

Eine Webanwendung zur prädikatenlogischen Modellprüfung von Graphstrukturen

B A C H E L O R A R B E I T

zur Erlangung des Grades eines Bachelor of Science
im Fachbereich Elektrotechnik/Informatik
der Universität Kassel

Eingereicht von: Jan Müller
Anschrift: Käthe-Kollwitz-Straße 7
34246 Vellmar

Matrikelnummer: 35011918
Emailadresse: janmueller3698@gmail.com

Vorgelegt im: Fachgebiet Theoretische Informatik/Formale Methoden

Gutachter: Prof. Dr. Martin Lange
Prof. Dr. Albert Zündorf

Betreuer: Dr. Norbert Hundeshagen

eingereicht am: 10. Februar 2021

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und nur mit den nach der Prüfungsordnung der Universität Kassel zulässigen Hilfsmitteln angefertigt habe. Die verwendete Literatur ist im Literaturverzeichnis angegeben. Wörtlich oder sinngemäß übernommene Inhalte habe ich als solche kenntlich gemacht.

Kassel, 10. Februar 2021

Jan Müller

Inhaltsverzeichnis

Erklärung	III
Abbildungsverzeichnis	VII
Algorithmen und Quelltexte	IX
1 Einleitung	1
1.1 Vorarbeiten	1
1.2 Motivation und Ziele	1
1.3 Struktur der Arbeit	2
2 Mathematischer Hintergrund	4
2.1 Signaturen und Strukturen	4
2.2 Interpretationen	4
2.3 Modellprüfungen	5
2.4 Erweiterung des Algorithmus zur Modellprüfung	6
3 Serveranwendung	8
3.1 Technologischer Hintergrund	8
3.1.1 Kotlin	8
3.1.2 Ktor	8
3.1.3 Architektur	9
3.2 Anpassung des zugrunde liegenden Quelltextes	9
3.2.1 Datenmodell prädikatenlogischer Formeln	9
3.2.2 Datenmodell gerichteter Graphen	10
3.2.3 Parser für prädikatenlogische Formeln	11
3.3 Implementierung des Algorithmus zur Modellprüfung	12
3.3.1 Generiertes Feedback	13
3.3.2 Varianten	14
3.3.3 Komplexität	14
3.4 Schnittstelle	15
4 Webanwendung	17
4.1 Technologischer Hintergrund	17
4.1.1 TypeScript	17
4.1.2 Angular	17
4.1.3 Angular Material	18
4.1.4 SCSS	19
4.1.5 RxJS	19
4.1.6 NgRx	19

4.1.7	ngx-translate	20
4.1.8	D3	20
4.2	Implementierung interaktiver Graphen mit D3	20
4.2.1	Konzept	20
4.2.2	Darstellung gerichteter Kanten	21
4.2.3	Erzeugen von Knoten und Kanten	23
4.2.4	Knotenanzordnung und Navigation	24
4.2.5	Touch-Steuerung	25
4.3	Funktionalitäten	26
4.3.1	Seitenleiste	26
4.3.2	Seite zur Verwaltung von Graphen	27
4.3.3	Seite für Modellprüfungen	28
4.3.4	Darstellung von generiertem Feedback	30
5	Fazit und Ausblick	32
5.1	Fazit	32
5.2	Ausblick	32
5.2.1	Aufgabenstellungen	32
5.2.2	Erweiterung und Analyse von Feedback	33
5.2.3	Analyse der Platzkomplexität	33
5.2.4	Heuristische Positionierung von Knoten und Kanten	34
A	Literatur	35
B	Anhang	38
B.1	Beispiele von Graphexporten	38
B.1.1	JSON Format	38
B.1.2	YAML Format	39

Abbildungsverzeichnis

2.1	Beispielerggebnis des Algorithmus zur Modellprüfung mit Tracing	6
3.1	Datenmodell der ursprünglichen Anwendung	10
3.2	Datenmodell der Prädikatenlogik	11
3.3	Datenmodell der Graphen	11
3.4	Datenmodell der Schnittstellen-Graphen	12
3.5	Datenmodell des Algorithmus-Feedbacks	13
3.6	Datenmodell der Schnittstelle	16
4.1	Graph mit verschiedenen Kanten	22
4.2	Pointer-Cursor für Interaktionen mit Kanten	24
4.3	Vorschau bei Kantenerzeugung	25
4.4	Seitenleiste der Webanwendung	27
4.5	Tabelle gespeicherten Graphen	27
4.6	Exportieren eines Graphs	28
4.7	Importfunktionen	28
4.8	Dialog zum Speichern von Graphen	29
4.9	Formulare zum Bearbeiten von Knoten und Kanten	29
4.10	Fortschrittsanzeige bei Modellprüfungen	29
4.11	„Vollständiges“ Feedback einer Modellprüfung	31
4.12	„Relevantes“ Feedback einer Modellprüfung	31
4.13	„Minimales“ Feedback einer Modellprüfung	31

Algorithmen und Quelltexte

2.1	Algorithmus zur Modellprüfung	5
2.2	Algorithmus zur Modellprüfung mit Tracing	7
4.1	Zähler-Beispiel mit Data Binding	18
4.2	Positionierung von Knoten	21
4.3	Positionierung von Kanten	22
4.4	Positionierung reflexiver Kanten	23
B.1	JSON Format eines Beispielgraphs	38
B.2	YAML Format eines Beispielgraphs	39

1 Einleitung

Das Fachgebiet Formale Methoden/Theoretische Informatik der Universität Kassel entwickelt eine Plattform für Anwendungen mit didaktischen Einsatzzwecken. Diese Bachelorarbeit befasst sich mit der Implementierung einer Webanwendung, welche Modellprüfungen für Graphstrukturen und Formeln der Prädikatenlogik erster Stufe durchführen kann und sich in diese Lernplattform integrieren lässt. Ziel dieser Webanwendung ist es, durch eine Visualisierung von Modellprüfungen die Semantik der Prädikatenlogik zu verdeutlichen und Studierenden so ein besseres Verständnis des Sachverhaltes zu ermöglichen.

1.1 Vorarbeiten

Im Rahmen eines Bachelorprojektes wurde von Ehle eine Desktopanwendung entwickelt, welche Modellprüfungen an Graphstrukturen durchführen kann [Ehl15]. Mithilfe dieser Anwendung lässt sich prüfen, ob von Nutzerinnen und Nutzern erstellte endliche Graphstrukturen Modelle von prädikatenlogischen Formeln sind. Nachträglich erweiterte Hruschka das Tool für ein Masterprojekt um einen Beweismodus [Hru16]. In diesem Modus können Modellprüfungen spielerisch durchgeführt werden, indem aus angezeigten Formelbäumen Unterformeln ausgewählt werden, welche eine zuvor aufgestellte Hypothese zum Wahrheitsgehalt belegen sollen. Die Anwendung bestätigt oder verneint solche Festlegungen anschließend und hilft Nutzerinnen und Nutzern so beim Finden eines Beweises. Abschließend werden Ergebnisse von Modellprüfungen ausgegeben. In einer zweiten Erweiterung entwickelte Hruschka die Anwendung erneut weiter. Im Rahmen seiner Masterarbeit implementierte er die Möglichkeit, Modellprüfungen auf einer Unterklasse der unendlichen Strukturen durchzuführen [Hru19].

1.2 Motivation und Ziele

Die zugrunde liegende Desktopanwendung wurde mit Java und der JavaFX-Plattform für Benutzeroberflächen entwickelt. Seit Java 11 ist JavaFX jedoch nicht mehr Bestandteil von Java-Installationen [Oraa]. Darüber hinaus ist es in den meisten OpenJDK-Versionen nicht enthalten, sondern Java Versionen von Oracle vorbehalten. Für eine Inbetriebnahme der Anwendung sind deshalb technische Kenntnisse zur manuellen Installation von JavaFX oder ein Lizenzvertrag mit Oracle notwendig [Orab]. Unter diesen Umständen entstand der

Bedarf, die Funktionalitäten der ursprünglichen Desktopanwendung über eine Webanwendung bereitzustellen. So würden nicht nur die beschriebene Installationshürden entfallen, sondern auch die Kernfunktionalitäten für Endgeräte ohne Java-Installationen verfügbar gemacht.

Ziel dieser Bachelorarbeit ist die Implementierung von Web- und Serveranwendung, welche äquivalent zur zugrunde liegenden Desktopanwendung Modellprüfungen von Graphstrukturen durchführen können. Dafür soll der Quelltext der ursprünglichen Desktopanwendung als Implementierungsgrundlage der Serveranwendung dienen und nach Bedarf angepasst und erweitert werden.

Die Webanwendung soll sowohl mit Maus- als auch mit Toucheingaben bedienbar sein, um eine Verwendung auf mobilen Endgeräten zu ermöglichen. Beweismodus und Erweiterung auf unendliche Strukturen sollen dabei nicht übernommen werden. Stattdessen soll der Algorithmus nun Feedback generieren, welches den Ausgang von Modellprüfungen begründet und einzelne Schritte visualisiert.

Die zugrunde liegende Desktopanwendung ermöglicht das Exportieren und Importieren von Graphen in einem Binärformat. Diese Funktionalität soll erhalten bleiben, jedoch auf die Klartextformate JSON und YAML umgestellt werden. Für eine vereinfachte Verwaltung von Graphen soll es zudem möglich sein, diese im LocalStorage eines Browsers zu hinterlegen.

Des Weiteren soll die Bedienung interaktiver Graphen vereinfacht werden. In der zugrunde liegenden Desktopanwendung muss zwischen dem Bearbeiten von Knoten und Kanten über Menüeintrag oder Tastenkürzel umgeschaltet werden. Dies ist insbesondere auf mobilen Endgeräten umständlich. Infolgedessen soll es jederzeit möglich sein, Kanten und Knoten zu erzeugen, löschen und bearbeiten.

Zuletzt soll die Webanwendung so implementiert werden, dass eine zukünftige Erweiterung durch Aufgabenstellungen möglich ist. Ein solcher Einsatz der Anwendung im Kontext der Lernplattform des Fachgebiets wurde bereits bei Beginn dieser Bachelorarbeit in Aussicht gestellt. Die Schnittstelle dafür wird in Unterabschnitt 5.2.1 beschrieben.

1.3 Struktur der Arbeit

Zunächst wird in Kapitel 2 der mathematische Hintergrund vorgestellt. Dabei werden einige grundlegende Begriffe definiert und ein Algorithmus zur Modellprüfung erläutert.

Anschließend wird in Kapitel 3 die entwickelte Serveranwendung beschrieben. Nach dem technischen Hintergrund werden Anpassungen des zugrunde liegenden Quelltextes und die Implementierung des Algorithmus für Modellprüfungen detailliert. Kapitel 4 befasst sich mit der entwickelten Webanwendung. Es werden verwendete Technologien sowie die Implementierung interaktiver Graphen und dabei aufgetretene Probleme vorgestellt. Anschließend werden die Funktionalitäten der Anwendung präsentiert. Dabei wird insbesondere auf die Visualisierung des Feedbacks von Modellprüfungen eingegangen. Kapitel 5 zieht ein Fazit und erläutert mögliche Erweiterungen von Web- und Serveranwendung.

2 Mathematischer Hintergrund

Dieses Kapitel stellt den mathematischen Hintergrund von Modellprüfungen vor und definiert dafür zunächst die Begriffe Signaturen, Strukturen und Interpretationen. In dieser Bachelorarbeit wird der Begriff Prädikatenlogik stellvertretend für die Prädikatenlogik erster Stufe mit Gleichheit verwendet. Eine formale Definition der Prädikatenlogik kann dem Skript entnommen werden [Lan17].

2.1 Signaturen und Strukturen

Signaturen sind Listen von Relations- und Funktionssymbolen inklusive ihrer Stelligkeiten. Sie legen Menge und Form der Relationen und Funktionen von Strukturen fest. In dieser Arbeit werden Signaturen der Form $\tau = \langle R_1^{st(R_1)}, \dots, R_n^{st(R_n)}, f_1^{st(f_1)}, \dots, f_m^{st(f_m)} \rangle$ betrachtet, wobei R_i und f_j Relations- beziehungsweise Funktionssymbole sind. Für die Stelligkeitsfunktion st gilt $st(R_i) \in \{1, 2\}$ und $st(f_j) \in \{0, 1\}$.

