UNIKASSEL
VERSITÄT

Fachbereich Theoretische Informatik/
Formale Methoden

Implementierung eines Interaktiven Model-Checkers für erststufige Logik über automatischen Strukturen

MASTERARBEIT

zur Erlangung des Grades eines Master of Science
im Fachbereich Theoretische Informatik/Formale Methoden
der Universität Kassel

Eingereicht von:	Benedikt Hruschka
Anschrift:	Alter Teichweg 84 22081 Hamburg
Matrikelnummer:	31251019
Emailadresse:	BenediktHr@icloud.com
Vorgelegt im:	Fachbereich Theoretische Informatik/Formale Methoden
Gutachter:	Prof. Dr. Martin Lange Prof. Dr. rer. nat. Albert Zündorf
Betreuer:	Dr. Florian Bruse Dr. Norbert Hundeshagen
eingereicht am:	12.07.2019

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, 12.07.2019

Benedikt Hruschka

Inhaltsverzeichnis

Abbildungsverzeichnis

Listings

1 Einleitung

1.1 Motivation

Studenten fällt es häufig schwer Konzepte der theoretischen Informatik zu erlernen und anzuwenden. Um diese im Lernprozess zu unterstützen, gibt es hilfreiche Anwendungen, welche bestimmte Sachverhalte erklären oder visuell aufbereiten. Ein solches Tool ist beispielsweise der grafische „Model Checker“, welcher von einem Kommilitonen entwickelt wurde. Ziel dieses Tools ist das Entscheiden, ob eine visuell definierte endliche Struktur Modell einer eingegebenen prädikatenlogischen Formel ist. Dieses Tool wurde im Rahmen eines Projektes meines Masterstudiums um einen „Beweismodus“ ergänzt.

Mithilfe der Erweiterung, welche eine Abwandlung bekannter Spielsemantiken beim Model Checking ist, wurde es Studenten ermöglicht, das Model Checking schrittweise mithilfe des dargestellten Formelbaums durchzuführen und nachzuvollziehen.

Um den Lerneffekt an dieser Stelle noch weiterzuführen, entstand die Idee die Anwendung um mächtigere Strukturen zu erweitern.

1.2 Ziel

Ziel dieser Masterarbeit ist es, die angesprochene Anwendung um unendliche Strukturen zu erweitern. Da es aber nicht möglich ist für alle unendlichen Strukturen automatisiert Model Checking durchzuführen, beschäftigt sich diese Arbeit mit der Sub-Klasse der automatischen Strukturen. Diese Strukturen werden mithilfe von regulären Ausdrücken beschrieben und sind dadurch endlich repräsentierbar. Eine weitere Möglichkeit der endlichen Repräsentation sind endliche Automaten, welche jedoch in der Lage sein müssen mehrere Spuren synchron einzulesen. Um diese Funktionalität nicht von Grund auf neu zu entwickeln, soll hier eine bestehende Bibliothek genutzt, erweitert und integriert werden. Zusätzlich muss die momentane Implementierung weiterhin funktionstüchtig bleiben und für die Erweiterung abstrahiert werden.

1.3 Aufbau der Arbeit

In Kapitel 2 werden zunächst die Grundlagen der Prädikatenlogik erster Stufe definiert sowie ein alternativer Ansatz des Model Checkings, welcher die Basis für die in Kapitel 3 definierten automatischen Strukturen bildet. Weiterhin werden in Kapitel 3 synchrone Mehrspurautomaten definiert und dessen Funktionen vorgestellt, welche für das Model Checking von automatischen Strukturen benötigt werden. Im nachfolgenden Kapitel 4 wird anschließend die Umsetzung und Implementierung der im vorherigen Kapitel beschriebene Definitionen erläutert. Dazu wird der Technologie Stack vorgestellt und die aktuelle Implementierung des Model Checkers sowie Beweismodus beschrieben. Anschließend werden die entwickelten Algorithmen mit Beispielen, Pseudocode und Ablaufdiagrammen erklärt. Das Kapitel endet mit der Integration der erweiterten Bibliothek in die bestehende Implementierung des Model Checkers sowie Beweismodus. Kapitel 5 bildet das Fazit und beschreibt einige mögliche Erweiterungen.

2 Prädikatenlogik erster Stufe

Im folgenden Kapitel wird die Definition der Prädikatenlogik erster Stufe unter relationalen Strukturen betrachtet. Die Grundlage dafür bildet das Vorlesungsskript für „Theoretische Informatik: Logik“ [Lan18].

2.1 Syntax und Semantik

Um eine formale Sprache wie die Prädikatenlogik zu beschreiben, müssen zunächst die Signaturen und Strukturen definiert werden.

Definition 2.1.1. *Eine Signatur τ ist eine Liste $\langle R_1^{(s_1)}, ..., R_n^{(s_n)} \rangle$, wobei*

- *die R_i jeweils Relationssymbole und*
- *$s_i \in \mathbb{N}$ für $i \in \{1, ..., n\}$ die Stelligkeit des jeweiligen Symbols sind.*

Die Stelligkeit kann auch als Funktion $st(R_i) = s_i$ angegeben werden, um auszudrücken, dass R_i mit Stelligkeit s_i in der zugrundeliegenden Signatur vorkommt.
Weiterhin werden Relationen $R^{(0)}$ mit der Stelligkeit 0 als Propositionen bezeichnet.
Da die Signatur ausschließlich die Komponenten der formalen Sprache definiert, wird noch eine Struktur benötigt. Diese stellt zu den Relationssymbolen der Signatur konkrete Relationen bereit.

