Ausgangspunkt war die Komplexität und schlechte Wartbarkeit des Quellcodes beim Einsatz von Präprozessoranweisungen. Um hier Abhilfe zu schaffen, wurde die Entwicklung einer Visualisierung für den Quellcode vorgeschlagen. Die soll helfen, einen Überblick über komplexe Programme zu gewinnen. Um das entwickelte Konzept zu überprüfen, wurde ein Prototyp, aufbauend auf dem Getaviz-Visualisierungsgenerator, entwickelt.

Im Folgenden sollen die Ergebnisse der Arbeit zusammengefasst werden. Zunächst wurden anhand des C-Standards relevante Sprachmittel für die Visualisierung identifiziert. Die hier entwickelte Visualisierung soll einen einfachen Überblick über die Struktur des Programms geben. Daraus folgt, dass folgende Elemente extrahiert werden sollen: Funktionen, globale Variablen, Structs, Unions, Enums und als übergreifendes Element die Übersetzungseinheit. Zu Structs und Unions wurden die darin deklarierten Variablen sowie zu Enums die deklarierten Konstanten aufgenommen. Um die Visualisierung mit zusätzlichen Informationen anreichern zu können, wurden zudem einige Metadaten für die Extraktion aus dem Quellcode identifiziert, darunter der Funktionskopf, Typspezifizierer und -qualifizierer.

Auch die Verarbeitung der Präprozessoranweisungen wurde detailliert beschrieben. Dabei liegt der Fokus auf Anweisungen zur bedingten Kompilierung. Makroexpansionen und Dateiinklusionen sollten vor dem Parsen des Quellcodes vorgenommen werden.

Anschließend wurden Anforderungen formuliert, welche ein Werkzeug zum Parsen von C-Quellcode erfüllen müsste, um die genannten Sprachmittel für die Visualisierung zu extrahieren. Eine Recherche ergab, dass das Entwickeln eines eigenen Parsers im Rahmen dieser Arbeit nicht möglich ist. Allerdings wurden mit MAPR, TypeChef und SuperC drei Softwaresysteme identifiziert, welche einen AST für C-Quellcode erzeugen, der die Anweisungen zur bedingten Kompilierung in einer vorverarbeiteten Form enthält. Nach einem Vergleich fiel die Entscheidung für den Einsatz von TypeChef.

Um den Quellcode nach dem Parsen zu speichern, wird in Getaviz die Graphdatenbank Neo4j verwendet. Deshalb musste zunächst ein Graphmodell entworfen werden, welches die extrahierten Elemente des C-Quellcodes abbildet. In diesem Graphmodell sind auch die Bedingungen enthalten, welche TypeChef während des Parsens für die einzelnen Sprachmittel berechnet.

Der zweite Teil der Konzipierung befasst sich mit der Visualisierung der Informationen. Dabei wurden zunächst die in Getaviz implementierten Metaphern betrachtet und auf ihre Tauglichkeit für C-Quellcode geprüft. Die City-Metapher wurde hierbei negativ bewertet, da für ihre beiden bedeutenden Elemente, der Bezirk und das Gebäude, keine Äquivalente in C bestehen. Für die RD-Metapher verlief die Übertragung erfolgreich. Anschließend wurde diese Metapher für die Implementierung des Prototypen herangezogen.

Neben der Konzipierung besteht ein zweiter bedeutender Beitrag dieser Arbeit in der prototypischen Implementierung der entworfenen Visualisierung. Hierzu erfolgte zunächst eine Anpassung von TypeChef, sodass der erzeugte AST mithilfe einer Kommandozeilenoption als XML-Datei ausgegeben werden kann. Auf Basis der hiermit erzeugten XML-Dateien wurde das jQAssistant-Plugin entwickelt. Dieses parst die XML-Dateien und legt die Strukturelemente des Quellcodes in einer Neo4j-Datenbank, entsprechend dem zuvor entworfenen Graphmodell, ab. Um die Bedingungsausdrücke korrekt zu parsen, wurde eine ANTLR-Grammatik geschrieben und in das jQAssistant-Plugin integriert.