τ -Graphstrukturen sind Tupel der Form $\mathcal{G} = (\mathcal{N}, R_1^{\mathcal{G}}, \dots, R_n^{\mathcal{G}}, f_1^{\mathcal{G}}, \dots, f_m^{\mathcal{G}})$. Dabei ist \mathcal{N} eine nicht-leere, endliche Menge an Knoten, welche das Universum von \mathcal{G} bildet. $R_i^{\mathcal{G}}$ und $f_j^{\mathcal{G}}$ sind konkrete Relationen beziehungsweise Funktionen, deren Stelligkeiten durch eine τ -Signatur gegeben sind. Für R_i^1 und R_j^2 gilt jeweils $R_i \subseteq \mathcal{N}$ beziehungsweise $R_i^2 \subseteq \mathcal{N} \times \mathcal{N}$. Für f_n^0 sowie f_m^1 gilt $f_n^{\mathcal{G}} \in \mathcal{N}$ beziehungsweise $f_m^{\mathcal{G}} : \mathcal{N} \rightarrow \mathcal{N}$. Die gerichteten Kanten von Graphstrukturen repräsentieren jeweils eine Relations- oder Funktionsbeziehung zwischen ihrem Start- und Endknoten. Nullstellige Relationssymbole können durch die booleschen Konstanten „wahr“ (**tt**) und „falsch“ (**ff**) modelliert werden.

2.2 Interpretationen

Im Kontext dieser Bachelorarbeit sind Interpretationen definiert als $\mathcal{I} = \mathcal{G}$ und verfügen nicht über Variablenbelegungen. Variablen dürfen demzufolge nur in gebundener Form, also innerhalb von Quantoren, auftreten. Die so entfallenden freien Variablen können durch eine Verwendung von Konstantensymbolen in Formeln und Graphen ersetzt werden.

2.3 Modellprüfungen

[Lan17] Eine Modellprüfung ist die Auswertung des Wahrheitsgehaltes einer Formel φ im Kontext einer Interpretation \mathcal{G} . Bei einem positiven Ergebnis spricht man auch davon, dass \mathcal{G} Modell von φ ist und schreibt $\mathcal{G} \models \varphi$. Im konträren Fall schreibt man $\mathcal{G} \not\models \varphi$.

Der Algorithmus verfährt nach dem Top-Down-Prinzip und überprüft Formeln gemäß ihrer Semantik. Bei nicht-atomaren Formeln finden rekursive Aufrufe statt, welche unmittelbare Unterformeln überprüfen. Dieses Konzept ist in Algorithmus 2.1 zu sehen. Die in Zeile 1 definierte Menge an Variablenbelegungen \mathcal{V} dient dabei zur Speicherung aktueller Belegungen und wird mit der leeren Menge initialisiert. Relevant ist sie für die Auswertung von Termen in Kombination mit einem Graphen, wie es in den Zeilen 8, 10 und 12 stattfindet.

Algorithmus 2.1 Algorithmus zur Modellprüfung

```

1: procedure MC( $\mathcal{G}, \varphi, \mathcal{V} = \emptyset$ )
2:   switch  $\varphi$  do
3:     case tt return true
4:     case ff return false
5:     case  $a \doteq b$  return  $\llbracket a \rrbracket_{\mathcal{V}}^{\mathcal{G}} == \llbracket b \rrbracket_{\mathcal{V}}^{\mathcal{G}}$            ▷ Rückgabewert ist true oder false
6:     case  $R(a)$  return  $\llbracket a \rrbracket_{\mathcal{V}}^{\mathcal{G}} \in R^{\mathcal{G}}$            ▷ Rückgabewert ist true oder false
7:     case  $R(a, b)$  return  $(\llbracket a \rrbracket_{\mathcal{V}}^{\mathcal{G}}, \llbracket b \rrbracket_{\mathcal{V}}^{\mathcal{G}}) \in R^{\mathcal{G}}$    ▷ Rückgabewert ist true oder false
8:     case  $\neg\psi$ 
9:       return  $\neg\text{MC}(\mathcal{G}, \psi, \mathcal{V})$ 
10:    case  $\psi \vee \chi$ 
11:      return  $\text{MC}(\mathcal{G}, \psi, \mathcal{V}) \vee \text{MC}(\mathcal{G}, \chi, \mathcal{V})$ 
12:    case  $\psi \wedge \chi$ 
13:      return  $\text{MC}(\mathcal{G}, \psi, \mathcal{V}) \wedge \text{MC}(\mathcal{G}, \chi, \mathcal{V})$ 
14:    case  $\psi \rightarrow \chi$ 
15:      return  $\neg\text{MC}(\mathcal{G}, \psi, \mathcal{V}) \vee \text{MC}(\mathcal{G}, \chi, \mathcal{V})$ 
16:    case  $\psi \leftrightarrow \chi$ 
17:       $left \leftarrow \text{MC}(\mathcal{G}, \psi, \mathcal{V})$ 
18:       $right \leftarrow \text{MC}(\mathcal{G}, \chi, \mathcal{V})$ 
19:      return  $left \wedge right \vee \neg left \wedge \neg right$ 
20:    case  $\exists x.\psi$ 
21:      for  $node$  in  $\text{GETNODES}(\mathcal{G})$  do
22:         $result \leftarrow \text{MC}(\mathcal{G}, \psi, \mathcal{V} \cup \{x \mapsto node\})$ 
23:        if  $result$  then
24:          return true
25:        return false
26:    case  $\forall x.\psi$ 
27:      for  $node$  in  $\text{GETNODES}(\mathcal{G})$  do
28:         $result \leftarrow \text{MC}(\mathcal{G}, \psi, \mathcal{V} \cup \{x \mapsto node\})$ 
29:        if  $\neg result$  then
30:          return false
31:      return true

```

2.4 Erweiterung des Algorithmus zur Modellprüfung

Der Algorithmus wurde so erweitert, dass Ausführungen mithilfe von *Tracing* nachvollziehbar werden. Der modifizierte Algorithmus zur Modellprüfung mit Tracing ist in Algorithmus 2.2 zu sehen. Er unterscheidet sich insbesondere in Rückgabewerten, welche Tupel der Form $(\varphi, isModel, children)$ sind. Dabei ist φ eine überprüfte Formel, $isModel$ das boolesche Ergebnis einer Modellprüfung und $children$ eine Menge an Traces von Unterformeln. Bei atomaren Formeln gilt folglich $children = \emptyset$.

Durch die Inklusion von Unterergebnissen entstehen Bäume, deren Knoten jeweils einen Trace repräsentieren. Im Folgenden werden solche Baumstrukturen auch als Feedback-Bäume bezeichnet, da sie Feedback über Ausführungen des Algorithmus zur Modellprüfung enthalten. Traces von atomaren Formeln bilden Blätter und ursprüngliche Eingaben des Algorithmus Wurzeln.

Eine Visualisierung einer solchen Baumstruktur ist in Abbildung 2.1 zu sehen. Dort abgebildet ist das Ergebnis einer Durchführung des Algorithmus mit der Eingabe $\mathcal{G} = (\mathcal{N} = \{0, 1\}, R = \{(0, 1)\}, a = 0)$ und $\varphi = \forall x. R(a, x) \vee a \doteq x$. Zur Verdeutlichung wurden gebundene Variablen durch vorhandene Variablenbelegungen ersetzt, welche wiederum als Kantenbeschriftungen dienen. Wie der Wurzel des Baumes zu entnehmen ist, gilt $\mathcal{G} \models \varphi$. Durch das Tracing lässt sich nachverfolgen, warum dies der Fall ist. An den Blättern lässt sich ablesen, dass bei der Variablenbelegung $x \mapsto 0$ die (Unter-)Formel $a \doteq x$ gilt. Für die Variablenbelegung $x \mapsto 1$ gilt hingegen das erste Disjunkt $R(a, x)$. Somit gilt in beiden Fällen die Disjunktion und infolgedessen auch der Allquantor.

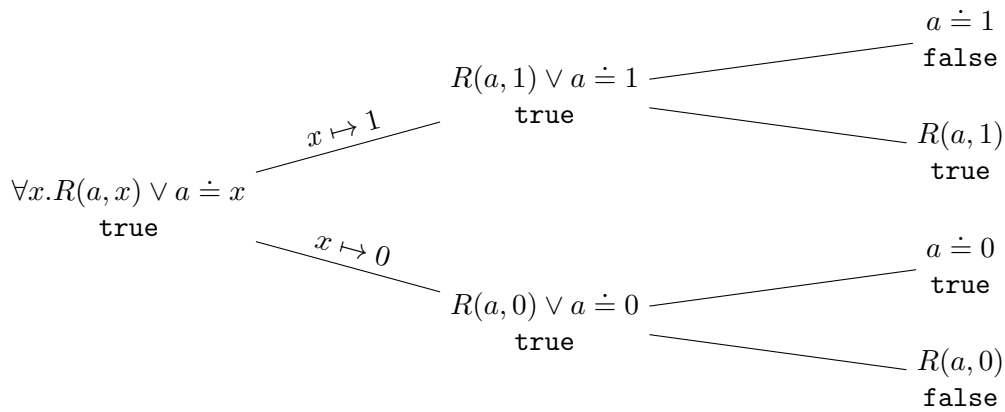


Abbildung 2.1: Beispielergebnis des Algorithmus zur Modellprüfung mit Tracing

Algorithmus 2.2 Algorithmus zur Modellprüfung mit Tracing

```

1: procedure MCT( $\mathcal{G}, \varphi, \mathcal{V} = \emptyset$ )
2:   switch  $\varphi$  do
3:     case tt return ( $\varphi, \text{true}, \emptyset$ )
4:     case ff return ( $\varphi, \text{false}, \emptyset$ )
5:     case  $a \doteq b$  return ( $\varphi, \llbracket a \rrbracket_{\mathcal{V}}^{\mathcal{G}} == \llbracket b \rrbracket_{\mathcal{V}}^{\mathcal{G}}, \emptyset$ )
6:     case  $R(a)$  return ( $\varphi, \llbracket a \rrbracket_{\mathcal{V}}^{\mathcal{G}} \in R^{\mathcal{G}}, \emptyset$ )
7:     case  $R(a, b)$  return ( $\varphi, (\llbracket a \rrbracket_{\mathcal{V}}^{\mathcal{G}}, \llbracket b \rrbracket_{\mathcal{V}}^{\mathcal{G}}) \in R^{\mathcal{G}}, \emptyset$ )
8:     case  $\neg\psi$ 
9:        $child \leftarrow \text{MCT}(\mathcal{G}, \psi, \mathcal{V})$ 
10:      return ( $\varphi, \neg child.isModel, \{child\}$ )
11:     case  $\psi \vee \chi$ 
12:        $left \leftarrow \text{MCT}(\mathcal{G}, \psi, \mathcal{V})$ 
13:        $right \leftarrow \text{MCT}(\mathcal{G}, \chi, \mathcal{V})$ 
14:        $isModel \leftarrow left.isModel \vee right.isModel$ 
15:       return ( $\varphi, isModel, \{left, right\}$ )
16:     case  $\psi \wedge \chi$ 
17:        $left \leftarrow \text{MCT}(\mathcal{G}, \psi, \mathcal{V})$ 
18:        $right \leftarrow \text{MCT}(\mathcal{G}, \chi, \mathcal{V})$ 
19:        $isModel \leftarrow left.isModel \wedge right.isModel$ 
20:       return ( $\varphi, isModel, \{left, right\}$ )
21:     case  $\psi \rightarrow \chi$ 
22:        $left \leftarrow \text{MCT}(\mathcal{G}, \psi, \mathcal{V})$ 
23:        $right \leftarrow \text{MCT}(\mathcal{G}, \chi, \mathcal{V})$ 
24:        $isModel \leftarrow \neg left.isModel \vee right.isModel$ 
25:       return ( $\varphi, isModel, \{left, right\}$ )
26:     case  $\psi \leftrightarrow \chi$ 
27:        $left \leftarrow \text{MCT}(\mathcal{G}, \psi, \mathcal{V})$ 
28:        $right \leftarrow \text{MCT}(\mathcal{G}, \chi, \mathcal{V})$ 
29:        $isModel \leftarrow left.isModel \wedge right.isModel \vee \neg left.isModel \wedge \neg right.isModel$ 
30:       return ( $\varphi, isModel, \{left, right\}$ )
31:     case  $\exists x.\psi$ 
32:        $results \leftarrow \emptyset$ 
33:       for  $node$  in GETNODES( $\mathcal{G}$ ) do
34:          $result \leftarrow \text{MCT}(\mathcal{G}, \psi, \mathcal{V} \cup \{x \mapsto node\})$ 
35:         if  $result.isModel$  then
36:           return ( $\varphi, \text{true}, \{result\}$ )
37:          $results \leftarrow results \cup \{result\}$ 
38:       return ( $\varphi, \text{false}, results$ )
39:     case  $\forall x.\psi$ 
40:        $results \leftarrow \emptyset$ 
41:       for  $node$  in GETNODES( $\mathcal{G}$ ) do
42:          $result \leftarrow \text{MCT}(\mathcal{G}, \psi, \mathcal{V} \cup \{x \mapsto node\})$ 
43:         if  $\neg result.isModel$  then
44:           return ( $\varphi, \text{false}, \{result\}$ )
45:          $results \leftarrow results \cup \{result\}$ 
46:       return ( $\varphi, \text{true}, results$ )

```

3 Serveranwendung

Die Serveranwendung, im Folgenden als *Backend* bezeichnet, ist für die Durchführung des Algorithmus zur Modellprüfung zuständig. Dazu wurde Quelltext der zugrunde liegenden Desktopanwendung verwendet, angepasst und erweitert. Dieses Kapitel befasst sich mit den Eigenschaften sowie der Entwicklung des Backends. Nach einer einführenden Dokumentation werden Änderungen am übernommenen Quelltext sowie die Re-Implementierung des Algorithmus zur Modellprüfung vorgestellt.