Definition 2.1.2. *Sei $\tau = \langle R_1^{(s_1)}, ..., R_n^{(s_n)} \rangle$. Eine τ-Struktur ist ein Tupel $\mathcal{A} = (U, R_1^{\mathcal{A}}, ..., R_n^{\mathcal{A}})$, wobei*

- *U eine nicht-leere Menge, genannt Universum von $\mathcal{A}$ ist und*
- *$R_i^{\mathcal{A}} \subseteq U^{s_i}$, für $i \in \{1, ..., n\}$ Relationen sind.*

Damit wird also ein n-stelliges Relationssymbol durch eine n-stellige Relation auf dem Universum U der Struktur interpretiert.

Beispiel 2.1.3. *Eine mögliche τ Struktur für den Vergleich von natürlichen Zahlen über der Signatur $\tau := \langle \leq^{(2)} \rangle$ wäre damit also $\mathcal{A}_1 = (\mathbb{N}, \leq^{\mathcal{A}})$.*

$$\leq^{\mathcal{A}_1} = \{(0,0),(0,1),...,(1,1),(1,2),...\} \subseteq \mathbb{N}^2$$

Um nun Aussagen über τ-Strukturen zu treffen wird eine Sprache zur Formalisierung benötigt. Die Element dieser Sprache werden τ-Formeln genannt. Grundlegende Bestandteile einer Formel für relationale Strukturen sind Variablen. Diese Variablen sind Teil der Variablenmenge $\mathcal{V} = \{x_1, x_2, ...\}$, welche eine geordnete, abzählbar unendlich große Menge ist.

Definition 2.1.4. *Wenn φ und ψ Formeln sind, gilt:*

- $\mathtt{tt}$ *und* $\mathtt{ff}$ *sind Formeln.*
- *Sind $x_1, ..., x_n$ Variablen, so ist $R(x_1, ..., x_n)$ Formel, falls $\tau = \langle ..., R^{(n)}, ... \rangle$.*
- *Sind x und y Variablen über τ und $\mathcal{V}$, so ist $x \doteq y$ eine Formel.*
- *Sind φ, ψ Formeln, so auch $\neg\varphi$, $(\varphi \vee \psi)$, $(\varphi \wedge \psi)$, $(\varphi \rightarrow \psi), (\varphi \leftrightarrow \psi)$.*
- *$\exists x\ \varphi$ und $\forall x\ \varphi$ sind auch Formeln, bei denen jeweils x eine gebundene Variable ist.*

Formeln der Form $\mathtt{tt}$, $\mathtt{ff}$ *und $R(x_1, ..., x_n)$ heißen auch atomar.*

Durch Definition 2.1.4 ist ersichtlich, dass Formeln auch geschachtelt werden können. Um die Menge aller Unterformeln einer Formel zu erhalten, gibt es die Funktion *Sub*.

Definition 2.1.5. *Sei φ eine prädikatenlogische Formel. Dann ist die Menge aller Unterformeln von φ $Sub(\varphi)$ wie folgt induktiv definiert:*

$$\begin{aligned}
Sub(R(x_1, ..., x_n)) &:= \{R(x_1, ..., x_n)\} \\
Sub(x \doteq x') &:= \{x \doteq x'\} \\
Sub(\varphi \wedge \psi) &:= \{\varphi \wedge \psi\} \cup Sub(\varphi) \cup Sub(\psi) \\
Sub(\varphi \vee \psi) &:= \{\varphi \wedge \psi\} \cup Sub(\varphi) \cup Sub(\psi) \\
Sub(\neg\varphi) &:= \{\neg\varphi\} \cup Sub(\varphi) \\
Sub(\exists\varphi) &:= \{\exists\varphi\} \cup Sub(\varphi) \\
Sub(\forall\varphi) &:= \{\forall\varphi\} \cup Sub(\varphi)
\end{aligned}$$

Zusätzlich wird das Vorkommen von Variablen in einer Formel in freie und gebundene Variablen unterschieden. Eine Formel ohne freie Variablen wird als Aussage bzw. geschlossene Formel bezeichnet.

Definition 2.1.6. *Die freien Variablen einer Formel φ, $free(\varphi)$, über einer Variablenmenge $\mathcal{V}$ sind folgendermaßen definiert:*

$$\begin{aligned} free(R(x_1, ..., x_n)) &:= \bigcup_{i=1}^{n} x_i \\ free(x \doteq x') &:= \{x, x'\} \\ free(\neg\varphi) &:= free(\varphi) \\ free(\varphi \wedge \psi) = free(\varphi \vee \psi) = free(\varphi \rightarrow \psi) = free(\varphi \leftrightarrow \psi) &:= free(\varphi) \cup free(\psi) \\ free(\exists x\varphi) = free(\forall x\varphi) &:= free(\varphi) \setminus x \end{aligned}$$

Die gebundenen Variablen einer Formel φ, $bnd(\varphi)$, sind folgendermaßen definiert:

$$bnd(\varphi) = \{x \mid \exists x\psi \ \in Sub(\varphi) \text{ oder } \forall x\psi \ \in Sub(\varphi)\}$$

Es ist jedoch zu beachten, dass gebundene Variablen nicht das Komplement der freien Variablen sind, da Variablen in einer Formel sowohl frei als auch gebunden vorkommen können.