Nach dem Parsen des Quellcodes muss eine Visualisierung generiert werden. Dies geschieht im Prototypen mithilfe des Getaviz-Visualisierungsgenerators. Der Generator wurde so erweitert, dass er auch die Elemente des C-Quellcodes in Visualisierungselemente übersetzen kann. Hier waren vor allem bei der Generierung der Metadaten und der Glyphen Anpassungen nötig. Das Erstellen des Layouts nach der Generierung musste ebenfalls leicht angepasst werden, um die modifizierte RD-Metapher ohne Überschneidungen darzustellen.

Für den entworfenen MacroExplorer wurde ein eigener Controller angelegt, welcher eine Liste mit Checkboxen enthält. Dieser wird beim Start der Oberfläche befüllt. Zudem wurden bestehende Komponenten für die Abbildung von C-Sprachmitteln erweitert, beispielweise der PackageExplorer oder die angezeigten Tooltips für die Visualisierungsglyphen.

Durch den erstellten Prototypen konnte gezeigt werden, dass die Visualisierung bedingter Kompilierung in C-Quellcode möglich ist. Notwendige Werkzeuge zur automatischen Erstellung einer solchen Darstellung konnten gefunden beziehungsweise entwickelt werden. Unit-Tests und manuelle Kontrollen zeigen bisher keine gravierenden Probleme in der Implementierung. Auch das erfolgreiche Parsen von BusyBox zeigt die Robustheit und den Nutzen des Prototypen.

Eine Einschränkung in der Visualisierung, welche sich durch TypeChef ergibt, ist die Beschränkung auf boolesche Bedingungsausdrücke. Dies resultiert aus einer sehr schlechten Performance bei der Verarbeitung von anderen Bedingungen durch den eingesetzten SAT-Solver. An dieser Stelle wären weitere Forschungen nötig, ob diese Fälle mithilfe einer anderen Technologie ausgewertet werden könnten. Dies ist jedoch nicht Bestandteil dieser Arbeit. Weitere Einschränkungen und mögliche angrenzende Forschungsgebiete werden in den folgenden Abschnitten erläutert.

5.2 Kritische Würdigung

Einschränkungen bezüglich der Implementierung ergeben sich in dem beschriebenen Prototypen zu großen Teilen aus der Nutzung von TypeChef. Durch die vollständige Auswertung aller Präprozessoranweisungen und die Ausgabe aller möglichen Bedingungen für einen Quellcodebestandteil ist die Erzeugung des AST sehr rechenintensiv. Bis zuletzt konnten reale Programme nur mithilfe einer Definition der zu berücksichtigenden Makros analysiert werden. Dies entspricht jedoch der Vorgehensweise der Entwickler bei ihrer Analyse des

Linux-Kernels (vgl. [Kästner et al. 2011, 13]), wobei sie die symbolischen Konstanten auf solche mit dem Präfix *CONFIG_* beschränken.

Eine weitere Beschränkung des Funktionsumfangs ergibt sich bezüglich der Art der Bedingungen. Wie im vorhergehenden Abschnitt geschildert, liefert TypeChef ausschließlich Informationen zu booleschen Bedingungen. Daraus ergab sich in dieser Arbeit ein engerer Fokus in der Verarbeitungspipeline und in der Erweiterung der Weboberfläche. Wie die Verarbeitung und Darstellung weiterer Bedingungstypen in die bestehende Implementierung integriert werden kann, muss in weiteren Forschungsarbeiten geklärt werden.