3.1 Technologischer Hintergrund

Das Backend ist eine Java Virtual Machine Anwendung, welche in der Programmiersprache Kotlin geschrieben wurde. Die folgenden Unterabschnitte stellen verwendete Technologien sowie die Architektur des Backends vor.

3.1.1 Kotlin

Kotlin ist eine Multiparadigmensprache und unterstützt eine Vielzahl an Plattformen, darunter auch die Java Virtual Machine [Jet20b]. Sie zeichnet sich nicht zuletzt durch ihre Kompatibilität mit Java Quelltext aus. Aufgrund dieser Eigenschaft und der resultierenden Verwendbarkeit von Java Bibliotheken eignet sich Kotlin insbesondere zur Entwicklung von Serveranwendungen. Verglichen mit Java lassen sich Programme mit Kotlin nicht nur kürzer (bezüglich der Quelltextlänge), sondern auch sauberer implementieren [Fla+18]. Die Wahl der Programmiersprache fiel deshalb auf Kotlin. Der ursprüngliche Java Quelltext wurde mithilfe des Übersetzungswerkzeugs der Entwicklungsumgebung IntelliJ IDEA Ultimate nach Kotlin migriert.

3.1.2 Ktor

Um das Potenzial der Programmiersprache Kotlin optimal zu nutzen, wurde das Serverframework Ktor verwendet. Sowohl Kotlin als auch Ktor werden von der Firma JetBrains entwickelt. Ktor setzt auf eine vollständige Asynchronität durch einen durchgängigen Einsatz von Kotlin Coroutines [Jet20a].

Zur Konfiguration des Backends wird eine *Domain Specific Language* verwendet. Mit dieser lassen sich Endpunkte strukturiert definieren und Funktionalitäten, wie beispielsweise Serialisierung und Logging, konfigurieren. Ktor-Anwendungen setzen sich aus Modulen zu-

sammen, welche sich auch unabhängig voneinander verwenden lassen. So können einzelne Teile der Funktionalität separat getestet werden.

3.1.3 Architektur

Das Backend verfügt lediglich über einen Endpunkt `/modelchecker`, welcher zum Aufrufen des Algorithmus für Modellprüfungen dient. Es ist keine Persistenz erforderlich, da Anfragen in sich geschlossen sind und zustandslos durchgeführt werden. Das heißt Anfragen sind unabhängig von vorherigen und parallelen Anfragen.

Um potenzielle Erweiterungen des Backends zu ermöglichen, wurde trotzdem eine Architekturgrundlage gelegt. Die Wahl fiel auf die Service-Repository-Architektur. Bei dieser sind *Services* für die Durchführung der Geschäftslogik verantwortlich. *Repositories* fungieren als reine Datenspeicher und *Single Source of Truth*. Nur sie entscheiden über den Zustand persistenter Daten, sodass die Anfälligkeit für Widersprüche mehrerer Datenquellen vermieden wird [PS14]. Sowohl Services als auch Repositories verwenden dabei Interfaces, um Implementierungen, insbesondere beim Testen, austauschen zu können.

Implementiert wurde ein Dienst für Modellprüfungen. Dieser validiert Anfragen, führt den Algorithmus zur Modellprüfung aus und konstruiert zugehörige Antworten. Bereitgestellt wird dieser Dienst mithilfe des leichtgewichtigen *Dependency Injection* Frameworks Koin [Koi21]. Durch dessen Ktor-Integration lassen sich Dienste gezielt für einzelne Endpunkte zugänglich machen.

3.2 Anpassung des zugrunde liegenden Quelltextes

Vom Quelltext der ursprünglichen liegenden Desktopanwendung wurde ein Parser für Formeln der Prädikatenlogik erster Stufe mit kleinen Änderungen übernommen. Der Algorithmus zur Modellprüfung sowie das Datenmodell von Parser und Algorithmus wurden gänzlich neu implementiert. Die folgenden Unterabschnitte stellen Änderungen an Datenmodell und Parser des zugrunde liegenden Quelltextes vor. Der Algorithmus zur Modellprüfung sowie dessen Neuimplementierung werden in Abschnitt 3.3 behandelt.

3.2.1 Datenmodell prädikatenlogischer Formeln

Abbildung 3.1 zeigt das Klassendiagramm des ursprünglichen Datenmodells, wobei blaue Pfeile eine Vererbungsbeziehung repräsentieren und von Kindes- nach Elternklassen gerichtet sind. Dieses und folgende Diagramme wurden mit der Entwicklungsumgebung IntelliJ

IDEA Ultimate generiert [Jet21]. Wie dort zu sehen ist, haben Terme in Form der Klassen `FOLFunction` und `FOLBoundVariable` die Elternklasse `FOLFormula`, obwohl es sich bei Termen nicht um Formeln handelt. Aufgrund dieser Umstände wird die Vererbungshierarchie nicht für eine Ausführungslogik verwendet. Stattdessen wird mithilfe des Feldes `type` der Klasse `FOLFormula` im Algorithmus zwischen verschiedenen Bestandteilen wie Operatoren, Variablen oder Funktionen unterschieden.

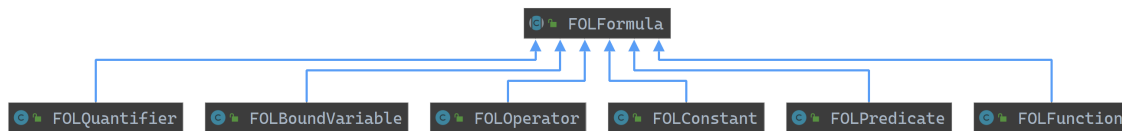


Abbildung 3.1: Datenmodell der ursprünglichen Anwendung

Das Konzept des ursprünglichen Datenmodells wurde deshalb verworfen und durch eine Vererbungshierarchie ersetzt, welche für den Kontrollfluss des Algorithmus zur Modellprüfung verwendet werden kann. Das neu implementierte Datenmodell differenziert insbesondere zwischen Formeln und Termen, da die Verwendung beider Konzepte sich gänzlich unterscheidet. Bei Ersteren ist eine Anwendung des Algorithmus zur Modellprüfung der essenzielle Aspekt. Auf Terme trifft dies nicht zu, da ein alleinstehender Term keine gültige Formel der Prädikatenlogik erster Stufe ist. Stattdessen müssen sich Terme im Kontext von Strukturen, in diesem Fall von Graphen, evaluieren beziehungsweise interpretieren lassen.

Insbesondere wurde Vererbung in Kombination mit *Sealed Classes* eingesetzt. Dabei handelt es sich um besondere Klassen der Programmiersprache Kotlin, welche nur spezifizierte Kindesklassen zulassen. So ermöglicht das neue Datenmodell beispielsweise eine Unterscheidung zwischen unären und binären Operatoren anhand ihrer Klassen. Gleichzeitig lässt sich Logik in dieser Hierarchie auf solche Elternklassen, wie beispielsweise binäre Operatoren, auslagern, um Dopplungen zu vermeiden. Die entstandene Klassenhierarchie ist in Abbildung 3.2 zu sehen, wobei innere Klassen für eine bessere Übersichtlichkeit ausgeblendet wurden. Insbesondere ist die Differenzierung zwischen Formeln und Termen zu erkennen.

3.2.2 Datenmodell gerichteter Graphen

Auch das Datenmodell der Graphen wurde angepasst. Das Feld `stringAttachments` wurde durch die semantisch aussagekräftigeren Felder `relations` und `functions` beziehungsweise `constants` ersetzt. Bidirektionale Beziehungen wurden gänzlich entfernt, da diese aus-

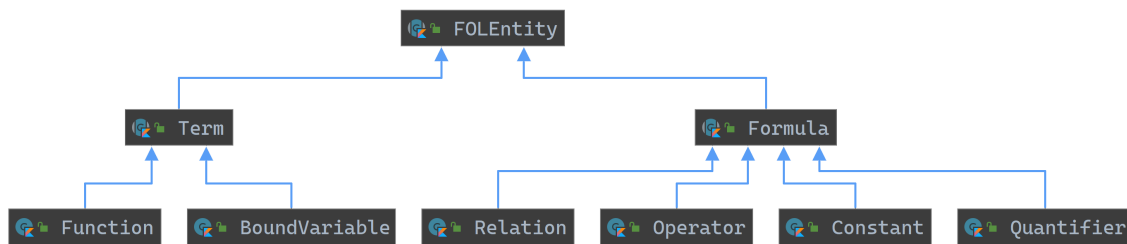


Abbildung 3.2: Datenmodell der Prädikatenlogik

schließlich für die Benutzeroberfläche der zugrunde liegenden Desktopanwendung verwendet wurden. Das resultierende Datenmodell der Graphen sowie ihre Knoten und Kanten, zu sehen in Abbildung 3.3, verfügt dementsprechend nur noch über unveränderliche Felder und Datenstrukturen.

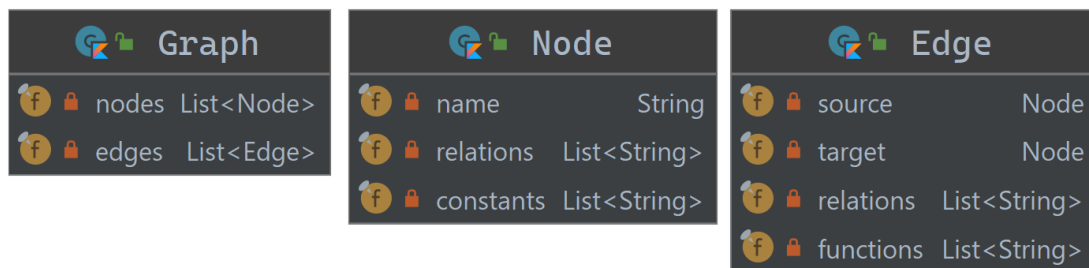


Abbildung 3.3: Datenmodell der Graphen

Um die Größe von Anfragen und Antworten gering zu halten wurde ein zweites Datenmodell implementiert, welches in Abbildung 3.4 dargestellt ist. Bei diesem haben Kanten keine Referenzen auf Start- und Endknoten, sondern deren Namen. Infolgedessen lässt sich dieses Datenmodell besser (de-)serialisieren, da Knotendaten nur einmalig und nicht redundant als Felder aller aus- und eingehenden Kanten vorhanden sind. Um es im Algorithmus zur Modellprüfung einzusetzen, muss dieses Datenmodell zunächst transformiert werden. Die dafür implementierte Funktion führt gleichzeitig eine Validierung des Modells durch. Insbesondere wird dabei sichergestellt, dass mindestens ein Knoten existiert und somit eine gültige Struktur vorliegt.

3.2.3 Parser für prädikatenlogische Formeln

Der Parser für prädikatenlogische Formeln wurde mit kleinen Änderungen übernommen. Die Unterstützung der Infixnotation von Funktionen wurde gänzlich entfernt. Im Kontext der unterstützten Graphen sind diese ohnehin nicht anwendbar, da Kanten lediglich einen Ursprung haben und dementsprechend auch nur Funktionen mit einem Argument abbilden können. Darüber hinaus wurde der Parser auf das neue Datenmodell angepasst, sodass

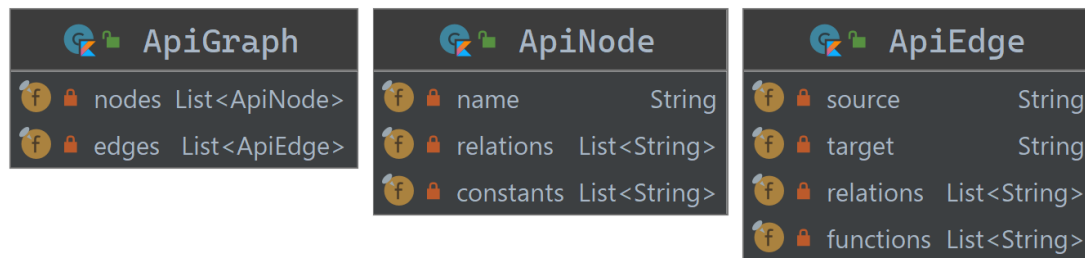


Abbildung 3.4: Datenmodell der Schnittstellen-Graphen

Validierungen der Eingabe verändert werden mussten. Sowohl die ursprüngliche als auch die neue Anwendung erlauben nullstellige Relationen nur in Form der booleschen Konstanten `tt` und `ff`. Die gegebene Implementierung des Parsers erlaubt jedoch eine Verwendung beliebiger nullstelliger Relationen in Formeln.