Definition 2.1.7. *Eine τ-Interpretation ist ein $\mathcal{I} = (\mathcal{A}, \vartheta)$, wobei $\mathcal{A}$ eine τ-Struktur mit Universum U und $\vartheta : \mathcal{V} \rightarrow U$ eine Variablenbelegung ist, die den zugrundeliegenden erststufigen Variablen jeweils Elemente im Universum von $\mathcal{A}$ zuweist.*
$[\![x]\!]^{\mathcal{A}}_{\vartheta}$ bezeichnet den Wert der Variable x in der Struktur $\mathcal{A}$ unter der Variablenbelegung ϑ.

$$\begin{aligned} [\![x]\!]^{\mathcal{A}}_{\vartheta} &:= \vartheta(x) && \text{, falls } x \in \mathcal{V} \\ \mathcal{V}[x \mapsto u](y) &:= \begin{cases} u & \text{, falls } y = x \\ \mathcal{V}(y) & \text{, sonst} \end{cases} \end{aligned}$$

Der Wert aller Variablen x in einer Struktur $\mathcal{A}$ wird unter der Variablenbelegung ϑ ermittelt, indem jede Variable x durch ihren Wert unter ϑ ersetzt wird.

Definition 2.1.8. *Eine Interpretation $\mathcal{I} = (\mathcal{A}, \vartheta)$ erfüllt eine Formel φ ($\mathcal{I} \models \varphi$), wenn φ durch die Interpretation als wahr ausgewertet wird. Sei U das Universum von $\mathcal{A}$.*

$\mathcal{I} \models R(x_1, ..., x_n)$	*gdw.*	$([\![x_1]\!]^{\mathcal{A}}_{\vartheta}, ..., [\![x_n]\!]^{\mathcal{A}}_{\vartheta}) \in R^{\mathcal{A}}$
$\mathcal{I} \models x \doteq x'$	*gdw.*	$[\![x]\!]^{\mathcal{A}}_{\vartheta} = [\![x']\!]^{\mathcal{A}}_{\vartheta}$
$\mathcal{I} \models \neg\varphi$	*gdw.*	$\mathcal{I} \not\models \varphi$
$\mathcal{I} \models \varphi \wedge \psi$	*gdw.*	$\mathcal{I} \models \varphi$ *und* $\mathcal{I} \models \psi$
$\mathcal{I} \models \varphi \vee \psi$	*gdw.*	$\mathcal{I} \models \varphi$ *oder* $\mathcal{I} \models \psi$
$\mathcal{I} \models \varphi \rightarrow \psi$	*gdw.*	*falls* $\mathcal{I} \models \varphi$ *dann* $\mathcal{I} \models \psi$
$\mathcal{I} \models \varphi \leftrightarrow \psi$	*gdw.*	$\mathcal{I} \models \varphi$ *genau dann wenn* $\mathcal{I} \models \psi$
$\mathcal{I} \models \exists x\ \varphi$	*gdw.*	*Es ein* $u \in U$ *gibt, sodass* $(\mathcal{A}, \vartheta[x \mapsto u]) \models \varphi$
$\mathcal{I} \models \forall x\ \varphi$	*gdw.*	*Für alle* $u \in U$ $(\mathcal{A}, \vartheta[x \mapsto u]) \models \varphi$

Gilt $\mathcal{I} \models \varphi$, so heißt $\mathcal{I}$ auch Modell von φ. Die Modellklasse einer Formel besteht aus allen ihren Modellen.

Wenn eine Formel φ durch die Interpretation $\mathcal{I}$ als wahr ausgewertet wird, wird diese als $\mathcal{I}$ ist ein Modell von φ, $\mathcal{I}$ erfüllt φ oder einfach $\mathcal{I} \models \varphi$ bezeichnet. Sobald es eine solche Interpretation für eine Formel φ gibt, ist φ erfüllbar.

Definition 2.1.9. *Sei φ eine Formel.*

- *φ ist erfüllbar, wenn es eine Interpretation $\mathcal{I}$ gibt, sodass $\mathcal{I} \models \varphi$.*
- *φ ist unerfüllbar, wenn für alle Interpretationen $\mathcal{I}$, $\mathcal{I} \not\models \varphi$ gilt.*
- *φ ist allgemeingültig, wenn für alle Interpretation $\mathcal{I}$, $\mathcal{I} \models \varphi$ gilt.*

2.2 Model Checking

Das übergeordnete Ziel des Model Checkings ist die Verifikation, das automatisierte „Erbringen eines Beweises dass ein gegebenes Programm sich an eine gegebenen Spezifikation hält“ [Lan15]. Model Checking ist ferner ein Verfahren der formalen Verifikation. Dabei wird das Modell als mathematische Struktur $\mathcal{A}$ angesehen und die Korrektheitseigenschaft als logische Formal φ formuliert. Anschließend wird überprüft, ob die Struktur als logische Interpretation ein Modell der Formel ist [Lan15].

Um Model Checking im Unendlichen besser zu verstehen, wird es zunächst im Endlichen betrachtet. Dafür wird anstelle des bekannten Top-Down-Algorithmus ein Bottom-Up-Ansatz anhand der Definition von [Ott] genutzt.