Was in dieser Arbeit keine Berücksichtigung findet, sind Bedingungen für Elemente, die keine vollständigen Sprachmittel darstellen. Da mit einem lexikalischen Präprozessor beliebig kleine Quellcodebestandteile in der bedingten Kompilierung eingefasst werden können, ergeben sich hier unzählige Möglichkeiten. TypeChef gibt diese vollständig aus. Jedoch zeigt die hier entwickelte Visualisierung nur Bedingungen für vollständige Sprachmittel an. Dies kann bei der Nutzung durch einen Entwickler gegebenenfalls zu Problemen führen, weil Sprachmittel unterschiedlich gestaltet sein können, je nach Konfiguration. Wie solche Fälle behandelt werden sollen, ist zu einem späteren Zeitpunkt zu klären.

Zumindest bei dem ersten genannten Problem kann möglicherweise die Verwendung von SuperC Abhilfe verschaffen, da dessen Algorithmus deutlich effizienter sein soll. Hier sind aber zunächst einige Aktualisierungen der verwendeten Technologien und Frameworks notwendig, was eine umfangreiche Einarbeitung erfordern würde.

Einige umständliche Verarbeitungsschritte in der Implementierung ergeben sich durch das sukzessive Durchlaufen der XML-Datei. Zudem enthält das automatisch erzeugte Dokument einige überflüssige oder unnötig aufgeblähte Elemente, welches das Einlesen zusätzlich erschweren. Hier sollte in einer nächsten Iteration über den direkten Zugriff auf den AST über eine Scala-Schnittstelle nachgedacht werden. Dies bringt möglicherweise auch eine größere Performanz mit sich.

5.3 Ausblick

Nachdem die Ergebnisse dieser Arbeit zusammengefasst und kritische Punkte angesprochen wurden, sollen nun noch einige weiterführende Themen zu dieser Arbeit besprochen und mögliche nächste Schritte aufgezeigt werden. Der Abschnitt gliedert sich in zwei Teile. Zunächst werden Erweiterungen besprochen, welche direkt die hier vorgenommene Implementierung betreffen. Anschließend folgt eine Menge von weiterführenden Forschungsthemen.

Zunächst sollte versucht werden, bekannte Probleme in der Implementierung zu lösen. Hier ist beispielweise die nicht eindeutige Darstellung von Sprachmitteln in der XML-Datei zu nennen. Um dieses Problem zu beseitigen, muss die Implementierung von TypeChef entsprechend verfeinert werden. Auch die Zeilenanzahl von Funktionen kann nicht in allen Fällen aus dem XML-Dokument ermittelt werden, was ebenfalls in TypeChef behoben werden muss.

Eine weitere wichtige Aufgabe besteht darin, die verwendeten Programme entsprechend dem Arbeitsablauf aus Abbildung 3.12 zu verknüpfen. Dies könnte beispielsweise durch ein Batch-Skript geschehen oder indem der Getaviz-Generator die vorhergehenden Programme ausführt. So wird mithilfe eines integrierten Verarbeitungsprozesses die Generierung der Modelle beschleunigt und für den praktischen Einsatz nutzbar. Zudem können auf diese Weise auch größere Softwareprojekte ohne eine aufwendige manuelle Ausführung der einzelnen Komponenten visualisiert werden.

Wie das BusyBox-Beispiel gezeigt hat, sind noch einige Tests notwendig, um Fehler in der Implementierung aufzudecken und zu beseitigen. Auch über automatisierte Integrations- und Oberflächentests kann die Korrektheit des Programms weiter sichergestellt werden. Diese Schritte waren im Rahmen dieser Arbeit leider nicht umzusetzen.

Anschließend sollte eine Nutzerevaluierung stattfinden, um die in Abschnitt 3.1 geforderte Zweckmäßigkeit der Visualisierung zu prüfen. Hierzu muss zunächst ein Versuchsaufbau konstruiert werden. Die erhaltenen Versuchsergebnisse müssen dann in einem nächsten Schritt in die Weiterentwicklung der Visualisierung fließen. Mehrere Iterationen und Vergleiche mit alternativen Darstellungsmöglichkeiten helfen, dass das Werkzeug nutzbringend in der Softwareentwicklung eingesetzt werden kann.