Die Klasse `FOLParser` ist zudem nicht für eine parallele Ausführung geeignet, da Daten von Ausführungen in Klassenvariablen verwaltet werden. Sie wurde infolgedessen so angepasst, dass Eingaben bei Erzeugung einer Instanz der Klasse unmittelbar geparkt werden. Dementsprechend sind Instanzen des Parsers nicht wiederverwendbar. Um dies zu verdeutlichen, wurden jegliche Zugriffe auf den Parser mithilfe einer Methode verkapselt.

Übersetzungstexte von Fehlermeldungen des Parsers befinden sich im Backend, da es der übernommene Quelltext so vorsieht. Dazu werden deutsche und englische Übersetzungsdateien einmalig geladen. Übersetzungen lassen sich anschließend über ihre zugehörigen Schlüssel lesen.

3.3 Implementierung des Algorithmus zur Modellprüfung

Die zugrunde liegende Desktopanwendung weist einen Algorithmus zur Modellprüfung auf, wie er in Abschnitt 2.3 beschrieben wird. Mit dem Ziel, hilfreiches und detailliertes Feedback zu generieren, wurde die in Abschnitt 2.4 detaillierte Erweiterung des Algorithmus als Implementierungsgrundlage verwendet. Dazu wurde die ursprüngliche Implementierung verworfen. Modellprüfungsmethoden wurden in die Klasse `Formula` beziehungsweise ihre Unterklassen ausgelagert. Funktionen zur Validierung von Eingaben wurden an das neue Datenmodell angepasst. Sie validieren Gültigkeit, Kompatibilität und Vollständigkeit der durch Graphen und Formeln definierten Symbole. Dabei wird die Totalität aller Funktionen, eine einheitliche Verwendung von Symbolen und eine Definition aller Konstanten sichergestellt.

3.3.1 Generiertes Feedback

Der implementierte Algorithmus erweitert die in Abschnitt 3.3 definierten Traces um weitere Informationen. `ModelCheckerTraces` verfügen über Übersetzungsschlüssel, welche eine Anzeige von Erklärungstexten zu Ergebnissen in der Webanwendung ermöglichen.

Für informatives Feedback wurde zudem die String-Repräsentation aller Formelbestandteile so angepasst, dass sich gebundene Variablen durch ihre Variablenbelegungen ersetzen lassen. So ist es möglich, Unterformeln von Quantoren während einer Durchführung des Algorithmus zur Modellprüfung mit Variablenbelegungen darzustellen (siehe Unterabschnitt 4.3.4).

Der Algorithmus wurde so implementiert, dass er positive Ergebnisse der Form $\mathcal{G} \models \varphi$ erwartet. Bei Formeln der Form $\varphi = \neg\psi$ wird jedoch das erwartete Ergebnis invertiert. Ein rekursiver Aufruf des Algorithmus zur Modellprüfung überprüft folglich, ob $\mathcal{G} \models \psi$ gilt. Um dies zu ermöglichen, wird das erwartete Ergebnis bei jedem Aufruf angegeben und bei Überprüfungen von Negationsoperatoren invertiert.

Besonders zu betrachten sind Bi-Implikationen $\varphi = \psi \leftrightarrow \chi$. Das erwartete Ergebnis der Modellprüfungen von ψ und χ kann sowohl positiv als auch negativ sein. Infolgedessen müssen beide Fälle betrachtet werden. Zuerst wird geprüft, ob $\mathcal{G} \models \psi$ und $\mathcal{G} \models \chi$ oder $\mathcal{G} \not\models \psi$ und $\mathcal{G} \not\models \chi$ gelten. Trifft einer der beiden Fälle zu, werden dessen Traces dem Feedback-Baum hinzugefügt und es gilt $\mathcal{G} \models \varphi$. Sollte weder noch zutreffen, gilt $\mathcal{G} \not\models \varphi$. In diesem Fall wird dem Feedback-Baum ein Knoten hinzugefügt, welcher die Traces beider Fälle als Kinder besitzt.

Das resultierende Datenmodell der Klasse `ModelCheckerTrace` ist in Abbildung 3.5 dargestellt. Das Feld `shouldBeModel` entspricht einem erwarteten Ergebnis. Übersetzungsschlüssel und Parameter zur Formatierung von Texten werden in der Klasse `TranslationDTO` zusammengefasst.








ModelCheckerTrace			TranslationDTO		
	formula	String		key	String
	description	TranslationDTO		params	Map<String, String>
	isModel	boolean			
	shouldBeModel	boolean			
	children	List<ModelCheckerTrace>			

Abbildung 3.5: Datenmodell des Algorithmus-Feedbacks

3.3.2 Varianten

Der Algorithmus zur Modellprüfung wurde in verschiedenen Varianten implementiert, welche den drei verfügbaren Feedback-Optionen „Vollständig“, „Relevant“ und „Minimal“ entsprechen.

Die letzten beiden Varianten führen, wie auch der ursprüngliche Algorithmus, nur notwendige Modellprüfungen durch. Das heißt, bei binären Operatoren wird zuerst nur eine Unterformel geprüft. Nur wenn es ihre Semantik in Kombination mit dem Ergebnis der ersten Prüfung erfordert, wird auch ihre zweite Unterformel geprüft. Bei Existenzquantoren wird eine Modellprüfung frühzeitig abgebrochen, sobald eine gültige Variablenbelegung gefunden wird. Analog dazu bricht eine Modellprüfung von Allquantoren frühzeitig ab, falls eine Variablenbelegung ungültig ist.

In der „minimalen“ Variante werden alle Knoten von Feedback-Bäumen, abgesehen von Wurzeln, entfernt. Diese Variante entspricht dem Feedback der zugrunde liegenden Anwendung und wurde implementiert, um eine Verwendung bei niedrigen Internet-Bandbreiten zu ermöglichen.

Die „vollständige“ Variante führt hingegen redundante Modellprüfungen durch. Bei binären Operatoren werden immer beide Unterformeln geprüft. Modellprüfungen von Existenz- und Allquantoren brechen niemals frühzeitig ab, stattdessen werden alle möglichen Variablenbelegungen betrachtet.

3.3.3 Komplexität

Offensichtlich haben Quantoren die höchste Komplexität aller Operatoren, da sie in Feedback-Bäumen $|\mathcal{N}|$ Verzweigungen verursachen (\mathcal{N} ist die Knotenanzahl eines Graphs). Binäre Operatoren resultieren hingegen nur in einer zweifachen Verzweigung, während Negationsoperatoren keine Verzweigungen verursachen. Die bekannte Formel zur Berechnung der Knotenanzahl \mathcal{K} eines Baumes der Tiefe d mit n Verzweigungen bei jedem Knoten lautet $\mathcal{K} = (n^{d+1} - 1) / (n - 1)$. Weil jeder Baumknoten einen Berechnungsschritt des Algorithmus repräsentiert, folgt, dass sowohl Laufzeit als auch Platzverbrauch in $O(n^d)$ liegen, wobei im Kontext von Graphstrukturen $n = |\mathcal{N}|$ gilt und d die Tiefe des Syntaxbaumes einer Formel ist.

Bei der „vollständigen“ Variante des Algorithmus zur Modellprüfung kann dies problematisch werden, da ihre worst-case- und best-case-Komplexitäten übereinstimmen. Dies folgt

aus der Tatsache, dass bei Quantoren alle möglichen Variablenbelegungen und bei binären Operatoren in jedem Fall beide Operanden überprüft werden.

Im folgenden Beispiel werden ein beliebiger Graph mit vierzig Knoten und die Formel $\exists x.\exists y.\exists z.\mathbf{tt}$ betrachtet. Die serialisierte Antwort des Servers ist für die Feedback-Option „Relevant“ lediglich 598 Bytes groß. Bei der „vollständigen“ Varianten resultiert dieses Beispiel jedoch in einer 5.98 Megabyte großen Antwort. Es ist absehbar, dass komplexere Formeln oder tiefer verschachtelte Quantoren zu serialisierten Antworten führen, welche für einen realistischen Einsatz in Webanwendungen zu groß sind.

Problematisch kann dieses Verhalten jedoch auch auf Seite des Backends werden. Abhängig von der Konfiguration der ausführenden Java Virtual Machine, kann bei Modellprüfung und Serialisierung ein `OutOfMemoryError` geworfen werden. Auf Grund der Verwendung von Ktor führt dieser nicht zu einem Absturz des Backends, der Fehler wird vom Framework jedoch auch nicht zuverlässig geworfen. Sollte er gefangen werden, wird eine entsprechende Fehlermeldung an den Client der zugehörigen Anfrage gesendet. Gelingt es nicht, erhält der Client eine unvollständige Antwort. Die Fehlerbehandlung findet dann in der Webanwendung statt. In jedem Fall wird dort eine Nachricht angezeigt, welche über aufgetretene Fehler informiert. In Unterabschnitt 5.2.3 wird eine mögliche Anpassung der Webanwendung mit dem Ziel, die Integration der „vollständigen“ Variante des Algorithmus für Modellprüfungen zu verbessern, vorgestellt.

3.4 Schnittstelle

Die Schnittstelle der Serveranwendung bildet der Endpunkt `/modelchecker`. Er erwartet Anfragen, deren Schema in Abbildung 3.6 zu sehen ist sind. Das Feld `language` der Klasse `ModelCheckerRequest` kann dabei die Werte `de` oder `en` annehmen, um eine Verwendung der deutschen beziehungsweise englischen Sprache für Fehlermeldungen des Parsers vorzugeben. Mittig sind die Feedback-Optionen abgebildet. Im JSON Format können diese über die Zeichenketten `full`, `relevant` oder `minimal` angegeben werden. Zuletzt ist rechts das Schema einer Antwort dargestellt. Dieses enthält den Wurzel-`ModelCheckerTrace` einer Durchführung des Algorithmus zur Modellprüfung sowie die Feedback-Option, mit welcher der Algorithmus durchgeführt wurde. Sollten bei einer Anfrage Fehler auftreten, beispielsweise durch invalide Graphen oder Formeln, werden diese als `TranslationDTO` unmittelbar versendet und die Durchführung des Algorithmus terminiert.

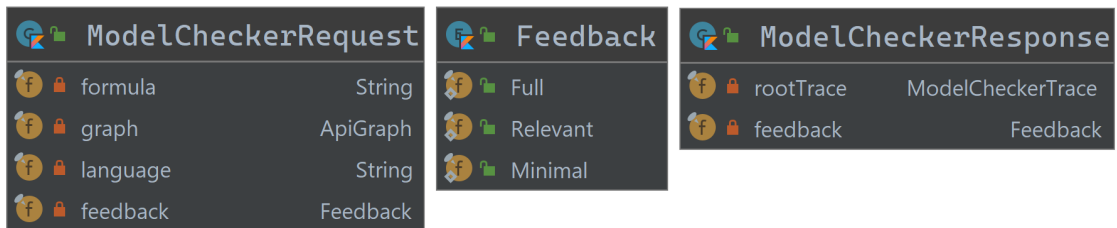


Abbildung 3.6: Datenmodell der Schnittstelle

4 Webanwendung

Als Ersatz für die Benutzeroberfläche der zugrunde liegenden Desktopanwendung wurde die Webanwendung entwickelt. Dieses Kapitel erläutert zunächst den technologischen Hintergrund der Webanwendung. Anschließend werden interaktive Graphen und Visualisierungen von Feedback detailliert. Zum Abschluss werden implementierte Funktionalitäten vorgestellt.

4.1 Technologischer Hintergrund

Die Webanwendung ist eine *Single Page Application*, geschrieben in der Programmiersprache TypeScript. Die folgenden Unterabschnitte detaillieren verwendete Technologien und Bibliotheken.

4.1.1 TypeScript

Bei TypeScript handelt es sich um eine Erweiterung der Programmiersprache JavaScript [Mic21]. Dem Namen entsprechend unterstützt TypeScript eine statische Typisierung. Typen lassen sich während Programmierung und Transpilierung¹ prüfen, sodass sich ungültige Verwendungen vermeiden lassen [BAT14]. Das Ergebnis einer Transpilierung ist JavaScript Quelltext, der sich in Browsern ausführen lässt. TypeScript eignet sich infolgedessen zur Entwicklung von Webanwendungen.