Definition 2.2.1. *Einer prädikatenlogischen Formel φ mit n freien Variablen $x_1, ..., x_n$ kann unter der τ-Struktur $\mathcal{A}$, mit dem Universum U die von φ definierte n-stellige Relation $[\![\varphi]\!]^{\mathcal{A}} := \{(u_1, ..., u_n) \in U^n : (\mathcal{A}, \vartheta[x_i \mapsto u_i]) \models \varphi\} \subseteq U^n$ zugewiesen werden. Die von der Relation $[\![\varphi]\!]^{\mathcal{A}}$ definierte Menge entspricht dabei der Menge der Variablenbelegungen der freien Variablen von φ, um diese unter der Struktur $\mathcal{A}$ zu erfüllen.*

Zusätzlich wird folgende Methode zum vergrößern von Relationen definiert:

Definition 2.2.2. *Gegeben ist die Menge $M = \{(a_1, ..., a_k), ...\} \subseteq U^k$.*
Dann ist $M' \subseteq U^{k+k'}$ eine Erweiterung von M welche wie folgt definiert ist:
$M' = \{(a_1, ..., a_k, a'_1, ...a'_k), ... : (a_1, ..., a_k) \in M; (a'_1, ..., a'_k) \in U^{k'}\}$

Anhand Definition 2.2.1 kann folgendes Verfahren abgeleitet werden, welches stufenweise von unten nach oben (Bottom-Up) die Formel durchläuft und dabei eine Menge erzeugt. Im Folgenden werden für den Fall einer Relation nur diese betrachtet, in welchen Variablen jeweils einmal vorkommen. Formeln mit Relationen in welchen Variablen mehr als einmal genutzt werden, müssen zunächst durch das existentielle Binden und Sicherstellen der Gleichheit umgeformt werden.
$\varphi = R(x_1, ..., x_n)$ mit $x_i = x_j \mathrel{\widehat{=}} \varphi' = \exists x_i R(x_1, ..., x_n) \wedge x_i = x_j$

Nun wird für jeden Bestandteil der Formel die entsprechende Menge bestimmt:

$$
\begin{aligned}
[\![R(x_1...x_k)]\!]^{\mathcal{A}} &:= \{(u_1, ..., u_k) \in R^{\mathcal{A}}\} \\
[\![x = x']\!]^{\mathcal{A}} &:= \{(u, u) : u \in U\} \\
[\![\neg\varphi]\!]^{\mathcal{A}} &:= U^n \setminus [\![\varphi]\!]
\end{aligned}
$$

Bei den Operationen $\wedge$ und $\vee$ zwischen den beiden Formeln φ und ψ kommt zunächst Definition 2.2.2 zum Einsatz. Dadurch werden beiden Relationen so vergrößert, dass diese dieselbe Menge von freien Variablen beschreiben.

$$
\begin{aligned}
[\![\varphi \wedge \psi]\!]^{\mathcal{A}} &:= [\![\varphi]\!]^{\mathcal{A}} \cap [\![\psi]\!]^{\mathcal{A}} \\
[\![\varphi \vee \psi]\!]^{\mathcal{A}} &:= [\![\varphi]\!]^{\mathcal{A}} \cup [\![\psi]\!]^{\mathcal{A}}
\end{aligned}
$$

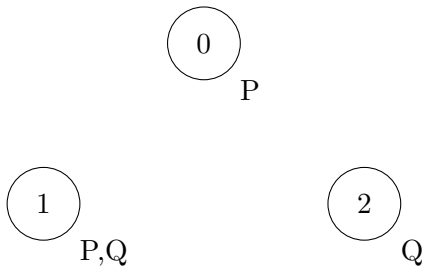
Bei den nachfolgenden Projektionen ist zu beachten, dass sich die Stelligkeit der in den Subformeln definierten Relation ändert.

$$
\begin{aligned}
[\![\exists x_i\ \varphi]\!]^{\mathcal{A}} &:= \{(u_1, ..., u_n) \in U^n : \exists u'_i \in U \text{ mit } (u_1, ..., u'_i, ..., u_n) \in [\![\varphi]\!]^{\mathcal{A}}\} \\
[\![\forall x_i\ \varphi]\!]^{\mathcal{A}} &:= \{(u_1, ..., u_n) \in U^n : \forall u'_i \in U \text{ mit } (u_1, ..., u'_i, ..., u_n) \in [\![\varphi]\!]^{\mathcal{A}}\}
\end{aligned}
$$

Diese dadurch erzeugte Menge beinhaltet, wenn die Struktur $\mathcal{A}$ die Formel φ ($\mathcal{A} \models \varphi$) erfüllt, eine Variablenbelegung oder im anderen Fall ($\mathcal{A} \not\models \varphi$) die leere Menge.

Somit würde für eine n-stellige Formel φ eine Menge erwartet werden, in welcher sich mindestens ein n-stelliges Tupel $\{(x_1, ..., x_n) : x_i \in U\} \subseteq U^n$ befindet. Im Falle einer 0-stelligen Formel haben die Tupel eine Länge von 0, woraus sich 2 mögliche Mengen ergeben, $\{\}$ und $\{()\}$. Nun hängt es davon ab, ob $R^{\mathcal{A}} = \{()\}$ ($\mathcal{A} \models \varphi$) oder $R^{\mathcal{A}} = \{\}$ ($\mathcal{A} \not\models \varphi$) [Lan18]. Durch Definition 2.2.1 ist eine Relation so definiert, dass diese die Modellbeziehung zwischen $\mathcal{A}$ und φ abbildet. Daher ist auch bekannt dass im Falle einer 0-stelligen Relation diese durch die Menge mit dem leeren Tupel ($\{()\}$) erfüllt wird und damit $\mathcal{A} \models \varphi$.