In einem nächsten Schritt sollte die Möglichkeit geschaffen werden, auch Bedingungen mit nicht-booleschen Werten aus dem Quellcode zu extrahieren und zu visualisieren. Hier stellt sich zunächst die Frage, wie diese Extraktion geschehen kann. Möglicherweise muss dazu ein eigenes Werkzeug entwickelt werden, da TypeChef diese Informationen in der bestehenden Implementierung nicht liefert und auch nicht berechnen kann (vgl. [Kästner et al. 2011, 16]). Wenn eine Extraktion aus dem Quellcode möglich ist, muss das Graphmodell im Bereich der Bedingungen angepasst werden, um diese Informationen ablegen zu können.

Auch die Darstellung der Bedingungen in der Visualisierung ist ein bisher unerforschtes Gebiet, in welchem verschiedene Varianten entworfen und mithilfe von Experimenten evaluiert werden müssen. Die hier vorgestellte Variante ist nur ein erster Versuch der Abbildung dieser Informationen. Diese Aufgabe wird zudem noch deutlich erweitert, wenn die Bedingungen auch nicht-boolesche Werte enthalten können. Dann stellt sich die Frage, wie die verschiedenen Werte abgelegt und angezeigt werden und welche Optionen der Nutzer hat, diese auszuwählen.

Die in dieser Arbeit genutzte Visualisierungsmetapher ist ursprünglich für objektorientierte Software entworfen worden. Deshalb musste sie für die Darstellung von C-Quellcode leicht modifiziert werden. Möglicherweise ist die Abbildung mit der Übersetzungseinheit als äußere Glyphe für die Nutzer des Systems nicht intuitiv. Zudem leidet die Übersichtlichkeit der Visualisierung unter der fehlenden Strukturierung des prozeduralen Codes. Gegebenenfalls kann hier die Ordnerstruktur eines Softwareprojekts als zusätzliche Gliederungsmöglichkeit betrachtet werden. Auch in diesem Bereich sind Nutzerevaluationen notwendig, um

Vor- und Nachteile der Metapher zu ermitteln und gegebenenfalls Anpassungen vorzunehmen. Zudem stellt sich die Frage, wie eine geeignete Metapher für prozedurale Programmiersprachen beschaffen sein muss, um die Zweckmäßigkeit der Visualisierung sicherzustellen.

Eine umfangreiche Filterfunktionalität kann ebenfalls hilfreich sein. Diese unterstützt dadurch, dass nur die Teile des Quellcodes angezeigt werden, die für die Entwicklerin im Moment der Betrachtung von Bedeutung sind.

Zusammenfassend lässt sich festhalten, dass der hier entwickelte Prototyp die Möglichkeit der Bedingungsvisualisierung zeigt. Es sind jedoch noch viele Fragen im Bereich des Prototyps, aber auch allgemein in der Visualisierung von prozeduralem Quellcode offen.

A Übersicht über die Sprachmittel im C-Standard

Tabelle A.1: Sprachmittel im C-Standard

Abschnitt	Sprachmittel	relevant	implementiert
6.2.5 Types	_Bool	M	x
	char	M	x
	signed char	M	x
	short int	M	x
	int	M	x
	long int	M	x
	long long int	M	x
	unsigned	M	x
	float	M	x
	double	M	x
	long double	M	x
	float _Complex	M	
	double _Complex	M	
	long double _Complex	M	
	void	M	x
	array type	M	x
	structure type	M	x
	union type	M	x
	function type	M	
	pointer type	M	x
6.4.1 Keywords	Keywords	t	x
6.4.2 Identifiers	Identifiers	M	x
6.4.4 Constants	Constants		

Tabelle A.1: Sprachmittel im C-Standard (Fortsetzung)