4.1.2 Angular

Angular ist ein Framework für Webanwendungen, dessen Entwicklung von Google unterstützt wird. Basierend auf TypeScript, der Node.js-Laufzeitumgebung und einer eigenen Kommandozeile ermöglicht Angular eine schnelle Entwicklung moderner Webanwendungen. Die Kommandozeile erlaubt es *Boilerplate*² und Dateien generieren zu lassen und ermöglicht so eine Erhöhung der Produktivität. Darüber hinaus existiert eine Vielzahl an Bibliotheken, welche die ohnehin umfangreiche Funktionalität des Frameworks erweitern [SO18]. Eine detaillierte Dokumentation kann der offiziellen Website entnommen werden [Goo20b].

¹Übersetzung einer Programmiersprache in eine andere Sprache.

²Quelltext, welcher mit nur kleinen Anpassungen an mehreren Stellen verwendet wird. Bei Angular sind dies vor allen Dingen Komponentendeklarationen.

Benutzeroberflächen werden in HTML geschrieben und verwenden *Data Binding* um eine selbstständige Aktualisierung bei Änderungen am Datenmodell zu ermöglichen. So werden programmatische Zugriffe auf HTML-Elemente in den meisten Anwendungsfällen überflüssig. Quelltext 4.1 zeigt ein Zähler-Beispiel mit Data Binding. Die Komponente `CounterComponent` enthält einen Button, welcher in den Zeilen 5 bis 7 definiert wird. Die Variable `counter` wird in Zeile 10 mit dem Wert „0“ initialisiert und bei jedem Klick auf den Button inkrementiert. Der aktuelle Zählerstand wird im Text des Buttons angezeigt und nach jeder Veränderung der Variable `counter` automatisch aktualisiert. Dies passiert mithilfe der Notation mit geschweiften Klammern, welche in Zeile 6 zu sehen ist. Zuvor wird in Zeile 5 das `click`-Event des Buttons mit der Methode `incrementCounter()` verbunden.

```
1 import {Component} from '@angular/core';
2
3 @Component({
4   selector: 'counter',
5   template: '<button (click)="incrementCounter()">'
6             + '{{ counter }}'
7             + '</button>',
8 })
9 export class CounterComponent {
10   counter = 0;
11
12   incrementCounter() {
13     this.counter++;
14   }
15 }
```

Quelltext 4.1: Zähler-Beispiel mit Data Binding

4.1.3 Angular Material

Bei Angular Material handelt es sich um eine ebenfalls von Google entwickelte Bibliothek für Benutzeroberflächenkomponenten und -werkzeuge [Goo20a]. Diese folgen den Grundlagen des Material Designs, welches Darstellung und Verhalten von Elementen vorgibt. Die umfangreiche Bibliothek bietet neben Komponenten für Nutzereingaben, Dialogen und Navigation auch Implementierungsgrundlagen für nicht-triviale Funktionalitäten. Dazu zählen beispielsweise sortierbare Tabellen, *Drag and Drop* Unterstützung für beliebige HTML-Elemente sowie anpassbare, dynamische Tooltips. Bei allen Komponenten ist zudem eine Verwendbarkeit bei Toucheingaben garantiert, weshalb sie insbesondere für mobile Endgeräte geeignet sind.

4.1.4 SCSS

[Sas21] Angular unterstützt zum Designen von Benutzeroberflächen reguläres CSS und Erweiterungen wie SASS sowie dessen alternative Syntax SCSS. Für diese Webanwendung wurde SCSS verwendet, da es für eine Anpassung von Angular Material Komponenten erforderlich ist und zusätzliche Funktionalitäten bietet. Bei SCSS lassen sich unter anderem Variablen definieren, welche anders als reguläre CSS-Variablen zur Kompilierzeit ersetzt werden. Darüber hinaus können mithilfe von Funktionen, Parametern und Operatoren komplexe, dynamische Designs implementiert werden. Im Rahmen dieser Bachelorarbeit wurden je ein dunkles und helles Design implementiert, welche Parameter verwenden, um Elemente anzupassen. Die Dokumentation stellt diese und weitere Funktionalitäten von SCSS vor.

4.1.5 RxJS

[Rea21] RxJS ist eine Bibliothek, mit der sich Ketten von asynchronen Operationen auf Events zusammenstellen lassen. Weil sich Events „beobachten“ lassen, werden ihre Container *Observables* genannt. RxJS ist standardmäßig in Angular integriert und wird an vielen Stellen des Frameworks verwendet. So resultieren HTTP-Anfragen des integrierten HTTP-Clients beispielsweise in einem Observable, welches „beobachtet“ werden muss, um Fehler oder Antworten zu erhalten. Observables können auch direkt in einer Benutzeroberfläche verwendet werden. Detaillierte Beschreibungen der Funktionalitäten sind auf der offiziellen Website zu finden.

4.1.6 NgRx

Die Verwaltung des Zustandes einer Webanwendung kann aufwendig werden, sobald Komplexität und Umfang zunehmen. Mit NgRx kann ein globaler Zustand definiert werden, welcher sich ausschließlich über programmatisch angelegte Aktionen verändern lässt [Rob+20]. Dazu werden Zustände, Aktionen und sogenannten *Reducer*, welche bei eingehenden Aktionen Zustände verändern, definiert. NgRx verwendet RxJS, sodass Zustände „beobachtbar“ sind und auf Zustandsänderungen reagiert werden kann.

Darüber hinaus existieren *Meta-reducers*, die beispielsweise Zustände mit dem *LocalStorage* synchronisieren können und so eine Persistenz von Daten ermöglichen. Mithilfe von NgRx und dem erwähnten Meta-reducer werden Einstellungen wie Sprache und Design, aber auch erstellte Graphen lokal gespeichert.

4.1.7 ngx-translate

Die Webanwendung verwendet keine fest hinterlegten Texte, sondern Übersetzungsschlüssel. Über sie werden, in Abhängigkeit der Spracheinstellung, deutsche oder englische Texte geladen. Verwendet wird dafür die Bibliothek ngx-translate, welche von Combe entwickelt wird [Com18]. Sie unterstützt das Einlesen von Übersetzungsdateien im JSON Format. Anschließend können Übersetzungen über ihre zugehörigen Schlüssel entweder als Zeichenketten oder Observable geladen werden. Letztere aktualisieren sich bei Sprachwechseln automatisch, sodass eine Implementierung von Spracheinstellungen trivialisiert wird.

4.1.8 D3

[Bos20] D3 ist eine JavaScript Bibliothek, welche über optionale Typdefinitionen für eine Verwendung mit TypeScript verfügt. Ihr Name repräsentiert ihre Funktionalität und steht für *Data Driven Documents*. Basierend auf Daten beziehungsweise Datenstrukturen lassen sich HTML- sowie SVG-Elemente erstellen, modifizieren und entfernen. Aufgrund des großen Umfangs von D3 wird nur ein kleiner Teil der Funktionalitäten betrachtet. Zusätzliche Information können der Dokumentation entnommen werden.

Mit D3 können Elemente über ihre HTML-Tags und -Hierarchie sowie CSS-Klassen und -IDs ausgewählt werden. Auf diesen Selektionen sind verschiedene Operationen durchführbar. Beispielsweise können Attribute modifiziert und CSS Klassen hinzugefügt oder entfernt werden. Interessant ist dabei die Möglichkeit, Elemente mit Daten zu verknüpfen. So lassen sich Operationen auf Elementen in Abhängigkeit ihrer zugewiesenen Daten ausführen.

4.2 Implementierung interaktiver Graphen mit D3

Die Implementierung interaktiver Graphen nahm einen Großteil der Entwicklungszeit ein. Die folgenden Unterabschnitte stellen Konzept und Eigenschaften der Implementierung sowie während der Entwicklung entdeckte Probleme vor.

4.2.1 Konzept

In D3 werden gerichtete Graphen über *Force Directed Graphs* implementiert. Bei diesen können optional Kräfte, wie eine Abstoßung zwischen Knoten, konfiguriert werden. D3 berechnet so Knotenpositionen und aktualisiert nach jedem Berechnungsschritt ein zugrunde liegendes Datenmodell. Verwaltung und Aktualisierung von Benutzeroberflächen werden jedoch nicht von D3 übernommen. Infolgedessen müssen diese nach Änderungen am Da-

tenmodell programmatisch durchgeführt werden. Dazu wird der in Unterabschnitt 4.1.8 beschriebene Selektionsmechanismus verwendet, um jeweils alle Knoten beziehungsweise Kanten auszuwählen. Anschließend lässt sich das **transform**-Attribut der SVG-Elemente in Abhängigkeit eines Datenmodells anpassen.

Für Knoten, welche aus Gruppen von SVG-Textelemente sowie einem SVG-Kreis bestehen, ist dies trivial. SVG-Textelemente werden zur Darstellung von Knotennamen sowie Relations- und Konstantensymbolen verwendet. Die verwendete Selektion von Knoten verbindet die beschriebenen Gruppen mit jeweils einem Knoten und ist in Zeile 1 von Quelltext 4.2 zu sehen. Zur Positionierung genügt es, die Koordinaten eines Knotens aus dem Datenmodell an die CSS-Funktion **translate()** zu übergeben, wie in Zeile 4 zu sehen ist.

```

1 private node?: d3.Selection<SVGGElement, D3Node, SVGGElement, unknown>;
2
3 private tick(): void {
4   this.node!.attr('transform', (d) => `translate(${d.x},${d.y})`);
5 }

```

Quelltext 4.2: Positionierung von Knoten

4.2.2 Darstellung gerichteter Kanten

Die Darstellung gerichteter Kanten ist, verglichen mit Knoten, deutlich aufwendiger. Im Kontrast zur Positionierung von Knoten kann nicht mit dem **transform**-Attribut gearbeitet werden. Stattdessen ist eine Verwendung von SVG-Pfaden erforderlich. Dabei müssen Pfade programmatisch auf Basis von Start- und Endknoten berechnet werden. Zusätzlich muss zwischen verschiedenen Darstellungsformen von Kanten unterschieden werden. Abbildung 4.1 zeigt sowohl Kanten mit Rückrichtung (links), Kanten ohne Rückrichtung (mittig), als auch reflexive Kanten (rechts).

In Zeile 1 von Quelltext 4.3 wird die für Kanten verwendete Selektion definiert. Analog zu Knoten verbindet diese eine SVG-Gruppe mit einer Kante (Kanten werden bei D3 als *Links* bezeichnet). Gruppen enthalten neben einem SVG-Pfad noch ein SVG-Textelement, welches zugehörige Relations- und Funktionssymbole enthält. Die in den Zeilen 8 und 10 aufgerufenen Funktionen wurden mithilfe der Bibliothek *ml-matrix* für Matrixrechnungen implementiert [Zas21]. Zunächst berechnen diese einen normierten Differenzvektor zwischen Start- und Endknoten. Existiert eine Kante in entgegengesetzter Richtung wird ein bogenförmiger Pfad durch Rotation des normierten Vektors berechnet. Ist dies nicht der

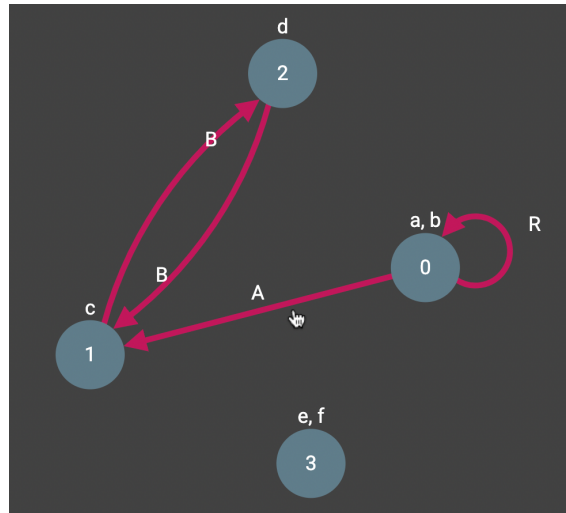


Abbildung 4.1: Graph mit verschiedenen Kanten

Fall, wird der normierte Vektor verlängert, um eine Verbindung zwischen Knoten zu erzeugen.

```

1 private link?: d3.Selection<SVGGElement, D3Link, SVGGElement, unknown>;
2
3 private tick(): void {
4   this.link!.selectAll<SVGPathElement, D3Link>('path').attr('d', (d: D3Link)
      => {
5     if (d.source.id === d.target.id) {
6       return paddedReflexivePath(d.source, [this.width / 2, this.height / 2],
          this.config);
7     } else if (this.isBidirectional(d.source, d.target)) {
8       return paddedArcPath(d.source, d.target, this.config);
9     } else {
10      return paddedLinePath(d.source, d.target, this.config);
11    }
12  });
13 }
```