Abbildung 2.1: Beispielstruktur $\mathcal{A}$



Im Folgenden wird die Definition 2.2.1 schrittweise mit Erläuterungen auf zwei Formeln angewendet.

Beispiel 2.2.3. *Gegeben ist die Formel $\varphi_2 = \forall x(P(x) \vee Q(x))$ und die Struktur $\mathcal{A}$ aus Abbildung 2.1.*

Es werden alle Variablenbelegungen gesucht, welche die Subformeln von φ_2 erfüllen:

$$P(x) := \{(0), (1)\}$$
$$Q(x) := \{(1), (2)\}$$

Die Ergebnismengen von $P(x)$ und $Q(x)$ werden vereinigt:

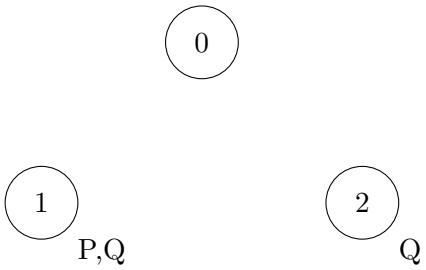
$$P(x) \vee Q(x) := \{(0), (1)\} \cup \{(1), (2)\} = \{(0), (1), (2)\}$$

Es wird eine Projektion auf x durchgeführt, da für alle Belegungen von x ein Tupel in der Ergebnismenge von $P(x) \vee Q(x)$ enthalten ist:

$$\forall x(P(x) \vee Q(x)) := \{()\}$$

Struktur $\mathcal{A}$ ist Modell der Formel φ_2.

Abbildung 2.2: Beispielstruktur $\mathcal{B}$



Beispiel 2.2.4. *Gegeben ist die Formel* $\varphi_3 = \forall x \exists y((P(x) \vee Q(y)) \wedge Q(x))$ *und die Struktur* $\mathcal{B}$ *aus Abbildung 2.2.*

Es werden alle Variablenbelegungen gesucht, welche die Subformeln von φ_3 *erfüllen:*

$$P(x) = \{(1)\}$$
$$Q(y) = \{(1), (2)\}$$
$$Q(x) = \{(1), (2)\}$$

Nun wird die in Definition 2.2.2 eingeführte Operation angewendet, um die Tupel der Ergebnismenge von $P(x)$ *sowie* $Q(y)$ *auf einen gemeinsamen Nenner zu bringen:*

$$P(x, y) := \{(1, 0), (1, 1), (1, 2)\}$$
$$Q(x, y) := \{(0, 1), (1, 1), (2, 1), (0, 2), (1, 2), (2, 2)\}$$

Diese beiden Mengen können anschließend vereinigt werden.

$$P(x, y) \vee Q(x, y) := \{(0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 1), (2, 2)\}$$

Wie auch im vorherigen Schritt, müssen die beiden Ergebnismengen von $(P(x, y) \vee Q(x, y))$ *und* $Q(x, y)$ *auf einen gemeinsamen Nenner gebracht werden. Die Ergebnismenge von* $(P(x, y) \vee Q(x, y))$ *beinhaltet schon Tupel für* (x, y)*.* $Q(x, y)$ *analog zum vorherigen Schritt:*

$$Q(x, y) := \{(1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2)\}$$

Diese beiden Mengen werden anschließend geschnitten:

$$(P(x, y) \vee Q(x, y)) \wedge Q(x, y) := \{(1, 0), (1, 1), (1, 2), (2, 1), (2, 2)\}$$

Es wird eine Projektion auf y *durchgeführt und jedes Tupel aus der Ergebnismenge von* $(P(x, y) \vee Q(x, y)) \wedge Q(x, y)$ *um die* y *Variable gekürzt:*

$$\exists y(P(x) \vee Q(x)) \wedge Q(x) := \{(1), (2)\}$$

Es wird probiert eine Projektion auf x durchzuführen, aber nur für die Tupel, welche mit jeder Belegung in der Ergebnismenge von $\exists y(P(x) \vee Q(y)) \wedge Q(x)$ enthalten sind. Da die Menge nur die beiden Tupel $(1), (2)$ und nicht (0) beinhaltet kann keine Projektion durchgeführt werden:

$$\forall x \exists y(P() \vee Q()) \wedge Q() := \{\}$$

Struktur $\mathcal{A}$ ist kein Modell der Formel φ_3.

3 Automatische Strukturen

Model Checking auf unendlichen Strukturen ist im Allgemeinen unentscheidbar [Lan18]. Es ist jedoch bekannt dass Model Checking für eine Struktur entscheidbar ist, falls für diese und jede relevante Teilstruktur eine endliche Repräsentation gefunden werden kann. Diese Repräsentation muss jedoch unter Schnitt, Vereinigung, Komplement sowie Projektion und Spuren-Tausch abgeschlossen sein.

Automatische Strukturen sind generell unendliche relationale Strukturen, dessen Universum und atomare Relationen von einem endlichen Automaten erkannt werden können [KNRS07].