Abschnitt	Sprachmittel	relevant	implementiert
6.4.5 String literals	String literals		
6.4.6 Punctuators	Punctuators	t	x
6.4.7 Header names	Header names		
6.4.8 Preprocessing numbers	Preprocessing number		
6.4.9 Comments	Comments		
6.5 Expressions	Expressions	t	
6.5.2.2 Function calls	Function calls	V	
6.6 Constant expressions	Constant expressions		
6.7 Declarations	Declarations	t	x
6.7.1 Storage-class specifiers	typedef		
	extern	M	
	static	M	
	_Thread_local	M	
	auto	M	
	register	M	
6.7.2 Type specifiers	void	M	x
	char	M	x
	short	M	x
	int	M	x
	long	M	x
	float	M	x
	double	M	x
	signed	M	x
	unsigned	M	x

Tabelle A.1: Sprachmittel im C-Standard (Fortsetzung)

Abschnitt	Sprachmittel	relevant	implementiert
	_Bool	M	x
	_Complex	M	
	atomic-type-specifier	M	
	struct-or-union-specifier	M	x
	enum-specifier	M	x
	typedef-name	M	x
6.7.2.1 Structure and union specifiers	struct	V	x
	union	V	x
6.7.2.2 Enumeration specifiers	enum	V	x
6.7.3 Type qualifiers	_Atomic	M	
	const	M	x
	volatile	M	x
	restrict	M	
6.7.4 Function specifiers	inline	M	
	_Noreturn	M	
6.7.5 Alignment specifier	_Alignas		
6.7.6.1 Pointer declarators	Pointer declarators	M	x
6.7.6.2 Array declarators	Array declarators	M	x
6.7.6.3 Function declarators	Function declarators	V	x
6.7.7 Type names	Type names	M	x
6.7.8 Type definitions	Type definitions		
6.7.9 Initialization	Initialization	t	x
6.7.10 Static assertions	Static assertions		
6.8 Statements and blocks	Statements and blocks		

Tabelle A.1: Sprachmittel im C-Standard (Fortsetzung)

Abschnitt	Sprachmittel	relevant	implementiert
6.9.1 Function definitions	Function definitions	V	x
6.9.2 External object definitions	External object definitions		
6.10.1 Conditional inclusion	#if	V	
	#elif	V	
	#ifdef / #if defined	V	x
	#ifndef / #if !defined	V	x
	#else	V	x
	#endif	V	x
6.10.2 Source file inclusion	#include		
6.10.3 Macro replacement	Macro replacement		
6.10.4 Line control	#line		
6.10.5 Error directive	#error		
6.10.6 Pragma directive	#pragma		
6.10.8 Predefined macro names	Predefined macro names		
6.10.9 Pragma operator	_Pragma		

Tabelle A.1: Übersicht der Sprachmittel im C-Standard mit Bewertung der Relevanz: Die erste Spalte gibt den Abschnitt im C-Standard von 2017 an [ISO/IEC 2017]. Spalte zwei nennt das Sprachmittel. In Spalte drei steht der Buchstabe “V” für Visualisierung. Die so gekennzeichneten Elemente werden direkt als Elemente der Visualisierungsmetapher abgebildet. “M” steht für Metainformationen und bezeichnet Informationen im Quellcode, welche nicht direkt in der Visualisierung abgebildet werden, aber beispielsweise als Tooltip angezeigt werden. “t” steht für teilweise, diese Informationen werden zum Teil extrahiert und genutzt. Die vierte Spalte zeigt, ob das Sprachmittel im Prototyp schon implementiert ist.

Literaturverzeichnis

[Antoniol et al. 1997] Antoniol, G., Fiutem, R., Lutteri, G., Tonella, P., Zanfei, S. und Merlo, E. „Program understanding and maintenance with the CANTO environment“. In: 1997 Proceedings International Conference on Software Maintenance. 1997, S. 72–81.

[Bassil/Keller 2001] Bassil, S./Keller, R. K. „Software visualization tools: Survey and analysis“. In: Proceedings 9th International Workshop on Program Comprehension. IWPC 2001. 2001, S. 7–17.