Quelltext 4.3: Positionierung von Kanten

Ein Sonderfall sind reflexive Kanten, weshalb die Funktion zur Berechnung ihrer Pfade, zu sehen in Quelltext 4.4, detaillierter vorgestellt wird. Anders als bei der zugrunde liegenden Anwendung lassen sich Kanten nicht durch Benutzereingabe positionieren. Um Überschneidungen von reflexiven Kanten mit Knoten beziehungsweise anderen Kanten möglichst zu reduzieren, wurden diese so implementiert, dass sie immer nach Außen gerichtet sind. Dafür muss zunächst ein normierter Differenzvektor zwischen Graphmitte und Knoten berechnet werden, wie in den Zeilen 2 bis 8 zu sehen ist. Anschließend können Start- und Endpunkt eines Pfades durch Rotation des normierten Vektors und anschließender Verlängerung bis

zum Knotenrand berechnet werden. Für den Startpunkt findet dies in den Zeilen 9 bis 12 statt. Bei Endpunkten muss der Platzbedarf von Pfeilmarkierungen berücksichtigt werden, sodass ihre Berechnung eine zusätzliche Verlängerung ihrer Vektoren enthält. Dies ist in den Zeilen 13 bis 16 zu sehen. Abschließend wird in den Zeilen 17 bis 19 ein bogenförmiger Pfad zwischen berechneten Start- und Endpunkten erstellt.

```

1  export function paddedReflexivePath(node: D3Node, center: [number, number],
    graphConfiguration: GraphConfiguration): string {
2    const n = new Matrix([[node.x!, node.y!]]);
3    const c = new Matrix([center]);
4    if (n.get(0, 0) === c.get(0, 0) && n.get(0, 1) === c.get(0, 1)) {
5      c.add([[0, 1]]);
6    }
7    const diff = Matrix.subtract(n, c);
8    const norm = diff.divide(diff.norm('frobenius'));
9    const rotation = degreesToRadians(40);
10   const start = rotate(norm, rotation)
11     .multiply(graphConfiguration.nodeRadius - 1)
12     .add(n);
13   const end = rotate(norm, -rotation)
14     .multiply(graphConfiguration.nodeRadius)
15     .add(n)
16     .add(rotate(norm, -rotation).multiply(2 *
        graphConfiguration.markerBoxSize));
17   return `M${start.get(0, 0)},${start.get(0, 1)}
18     A${graphConfiguration.nodeRadius},${graphConfiguration.nodeRadius},
19     0,1,0,${end.get(0, 0)},${end.get(0, 1)}';
20 }

```

Quelltext 4.4: Positionierung reflexiver Kanten

Während der Entwicklung zeigte sich, dass das Interagieren mit Kanten aufgrund ihrer geringen Breite schwerfallen kann. Als Lösung wurden zusätzliche, transparente Kanten implementiert. Für jede sichtbare Kante wird deshalb eine breitere, unsichtbare Kante mit äquivalentem Pfad zum Abfangen von Nutzereingaben erzeugt. Verdeutlicht werden diese unsichtbaren Kanten durch Verwendung eines Pointer-Cursors, sobald dieser in Reichweite einer Kante ist. Das Resultat dieses Ansatzes ist in Abbildung 4.2 grün hervorgehoben.

4.2.3 Erzeugen von Knoten und Kanten

Zur Erstellung beliebiger Graphen müssen Knoten und Kanten durch Nutzereingaben erzeugbar sein. Bei Knoten kann dies über Doppelklicks sowie den Plus-Button in der rechten unteren Ecke eines Graphs erfolgen. Namen der erzeugten Knoten werden dabei automa-

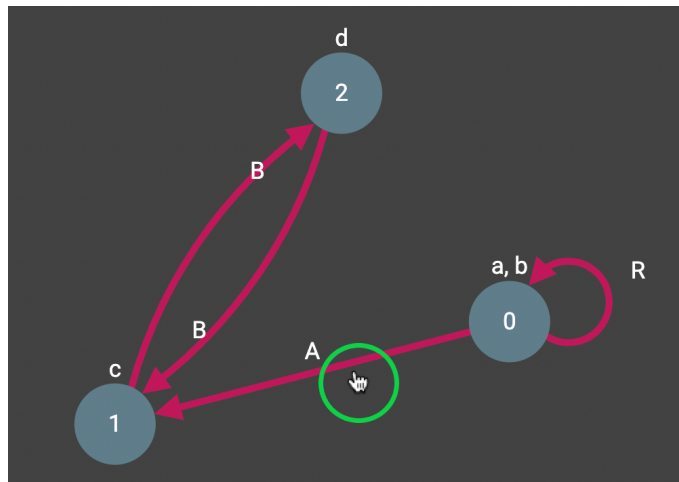


Abbildung 4.2: Pointer-Cursor für Interaktionen mit Kanten

tisch generiert und entsprechen einer aufsteigenden Nummerierung, beginnend bei 0. Das Datenmodell unterstützt jedoch beliebige Zeichenketten für Knotennamen. Eine manuelle Benennung von Knoten ist für den Algorithmus zur Modellprüfung nicht erforderlich, da Knotennamen nur zur Identifikation dienen und nicht in Formeln verwendet werden können.

Kanten lassen sich durch Ziehen von Linien zwischen Knoten erzeugen. Dabei werden gestrichelte Vorschaukanten angezeigt, wie in Abbildung 4.3a zu sehen ist. Sobald ein Knoten erreicht wird, passen sich Vorschaukanten an und werden als bogenförmige oder gegebenenfalls reflexive Kanten dargestellt. Dieser Zustand ist in Abbildung 4.3b abgebildet. Bei „Loslassen“ einer Vorschaukante in diesem Zustand wird eine Kante zwischen gewählten Start- und Endknoten erzeugt. Bei Abbruch eines Erzeugungsvorgangs werden Vorschaukanten entfernt.

Während der Entwicklung zeigte sich schnell, dass beim Erzeugen von Kanten mit Toucheingaben die Sicht durch Finger blockiert werden kann. Vorschaukanten verwenden bei Toucheingaben deshalb einen negativen vertikalen Versatz und haben ihren Endpunkt über eigentlichen Berührungspunkten. So sind Endpunkte von Vorschaukanten jederzeit sichtbar und eine einfache Bedienung wird auch auf mobilen Endgeräten ermöglicht.

4.2.4 Knotenanordnung und Navigation

Knoten lassen sich bei gedrückter mittlerer Maustaste bewegen. So ist es möglich, gewünschte Knotenanordnungen zu erreichen. Nachdem ein Knoten auf diese Art bewegt wurde, wird er nicht mehr von der zugrunde liegenden D3-Simulation kontrolliert. Stattdessen lässt er sich nur noch manuell positionieren. Dies lässt sich auch global für alle Knoten

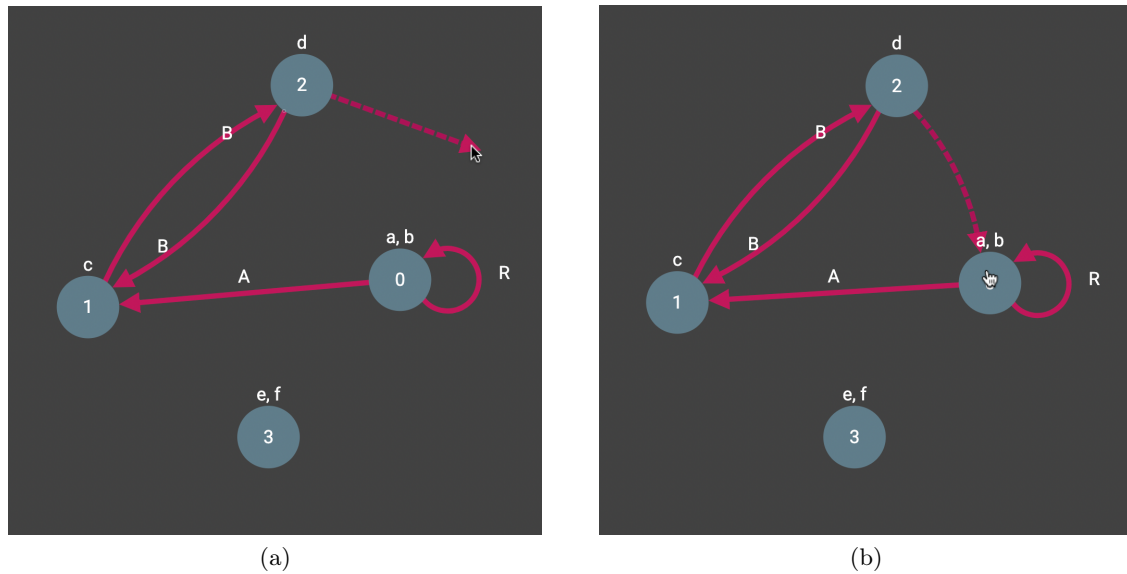


Abbildung 4.3: Vorschau bei Kantenerzeugung

eines Graphs über die Option „Automatische Anordnung“ ein- und ausschalten. Zusätzlich existiert eine Funktion zum Zurücksetzen von Graphen, welche die D3-Simulation für alle Knoten aktiviert.

Darüber hinaus lässt sich die gesamte Graphansicht bei gedrückter linker Maustaste oder Toucheingabe mit mindestens zwei Berührungspunkten verschieben. So lassen sich auch große Graphen bei begrenzter Fenstergröße navigieren. Beim Zurücksetzen einer Graphansicht wird auch die zugehörige Navigation auf ihren Ursprungswert zurückgesetzt, sodass aus dem sichtbaren Bereich verschobene Graphen gefunden werden können.

4.2.5 Touch-Steuerung

Kompatibilität mit Touch-Steuerung stellte sich als besonderes Problem heraus. Um Endgeräte, welche sowohl über Touch- als auch Mauseingaben verfügen, zu unterstützen, ist eine Verwendung von `PointerEvents` erforderlich. Diese abstrahieren Maus- und Touchereignisse, sodass diese einheitlich behandelt werden können.

Während sie sich bei Mauseingaben in den Browsern Google Chrome, Mozilla Firefox und Microsoft Edge einheitlich verhalten, treten bei Toucheingaben jedoch Probleme auf. Die Events `pointerenter` und `pointerout` registrieren beispielsweise Toucheingaben nicht korrekt. Zudem wird das Event `pointerup` in Google Chrome nicht vom Zielelement einer Toucheingabe, sondern dem Ursprungselement der Eingabe ausgelöst. Infolgedessen wurde die Erzeugung von Kanten in Maus- und Toucheingaben separiert. Bei Mauseingaben werden die Events `mouseenter` und `mouseleave` verwendet um Endknoten von Vorschau-

kanten zu erkennen. Weil dies bei Toucheingaben nicht möglich ist, wird stattdessen nach Knoten in Reichweite gesucht. Dazu wird der euklidische Abstand zwischen Knoten und Koordinaten von Toucheingaben berechnet. Sollte dieser kleiner als der Radius eines Knotens sein, wurde der Endknoten der Vorschaukante gefunden. Weil dieser Ansatz bei hoher Knotenanzahl rechenaufwendig ist, wird er ausschließlich bei Toucheingaben verwendet.

Aufgrund der limitierten Eingabemöglichkeit bei Touch-Steuerung konnten die in Unterabschnitt 4.2.4 beschriebene Funktion zur Knotenanordnung nicht für Toucheingaben umgesetzt werden. Anders als bei der Navigation kann dabei nicht auf die Anzahl an Berührungspunkten zurückgegriffen werden, um Eingaben zu filtern. Die zugrunde liegende D3-Funktionalität zum Anordnen von Knoten unterstützt eine lediglich reduzierte Eingabeerkennung, welche bereits zum Erzeugen von Kanten verwendet wird. Eine Einschränkung der Verwendbarkeit liegt jedoch nicht vor, da die optionale automatische Knotenanordnung auf allen Endgeräten aktivierbar ist.

4.3 Funktionalitäten

Dieser Abschnitt stellt die Funktionen der Webanwendung vor. Zunächst wird die jederzeit zugängliche Seitenleiste erläutert. Anschließend werden die Hauptseite der Anwendung, welche zur Verwaltung von Graphen dient, sowie die Seite für Modellprüfungen detailliert. Zuletzt wird die Darstellung von Feedback des Algorithmus für Modellprüfungen vorgestellt.

4.3.1 Seitenleiste

Die Seitenleiste ist auf allen Seiten der Webanwendung vorhanden. Nahaufnahmen sind in Abbildung 4.4 zu sehen. Sie lässt sich über den Button in der linken oberen Ecke der Webanwendung aus- und einklappen. Über die Navigationsbuttons „Home“ und „Model-Checker“ lässt sich zu den zugehörigen Seiten, welche in den folgenden Unterabschnitten beschrieben werden, navigieren. Darunter befindet sich eine Sprachauswahl, bei welcher die ausgewählte Sprache dunkel hinterlegt wird. Das Design der Webanwendung lässt sich über den Button mit Sonnen- beziehungsweise Mond-Symbol umschalten. Ein Beispiel des hellen Designs ist in Abbildung 4.4b abgebildet. Der letzte Eintrag der Seitenleiste ist ein Link zum GitHub Repository der Webanwendung.