Die grundsätzliche Idee dahinter ist, eine Struktur vollständig durch endliche Automaten darzustellen. Ist dies möglich, wird diese Struktur automatisch präsentierbar genannt. Im Kontext dieser Arbeit bedeutet das, dass die unterliegende Menge immer Σ^* für ein passendes Σ ist. Im Falle einer einstelligen Relation R^1 ist das unkompliziert. Es wird ein Automat konstruiert, welcher diese Relation R^1 definiert. Um aber eine n-stellige Relationen R^n definieren zu können, wird ein Automat benötigt, welcher synchron n Eingabewörter lesen kann, um zu entscheiden, ob diese Teil der Relation R^n sind oder nicht. Mit diesen synchronen Mehrspurautomaten beschäftigt sich der folgende Abschnitt.

3.1 Synchrone Mehrspurautomaten

Um einen synchronen n-Spurenautomaten $\mathscr{A}$ zu definieren, wird zunächst das Format der Eingaben betrachtet, welche gelesen werden können. Dieses zeichnet sich dadurch aus, dass jede Eingabe $w = (w_1, ..., w_n)$ mit $w_i \in \Sigma^*$ genau n Spuren besitzt, und zwar $w_1, ..., w_n$. w wird im folgenden als „Wort“ bezeichnet, welches n Spuren besitzt. Die Schreibweise $w = (w_1, ..., w_n)$ bezeichnet dabei kein übliches Tupel, sondern Spuren welche „untereinander“ geschrieben werden.

Da es jedoch möglich ist, dass Spuren eines Wortes nicht immer dieselbe Länge haben, gibt es eine Art Vorverarbeitung um die Spuren eines Wortes auf dieselbe Länge zu bringen. Dafür wird ein Padding-Symbol $\square \notin \Sigma$ genutzt, welches auf der rechten Seite der zu kurzen Spuren solange angehängt wird, bis diese dieselbe Länge haben. Das Eingabealphabet

eines synchronen Mehrspurautomat ist dementsprechend folgendermaßen definiert $\Sigma_{\square}^{n} = (\Sigma \cup \{\square\})^{n}$.

Die Vorverarbeitung kann wie eine Funktion angesehen werden, in welche ein Wort w mit n Spuren hereingegeben wird und ein Wort w' mit n gleichlangen Spuren zurückgegeben wird. Diese Funktion sucht dabei zuerst die längste Spur und verlängert anschließend alle anderen Spuren um die jeweilige Anzahl an Padding Symbolen, bis diese dieselbe Länge haben. Definiert ist diese wie folgt [BG00]:

Definition 3.1.1. *Gegeben ist ein Wort $w = (w_1, ..., w_n)$ mit n-Spuren. Eine Spur $w_i = (w_i^1, w_i^2, ..., w_i^{l_i})$ mit $1 \leq i \leq n$ hat die Länge $l_i = |w_i|$. $l_{max} = max\{l_1, ..., l_n\}$ gibt an wie lang die längste Spur des Wortes w ist.*

$$w_i' = \begin{cases} w_i^1 ... w_i^{l_i} \underbrace{\square \cdots \square}_{l_{max} - l_i \text{ mal}} & \text{für } l_i < l_{max} \\ w_i^1 ... w_i^{l_i} & \text{sonst} \end{cases}$$

$$w' = \begin{bmatrix} w_1'^1 \\ \vdots \\ w'1_n \end{bmatrix} \ldots \begin{bmatrix} w_1'^{l_{max}} \\ \vdots \\ w_n'^{l_{max}} \end{bmatrix} \in ((\Sigma \cup \{\square\})^n)^*$$

Das Wort w' besteht dabei am Ende aus genau l_{max} n-stelligen Tupeln. Dieses Format wird benötigt, damit der Automat das Wort schrittweise synchron einlesen kann.

Beispiel 3.1.2. *Gegeben ist das Alphabet $\Sigma = \{a, b\}$ und ein Wort $w = (w_1, w_2)$ mit den Spuren $w_1 = aabbaa$ und $w_2 = abab$ wobei $w_1, w_2 \in \Sigma^*$. Damit dieses Wort von einem synchronen 2-Spurenautomat gelesen werden kann, wird es zunächst mithilfe des in Definition 3.1.1 vorgestellten Verfahrens vorverarbeitet. Das Resultat ist das Wort $w' = (w_1', w_2')$ mit den Spuren $w_1' = aabbaa$ und $w_2' = abab\square\square$, wobei $w_1', w_2' \in ((\Sigma \cup \{\square\})^2)^*$.*

Um einen synchronen Mehrspurautomaten zu konstruieren, wird die Definition eines nicht deterministischen endlichen Automaten (NFA) genutzt, um folgenden synchronen nicht-deterministischen endlichen n-Spuren Automaten (sNFA) mit dem speziellen Eingabealphabet $(\Sigma \cup \{\square\})^n$ zu definieren.