[Baum et al. 2017] Baum, D., Schilbach, J., Kovacs, P., Eisenecker, U. und Muller, R. „GETAVIZ: Generating Structural, Behavioral, and Evolutionary Views of Software Systems for Empirical Evaluation“. In: VISSOFT 2017. Piscataway, NJ: IEEE, 2017, S. 114–118.

[Bhattacharjee et al. 1994] Bhattacharjee, A. K., Seby, A., Sen, G. und Dhodapkar, S. D. „CLAS: a reverse engineering tool“. In: Proceedings of 1994 1st International Conference on Software Testing, Reliability and Quality Assurance (STRQA'94). 1994, S. 126–130.

[Bibliographisches Institut GmbH 2018] Bibliographisches Institut GmbH, Hrsg. Duden: variabel. 2018. URL: https://www.duden.de/rechtschreibung/variabel (gelesen am 18. 06. 2019).

[Czarnecki 2013] Czarnecki, K. „Variability in Software: State of the Art and Future Directions“. In: Fundamental Approaches to Software Engineering. Hrsg. von Vittorio Cortellessa/Dániel Varró. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, S. 1–5.

[Diehl 2007] Diehl, S. Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software. Berlin, Heidelberg: Springer-Verlag Berlin Heidelberg, 2007.

[Eisenecker 2008] Eisenecker, U. C++: der Einstieg in die Programmierung: strukturiert & prozedural programmieren. W3l GmbH, 2008.

[Ernst et al. 2002] Ernst, M. D., Badros, G. J. und Notkin, D. „An empirical analysis of C preprocessor use“. In: IEEE Transactions on Software Engineering 28.12 (2002), S. 1146–1170.

[Gazzillo/Grimm 2012] Gazzillo, P./Grimm, R. „SuperC: parsing all of C by taming the preprocessor“. In: ACM SIGPLAN Notices 47.6 (2012), S. 323–334.

[Georget et al. 2015] Georget, L., Tronel, F. und Tong, V. V. T. „Kayrebt: An activity diagram extraction and visualization toolset designed for the Linux codebase“. In: 2015 IEEE 3rd Working Conference on Software Visualization (VISSOFT). 2015, S. 170–174.

[Henning/Vogelsang 2007] Henning, P. A./Vogelsang, H., Hrsg. Handbuch Programmiersprachen: Softwareentwicklung zum Lernen und Nachschlagen : [ABAP, Ada, C, C#, C++, Delphi, Eiffel, FORTRAN, Java, JavaScript, LISP, Maple, Mathematica, MATLAB, Perl, PHP, PROLOG, Python, Ruby, Smalltalk, SQL, Tcl/Tk, Visual Basic.NET. München: Hanser, 2007.

[Hundhausen 1996] Hundhausen, C. D. A Meta-Study of Software Visualization Effectiveness. 1996. URL: www.eecs.wsu.edu/~veupl/pub/MetaStudy.pdf.

[ISO/IEC 2017] ISO/IEC, Hrsg. ISO/IEC 9899:2017: N2176. 2017. URL: https://web.archive.org/web/20181230041359if_/http://www.open-std.org/jtc1/sc22/wg14/www/abq/c17_updated_proposed_fdis.pdf (gelesen am 18. 06. 2019).

[Kästner et al. 2011] Kästner, C., Giarrusso, P. G., Rendel, T., Erdweg, S., Ostermann, K. und Berger, T. „Variability-aware parsing in the presence of lexical macros and conditional compilation“. In: ACM SIGPLAN Notices. Bd. 46. 10. ACM. 2011, S. 805–824.

[Kästner/Liebig 2015] Kästner, C./Liebig, J. ckaestne/TypeChef: TypeChef/Parameter.txt. 2015. URL: https://github.com/ckaestne/TypeChef/blob/master/Parameter.txt (gelesen am 18. 06. 2019).