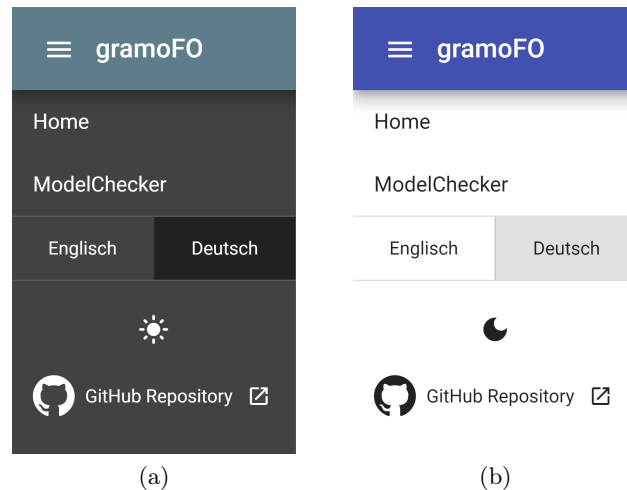


Abbildung 4.4: Seitenleiste der Webanwendung

4.3.2 Seite zur Verwaltung von Graphen

Die Hauptseite der Webanwendung enthält neben einer Tabelle lokal gespeicherter Graphen noch verschiedene Optionen zum Importieren von Graphen. In Abbildung 4.5 ist genannte Tabelle zu sehen. Einträge lassen sich nach Name, Beschreibung oder Zeitstempel der letzten Änderung sortieren. Graphen lassen sich zudem jeweils öffnen, wodurch eine Weiterleitung zur Seite für Modellprüfungen stattfindet, exportieren und löschen. Bei Letzterem informiert eine Nachricht über den Vorgang und bietet ein Rückgängigmachen von Löschungen an.

Name	Beschreibung	Letzte Änderung	Aktionen
Mein Graph	Dies ist ein Beispiel.	22.01.2021 04:16	Öffnen Exportieren Löschen
Demo Graph	A simple demonstration Graph.	22.01.2021 06:14	Öffnen Exportieren Löschen

Abbildung 4.5: Tabelle mit gespeicherten Graphen

Abbildung 4.6 zeigt die Optionen eines Exportvorganges. Graphen können im JSON und YAML Format exportiert werden. Beispiele für beide Formate sind in Abschnitt B.1 zu finden. Bei beiden Formaten bestehen die Optionen, Textrepräsentationen in die Zwischenablage zu kopieren oder Dateien herunterzuladen. Für Letzteres werden unsichtbare Links generiert, für ausgewählte Formate konfiguriert und durch virtuelle Klicks auf selbige Downloads ausgelöst.

Ein Import dieser Dateien und Textrepräsentationen kann über die in Abbildung 4.7 abgebildeten Bedienelemente stattfinden. Im Textfeld können Textrepräsentationen eingefügt

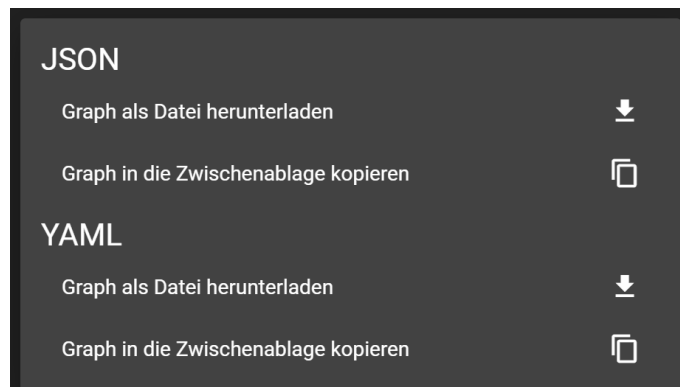


Abbildung 4.6: Exportieren eines Graphs

werden. Der Button „Datei importieren“ öffnet einen Dialog zur Dateiauswahl. Bei beiden Optionen werden importierte Graphen zunächst auf Gültigkeit geprüft. Treten dabei Fehler auf, wird eine entsprechende Nachricht angezeigt. Ansonsten werden importierte Graphen unmittelbar in der Seite für Modellprüfungen geöffnet. Zuletzt existiert die Option, einen vordefinierten Graphen zu öffnen, welcher für Demonstrationszwecke hinterlegt wurde.

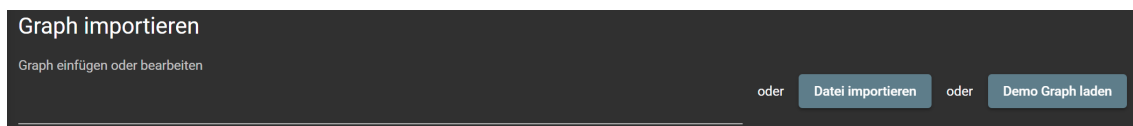


Abbildung 4.7: Importfunktionen

4.3.3 Seite für Modellprüfungen

Modellprüfungen finden über eine eigene Seite der Webanwendung statt. Dort sind über einem Bereich zur Bearbeitung von Graphen noch Formeleingabe und Feedbackauswahl vorhanden. Sowohl Formeleingabe als auch Graph verfügen über Hilfstexte bezüglich Syntax und Steuerung, welche über einen Button mit einem Fragezeichensymbol abrufbar sind. Die Feedbackauswahl verfügt über die in Unterabschnitt 3.3.2 vorgestellten Auswahlmöglichkeiten.

Die Graphansicht bietet eine Option, Beschriftungen von Knoten und Kanten auszublenden. Zudem können Graphen mithilfe des in Abbildung 4.8 abgebildeten Dialogs gespeichert werden. Dazu werden sie mit einem Namen eindeutig identifiziert und können optional mit einer Beschreibung versehen werden. Die Webanwendung speichert sie im LocalStorage des Browsers und macht sie über die Seite zur Verwaltung von Graphen zugänglich.

Knoten und Kanten lassen sich über Rechtsklicks und lange Berührungen bei Touch-Steuerung auswählen. Zudem werden erzeugte Knoten und Kanten automatisch ausgewählt. Über die in Abbildung 4.9 zu sehenden Formulare können ausgewählte Knoten so-

Abbildung 4.8: Dialog zum Speichern von Graphen

wie Kanten bearbeitet und gelöscht werden. Relationen sowie Funktionen beziehungsweise Konstanten lassen sich den jeweiligen Entitäten zuschreiben und von ihnen entfernen.

(a)

(b)

Abbildung 4.9: Formulare zum Bearbeiten von Knoten und Kanten

Sobald eine Formel eingegeben wurde, kann über den „Überprüfen“-Button eine Anfrage an das Backend gesendet werden. Wird dieser betätigt, öffnet sich eine Fortschrittsanzeige, welche in Abbildung 4.10 zu sehen ist. Diese deutet zunächst die Durchführung einer Modellprüfung und anschließend das Empfangen einer Antwort an. Bei Letzterem werden totale Antwortgröße und Größe bereits empfangener Pakete verwendet, um einen prozentualen Fortschritt anzuzeigen. Bei Bedarf können Anfragen über den roten Button am rechten Rand der Anzeige abgebrochen werden.

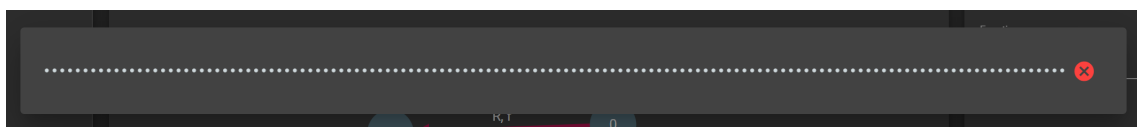


Abbildung 4.10: Fortschrittsanzeige bei Modellprüfungen

4.3.4 Darstellung von generiertem Feedback

Das in Unterabschnitt 3.3.1 detaillierte Feedback des Algorithmus zur Modellprüfung wird in einer Baumstruktur dargestellt. Dazu wird ein Dialog angezeigt, welcher sich im gesamten Fensterbereich verschieben lässt. Er wird beim Verlassen der Seite für Modellprüfungen, erneuter Überprüfung oder Betätigung des roten Buttons in der rechten oberen Ecke geschlossen.

In Abbildung 4.11 ist der Feedback-Baum einer „vollständigen“ Modellprüfung zu sehen. Zeilen repräsentieren jeweils einen Trace des Algorithmus. Links werden die Formeln der Traces angezeigt, während am rechten Rand Erklärungen der Ergebnisse vorhanden sind. Knoten des Feedback-Baumes, welche über Kinder verfügen, lassen sich erweitern und einklappen. Über die Buttons in der linken unteren Ecke können alle Knoten gleichzeitig erweitert und eingeklappt werden. Exklusiv für „vollständiges“ Feedback existiert die Option nur „relevante“ Traces anzuzeigen. Bei Aktivierung entspricht die Anzeige beinahe der einer Durchführung des Algorithmus mit „relevantem“ Feedback. Lediglich bei Quantoren können weitere Traces angezeigt werden, da kein frühzeitiger Abbruch der Evaluation erfolgt. Rot markiert werden alle Traces, bei denen sich erwartetes und tatsächliches Ergebnis unterscheiden. So lässt sich beispielsweise schnell erkennen, dass die Formel $R(0,0) \wedge A(0,0)$ nicht gilt, da ihr zweites Konjunkt $A(0,0)$ nicht gilt. Daran ist insbesondere die Ersetzung von gebundenen Variablen durch ihre Belegungen beteiligt.

Die Möglichkeit zum Verschieben des Feedback-Dialogs wurde implementiert, um eine gleichzeitige Betrachtung von Graph, Formel und Feedback zu ermöglichen. Diese Situation ist in Abbildung 4.12 mit „relevantem“ Feedback abgebildet. Graphen lassen sich uneingeschränkt bedienen, während ein Feedback-Dialog geöffnet ist, sodass Feedback zur nachträglichen Bearbeitung von Graphen gezielt verwendet werden kann.

Sollten Ursachen von Modellprüfungen nicht von Interesse sein, ist „minimales“ Feedback eine bandbreitensparende Option. Weil lediglich Endergebnisse und keine Zwischenschritte übermittelt werden, fallen auch zugehörige Dialoge klein aus, wie in Abbildung 4.13 zu sehen ist. Neben der Größe von Antworten ist auch der Rechenaufwand zur Darstellung von Feedback-Bäumen zu berücksichtigen. Insbesondere bei älteren oder mobilen Endgeräten in Kombination mit „vollständigem“ Feedback kann eine Darstellung großer Bäume zu Problemen wie einem Einfrieren der Webanwendung führen.

✓ Der Graph ist ein Modell der Formel $\exists x. \exists y. R(x, x) \wedge A(x, y)$

- ✓ $\exists x. \exists y. R(x, x) \wedge A(x, y)$
 - Unterformel gilt für mindestens eine Variablenzuweisung
 - ✓ $\exists y. R(0, 0) \wedge A(0, y)$
 - Unterformel gilt für mindestens eine Variablenzuweisung
 - ✓ $R(0, 0) \wedge A(0, 0)$
 - Zweites Konjunkt gilt nicht
 - $R(0, 0)$ $(0, 0) \in R$
 - $A(0, 0)$ $(0, 0) \notin A$
 - ✓ $R(0, 0) \wedge A(0, 1)$
 - Beide Konjunkte gelten
 - $R(0, 0)$ $(0, 0) \in R$
 - $A(0, 1)$ $(0, 1) \in A$
 - > $R(0, 0) \wedge A(0, 2)$
 - Zweites Konjunkt gilt nicht
 - > $R(0, 0) \wedge A(0, 3)$
 - Zweites Konjunkt gilt nicht
 - > $\exists y. R(1, 1) \wedge A(1, y)$
 - Unterformel gilt für keine Variablenzuweisungen
 - > $\exists y. R(2, 2) \wedge A(2, y)$
 - Unterformel gilt für keine Variablenzuweisungen

Erweitern Einklappen Relevantes filtern

Abbildung 4.11: „Vollständiges“ Feedback einer Modellprüfung

Formel
exists x. exists y. R(x,x) && A(x,y)

Automatische Anordnung

Der Graph ist ein Modell der Formel $\exists x. \exists y. R(x, x) \wedge A(x, y)$

- ✓ $\exists x. \exists y. R(x, x) \wedge A(x, y)$
 - Unterformel gilt für mindestens eine Variablenzuweisung
 - ✓ $\exists y. R(0, 0) \wedge A(0, y)$
 - Unterformel gilt für mindestens eine Variablenzuweisung
 - ✓ $R(0, 0) \wedge A(0, 1)$
 - Beide Konjunkte gelten
 - $R(0, 0)$ $(0, 0) \in R$
 - $A(0, 1)$ $(0, 1) \in A$

Erweitern Einklappen

Abbildung 4.12: „Relevantes“ Feedback einer Modellprüfung

✗ Der Graph ist kein Modell der Formel $\neg(\exists x. \exists y. R(x, x) \wedge A(x, y))$

Abbildung 4.13: „Minimales“ Feedback einer Modellprüfung

5 Fazit und Ausblick

Dieses Kapitel zieht zunächst ein Fazit zur Umsetzung der gesetzten Ziele. Anschließend werden im Rahmen eines Ausblicks mögliche Erweiterungen der Webanwendung vorgestellt.