Definition 3.1.3. *Ein n-Spur sNFA ist ein Tupel $\mathscr{A} = (Q, \Sigma_{\square}^{n}, \Delta, q_0, F)$ wobei*

- *Q eine endliche, nicht leere Menge von Zuständen,*
- *$\Sigma_{\square}^{n} = (\Sigma \cup \{\square\})^n$, wobei Σ ein endliches, nicht leeres Eingabealphabet,*
- *$\Delta \subseteq Q \times \Sigma_{\square}^{n} \times Q$ die Transitionsrelation,*

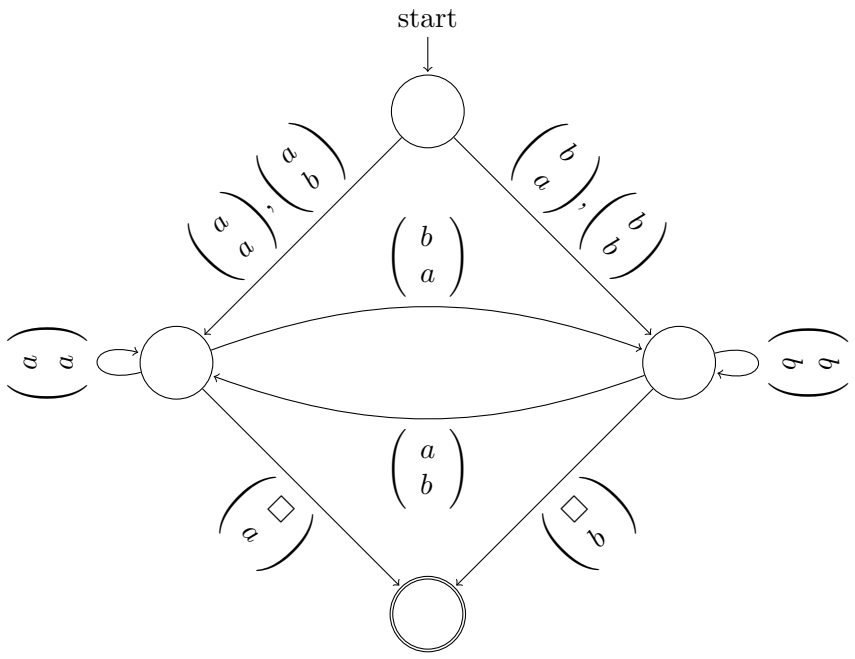
- $q_0 \in Q$ *Startzustand und*
- $F \subseteq Q$ *ist eine Menge von Endzuständen*

Ein sNFA funktioniert analog zum NFA. Ein Lauf ist eine Folge von Zuständen $p = q_0, q_1, q_2, ...$ mit $q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2...$ wobei $a_i \in \Sigma$. Ein solcher Lauf wird akzeptierend genannt, wenn dieser einen Zustand aus F beinhaltet. Ein Eingabewort w wird als akzeptiert bezeichnet, falls es dafür einen akzeptierenden Lauf gibt. Alle akzeptierenden Wörter bezeichnen die Sprache $\mathcal{L}(\mathscr{A})$[Kum].
Die Sprache $\mathcal{L}(\mathscr{A})$ eines sNFAs ist unter Vereinigung, Schnitt sowie Komplement abgeschlossen.

Satz 3.1.4. *sNFAs sind unter Schnitt, Vereinigung, Komplement sowie Konkatenation und Kleene-Stern abgeschlossen [HMU06].* □

Damit in der folgenden Implementierung jede Spur eines Automaten angesprochen werden kann, repräsentiert ein n-Spur Automaten eine n-stellige Relation $R \subseteq \Sigma^* \times ... \times \Sigma^*$, welcher durch $R(x_1, ..., x_n)$ benannt wird. Ein solcher Automat kann beispielsweise wie in Abbildung 3.1 aussehen.

Abbildung 3.1: 2-Spur Automat, welcher die Relation $R(x, y)$ repräsentiert. Diese Relation beinhaltet die Wortpaare, bei welchen das letzte Zeichen des zweiten Wortes gleich dem des ersten, aber um ein Symbol länger ist.



Zusätzlich zu den Operationen Schnitt, Vereinigung und Komplement wird eine Möglichkeit benötigt, bestimmte Spuren wegzuprojizieren, damit diese Automaten für das Model Checking mithilfe des Bottom-Up-Verfahren genutzt werden können. Um für einen Automat $\mathscr{A}$ mit der Stelligkeit n die Spur k wegzuprojizieren, wird ein Automat $\Pi_k(\mathscr{A})$ mit $n-1$ Spuren konstruiert, welcher somit nur Wörter mit $n-1$ Spuren akzeptiert. Dabei wird auch die Transitionsrelation so definiert, dass diese auch nur noch Eingabewörter mit $n-1$ Spuren akzeptiert, indem die Spur k herausprojiziert wird.

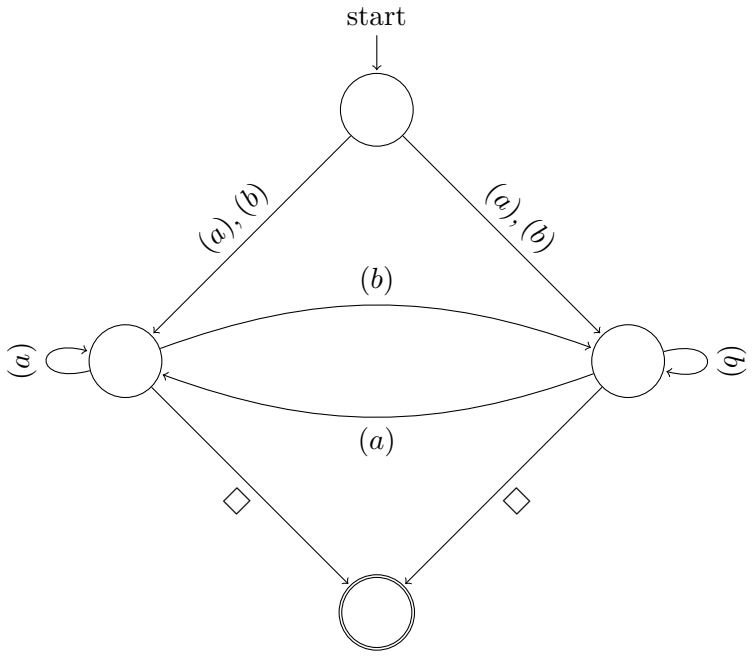
Satz 3.1.5. *Der Automat $\Pi_k(\mathscr{A}) = (Q, \Sigma_\square^{n-1}, \Delta', q_0, F)$ akzeptiert die Sprache des Automaten $\mathscr{A}$ ohne Spur k [Fur12], wobei*