[Liebig et al. 2010] Liebig, J., Apel, S., Lengauer, C., Kästner, C. und Schulze, M. „An analysis of the variability in forty preprocessor-based software product lines“. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1. 2010, S. 105–114.

[Mahler 2017] Mahler, D. eXtended Objects. 2017. URL: http://buschmais.github.io/extended-objects/doc/0.8.0/doc/ (gelesen am 18. 06. 2019).

[Mahler 2018] Mahler, D. Implementation Of A Scanner Plugin. 2018. URL: https://101.jqassistant.org/implementation-of-a-scanner-plugin/index.html (gelesen am 18. 06. 2019).

[Medeiros et al. 2013] Medeiros, F., Lima, T., Dalton, F., Ribeiro, M., Gheyi, R. und Andfonseca, B. „Colligens: A Tool to Support the Development of Preprocessor-Based Software Product Lines in C“. In: Proc. áBrazilian Conf. áSoftware: Theory and Practice (CBSoft). 2013.

[Müller 2019] Müller, Richard, Hrsg. Visual Software Analytics. 2019. URL: http://home.uni-leipzig.de/svis/Showcases/ (gelesen am 18. 06. 2019).

[o.V. 2018] o.V. Class Stack<E>. 2018. URL: https://docs.oracle.com/javase/8/docs/api/java/util/Stack.html (gelesen am 18. 06. 2019).

[o.V. 2019a] o.V. A Graph Platform Reveals and Persists Connections. 2019. URL: https://neo4j.com/product/ (gelesen am 18. 06. 2019).

[o.V. 2019b] o.V. The Neo4j Drivers Manual v1.7: About the official drivers. Hrsg. von Neo4j, Inc. 2019. URL: https://neo4j.com/docs/driver-manual/1.7/get-started/#driver-get-started-about (gelesen am 18. 06. 2019).

[o.V. 2019c] o.V. The Neo4j HTTP API Docs v3.5. Hrsg. von Neo4j, Inc. 2019. URL: https://neo4j.com/docs/http-api/3.5/ (gelesen am 18. 06. 2019).

[Parr 2007] Parr, T. The definitive ANTLR reference: Building domain-specific languages. Raleigh, NC: Pragmatic Bookshelf, 2007.

[Platoff et al. 1991] Platoff, M., Wagner, M. und Camaratta, J. „An integrated program representation and toolkit for the maintenance of C programs“. In: Proceedings. Conference on Software Maintenance 1991. 1991, S. 129–137.

[Robinson et al. 2015] Robinson, I., Webber, J. und Eifrem, E. Graph databases: new opportunities for connected data. O'Reilly Media, Inc, 2015.

[Spencer/Collyer 1992] Spencer, H./Collyer, G. # ifdef considered harmful, or portability experience with C News. 1992.

[Thüm et al. 2014] Thüm, T., Kästner, C., Benduhn, F., Meinicke, J., Saake, G. und Leich, T. „FeatureIDE: An extensible framework for feature-oriented software development“. In: Science of Computer Programming 79 (2014), S. 70–85.

[Ullenboom 2014] Ullenboom, C. Java SE 8 Standard-Bibliothek: das Handbuch für Java-Entwickler; [aktuell zu Java 8; Zweite Insel] /. 2. Aufl. Galileo Press, 2014.

[Wilde/Hess 2007] Wilde, T./Hess, T. „Forschungsmethoden der Wirtschaftsinformatik“. In: Wirtschaftsinformatik 49.4 (2007), S. 280–287.

[Wolf 2019] Wolf, J. C von A bis Z: Das umfassende Handbuch. 3., aktualisierte und erweiterte Auflage 2009, 7. korrigierter Nachdruck 2019. Bonn: Rheinwerk Verlag, 2019.

Ehrenwörtliche Erklärung

Ich versichere, dass ich die Masterarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Darüber hinaus versichere ich, dass die elektronische Version der Masterarbeit mit der gedruckten Version übereinstimmt.