5.1 Fazit

Alle in Abschnitt 1.2 definierten Ziele konnten umgesetzt werden. Der zugrunde liegende Quelltext konnte in eine Serveranwendung integriert werden. Durch Änderungen und Erweiterungen war es möglich, Ergebnisse des Algorithmus für Modellprüfungen um detailliertes und strukturiertes Feedback zu ergänzen. Darüber hinaus konnten drei verschiedene Varianten des Algorithmus implementiert werden, welche unterschiedliche Detailgrade für ihr Feedback verwenden.

Bei der Implementierung interaktiver Graphen mit D3 traten zunächst verschiedene Probleme auf. Toucheingaben führten bei Verwendung in verschiedenen Browsern zu Fehlern und unerwartetem Verhalten. Durch gezielte Anpassungen der Implementierungen konnten diese Probleme jedoch behoben werden.

Eine Visualisierung von Ergebnissen und Schritten des Algorithmus für Modellprüfungen wurde durch eine interaktive Benutzeroberfläche in Baumform realisiert. Wie die in Unterabschnitt 4.3.4 verwendeten Beispiele zeigen, lassen sich Ursachen von Ergebnissen schnell erkennen.

5.2 Ausblick

Die folgenden Unterabschnitte beschreiben mögliche Erweiterungen der Webanwendung. Die Konzepte und Ideen entstanden im Laufe der Entwicklung, wurden jedoch nicht umgesetzt.

5.2.1 Aufgabenstellungen

Um eine Verwendung als pädagogisches Werkzeug zu realisieren, könnten Aufgabenstellungen in die Webanwendung integriert werden. Denkbar sind beispielsweise Aufgaben, bei denen zwei oder mehr Graphstrukturen vorgegeben sind. Ziel könnte dann eine Eingabe von Formeln sein, welche in einer definierten Teilmenge der vorgegebenen Graphstrukturen

gelten, also eine Menge von Graphstrukturen in zwei Gruppen teilen. Ebenfalls vorstellbar wäre die Rückrichtung, also ein Erstellen von Graphstrukturen, in denen angegebene Formeln gelten oder nicht gelten.

Die Architektur der Webanwendung würde ein solches Vorhaben unterstützen. Die Benutzeroberfläche von Graphen wurde so implementiert, dass sich schreibgeschützte Instanzen erzeugen lassen. Infolgedessen sind sie bereits für eine reine Anzeige von Graphen, ohne Möglichkeiten zur Bearbeitung, geeignet. Die Struktur der Webanwendung sieht zudem vor, dass Graphanzeige und Grapheditor als alleinstehende Komponenten einsetzbar sind und sich so in anderen Kontexten wiederverwenden lassen.

5.2.2 Erweiterung und Analyse von Feedback

In der implementierten Webanwendung wird Feedback nur visualisiert. Denkbar wäre eine Analyse von Feedback, um zusätzliche Informationen zu gewinnen und in alternativen Formen darzustellen. Beispielsweise könnte eine Generierung von natürlichsprachigen Texten die Feedback-Bäume begleiten. Denkbar sind Texte wie „Der Allquantor gilt nicht für die Variablenbelegung $x \mapsto 2$ “. Dafür könnte eine Analyse von Kindern beziehungsweise Nachfahren von Traces durchgeführt werden. Optional könnten Traces bereits während einer Durchführung des Algorithmus zur Modellprüfung um zusätzliche Informationen, wie Variablenbelegungen und Typen überprüfter Formeln, angereichert werden.

5.2.3 Analyse der Platzkomplexität

Wie in Unterabschnitt 3.3.3 beschrieben, können bei einer Verwendung des Algorithmus für Modellprüfungen in der „vollständigen“ Variante verschiedene Probleme auftreten. Aktuell wird nur in einem Tooltip bei Auswahl des Feedbacks darauf hingewiesen. Formeln werden in der Webanwendung nicht geparkt, sodass eine vollständige Analyse der Platzkomplexität unter Berücksichtigung ihrer Semantik nicht möglich ist. Lösungsansätze wären entweder die Implementierung eines Parsers im Frontend oder eine heuristische Analyse ihrer ungeparkten Zeichenketten in Kombination mit Graphen. Insbesondere Knotenanzahl und Verschachtelungstiefe von Quantoren wären dabei relevant, wie in Unterabschnitt 3.3.3 beschrieben wird. So könnten bereits vor einer Anfrage an das Backend die Wahrscheinlichkeit eines `OutOfMemoryErrors` näherungsweise abgeschätzt und ein entsprechender Warnhinweis angezeigt werden.

5.2.4 Heuristische Positionierung von Knoten und Kanten

D3 berücksichtigt bei der Positionierung von Knoten nicht, ob sich Kanten überschneiden, worunter die Übersichtlichkeit von Graphen leiden kann. Dafür gibt es zwei Ursachen. Zum einen kann D3 keine Annahmen über die Darstellung von Kanten treffen, da diese implementierungsabhängig sind, wie in Unterabschnitt 4.2.2 beschrieben wurde. Zum anderen ist nicht jeder Graph planar. Bereits der vollständige bipartite Graph mit je drei Knoten pro Knotenteilmenge ist nicht planar [Vol15], lässt sich jedoch in der Webanwendung erstellen. Es gilt außerdem, dass bereits eine Berechnung der Anzahl minimaler Kantenüberschneidungen eines Graphs NP-vollständig ist [GJ83]. Das Problem, eine möglichst überschneidungsfreie Anordnung zu finden, ist demzufolge nicht trivial.

Denkbar sind verschiedene Lösungsansätze. Einerseits könnte die Webanwendung so angepasst werden, dass sich Kanten manuell positionieren und ausrichten lassen. Nachteilig ist dabei jedoch, dass bei Graphen mit vielen Kanten ein nicht trivialer Aufwand entsteht und somit eine Verwendung nicht unmittelbar erleichtert wird. Ein gänzlich anderer Ansatz wäre ein Einsatz von Algorithmen zur Optimierung von Knoten- und Kantenanordnungen. Utech u. a. schlagen dafür beispielsweise evolutionäre Algorithmen zur Minimierung von Kantenüberschneidungen vor [Ute+98].

A Literatur

- [BAT14] Gavin Bierman, Martín Abadi und Mads Torgersen. „Understanding TypeScript“. In: *ECOOP 2014 – Object-Oriented Programming*. Hrsg. von Richard Jones. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, S. 257–281.
- [Bos20] Mike Bostock. *D3.js - Data-Driven Documents*. 2020. URL: <https://d3js.org/> (besucht am 26.01.2021).
- [Com18] Olivier Combe. *NGX-Translate: The i18n library for Angular 2+*. 2018. URL: <http://www.ngx-translate.com/> (besucht am 26.01.2021).
- [Ehl15] Arno Ehle. *Modellchecker – Projektarbeit*. 2015.
- [Fla+18] Matheus Flauzino, Júlio Veríssimo, Ricardo Terra, Elder Cirilo, Vinicius Durelli und Rafael Durelli. „Are you still smelling it?: A comparative study between Java and Kotlin language“. In: Sep. 2018, S. 23–32. DOI: 10.1145/3267183.3267186.
- [GJ83] M. R. Garey und D. S. Johnson. „Crossing Number is NP-Complete“. In: *SIAM Journal on Algebraic Discrete Methods* 4.3 (1983), S. 312–316. DOI: 10.1137/0604033. eprint: <https://doi.org/10.1137/0604033>. URL: <https://doi.org/10.1137/0604033>.
- [Goo20a] Google. *Angular Material UI component library*. 2020. URL: <https://material.angular.io/> (besucht am 26.01.2021).
- [Goo20b] Google. *The modern web developer’s platform*. 2020. URL: <https://angular.io/> (besucht am 26.01.2021).
- [Hru16] Benedikt Hruschka. *Modellchecker mit Beweismodus – Projektarbeit*. 2016.
- [Hru19] Benedikt Hruschka. *Implementierung eines Interaktiven Model-Checkers für erststufige Logik über automatischen Strukturen – Masterarbeit*. 2019. URL: http://www.uni-kassel.de/eecs/fileadmin/datas/fb16/Fachgebiete/FMV/thesis_bhruschka_31251019.pdf (besucht am 23.01.2021).
- [Jet20a] JetBrains. *Ktor: Build Asynchronous Servers and Clients in Kotlin*. 2020. URL: <https://ktor.io/> (besucht am 26.01.2021).
- [Jet20b] JetBrains. *Reference - Kotlin Programming Language*. 2020. URL: <https://kotlinlang.org/docs/reference/> (besucht am 26.01.2021).

- [Jet21] JetBrains. *UML class diagrams—IntelliJ IDEA*. 2021. URL: <https://www.jetbrains.com/help/idea/class-diagram.html> (besucht am 06.02.2021).
- [Koi21] Koin. *Koin - The Kotlin Injection Framework / Koin*. 2021. URL: <https://insert-koin.io/> (besucht am 27.01.2021).
- [Lan17] Martin Lange. *Theoretische Informatik: Logik (2017/18) – Unveröffentlichtes Vorlesungsskript, Universität Kassel*. 2017.
- [Mic21] Microsoft. *Typed JavaScript at Any Scale*. 2021. URL: <https://www.typescriptlang.org/> (besucht am 26.01.2021).
- [Oraa] Oracle Corporation. *JDK 11 Release Notes*. URL: <https://www.oracle.com/java/technologies/javase/jdk-11-relnote.html> (besucht am 23.01.2021).
- [Orab] Oracle Corporation. *Oracle Java SE 8 Release Updates*. URL: https://java.com/en/download/release_notice.jsp (besucht am 23.01.2021).
- [PS14] Candy Pang und Duane Szafron. „Single Source of Truth (SSOT) for Service Oriented Architecture (SOA)“. In: *Service-Oriented Computing*. Hrsg. von Xavier Franch, Aditya K. Ghose, Grace A. Lewis und Sami Bhiri. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, S. 575–589.
- [Rea21] ReactiveX. *RxJS - Introduction*. 2021. URL: <https://rxjs-dev.firebaseapp.com/guide/overview> (besucht am 26.01.2021).
- [Rob+20] Brandon Roberts, Mike Ryan, Victor Savkin und Rob Wormald. *NgRx - @ngrx/store*. 2020. URL: <https://ngrx.io/guide/store> (besucht am 26.01.2021).
- [Sas21] Sass. *Sass: Syntactically Awesome Style Sheets*. 2021. URL: <https://sass-lang.com/> (besucht am 26.01.2021).
- [SO18] Mohamed Sultan und Opencast. „Angular and the Trending Frameworks of Mobile and Web-based Platform Technologies: A Comparative Analysis“. In: 2018.
- [Ute+98] J. Utech, J. Branke, H. Schmeck und P. Eades. „An Evolutionary Algorithm for Drawing Directed Graphs“. In: 1998.
- [Vol15] Andrea Vollmer. *Planare Graphen*. 2015. URL: <https://ivv5hpp.uni-muenster.de/u/timmermt/Lehre/15/S-GT/9-vollmer.pdf> (besucht am 25.01.2021).

- [Zas21] Michaël Zasso. *mljs/matrix: Matrix manipulation and computation library*. 2021. URL: <https://github.com/mljs/matrix> (besucht am 27.01.2021).

B Anhang

B.1 Beispiele von Graphexporten

B.1.1 JSON Format

```
1 {
2   "name": "GraphName",
3   "description": "Dies ist ein Beispielgraph.",
4   "nodes": [
5     {
6       "name": "0",
7       "relations": [],
8       "constants": [
9         "c"
10      ]
11    },
12    {
13      "name": "1",
14      "relations": [],
15      "constants": []
16    }
17  ],
18  "edges": [
19    {
20      "source": "1",
21      "target": "0",
22      "relations": [
23        "R"
24      ],
25      "functions": []
26    }
27  ]
28 }
```

Quelltext B.1: JSON Format eines Beispielgraphs

B.1.2 YAML Format

```
1 name: GraphName
2 description: "Dies ist ein Beispielgraph."
3 nodes:
4   - name: "0"
5     relations: []
6     constants:
7       - c
8   - name: "1"
9     relations: []
10    constants: []
11 edges:
12   - source: "1"
13     target: "0"
14     relations:
15       - R
16   functions: []
```

Quelltext B.2: YAML Format eines Beispielgraphs