$$\Delta' = \left\{ \left(q, \begin{pmatrix} a_1 \\ \vdots \\ a_{k-1} \\ a_{k+1} \\ \vdots \\ a_n \end{pmatrix}, q' \right) : \exists a_k \in \Sigma_\square^n \text{ mit } \left(q, \begin{pmatrix} a_1 \\ \vdots \\ a_{k-1} \\ a_k \\ a_{k+1} \\ \vdots \\ a_n \end{pmatrix}, q' \right) \in \Delta \right\}.$$

Der neue Automat $\Pi_k(\mathscr{A})$ akzeptiert damit also die Wörter, welche sich zwischen Spur $k-1$ und $k+1$ so ergänzen lassen, so dass sie in $\mathcal{L}(\mathscr{A})$ liegen.

Beispiel 3.1.6. *Der Automat in Abbildung 3.1 akzeptiert die Relation $R(x, y)$. Abbildung 3.2 zeigt diesen nach dem wegprojizieren der ersten Spur (x).*

Abbildung 3.2: 1-Spur Automat, welcher die Relation $R(y)$ abbildet.



Weiterhin wird eine Operation definiert um Spuren innerhalb von n-Spuren zu tauschen. Um die beiden Spuren p und p' miteinander zu tauschen, werden diese in der Transitionsrelationen miteinander vertauscht.

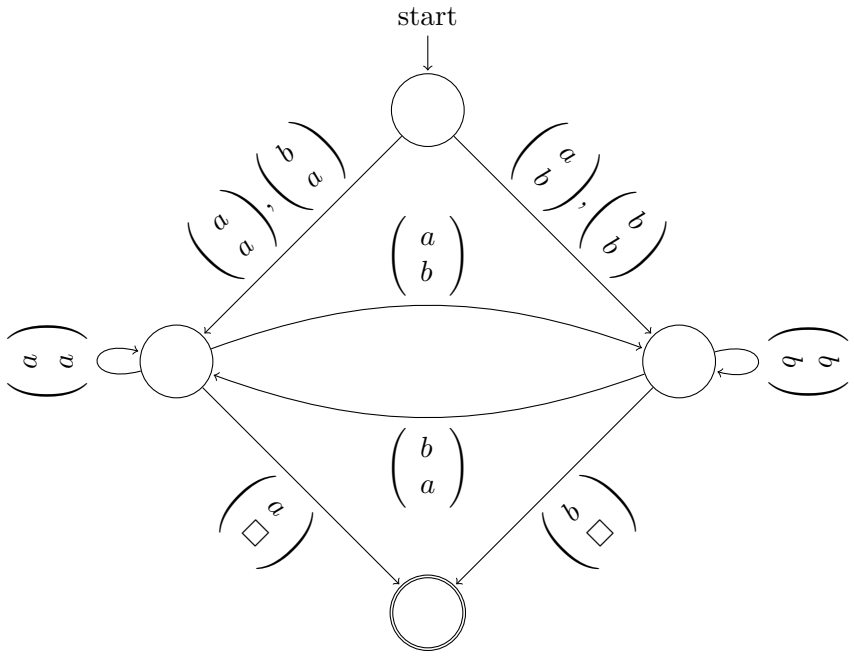
Definition 3.1.7. *Der Automat $\Xi_p(\mathscr{A}) = (Q, \Sigma_\square^n, \Delta', q_0, F)$ akzeptiert die Sprache des Automaten $\mathscr{A}$ bei welcher die Spuren p und p' miteinander getauscht wurden [Fur12].*

$$\left\{\left(q, \begin{pmatrix} a_1 \\ \vdots \\ a_{p-1} \\ a_{p'} \\ a_{p+1} \\ \vdots \\ a_{p'-1} \\ a_p \\ a_{p'+1} \\ \vdots \\ a_n \end{pmatrix}, q'\right)\right\} \in \Delta' \text{ gdw. } \left\{\left(q, \begin{pmatrix} a_1 \\ \vdots \\ a_{p-1} \\ a_p \\ a_{p+1} \\ \vdots \\ a_{p'-1} \\ a_{p'} \\ a_{p'+1} \\ \vdots \\ a_n \end{pmatrix}, q'\right)\right\} \in \Delta$$

Der neue Automat $\Xi_p(\mathscr{A})$ akzeptiert damit also die Sprache der Wörter aus $\mathscr{A}$, wobei die Spuren p und p' miteinander vertauscht wurden.

Beispiel 3.1.8. *Der Automat in Abbildung 3.1 akzeptiert die Relation $R(x, y)$. Um nun einen Automaten zu konstruieren, welcher die Relation $R(y, x)$ akzeptiert müssen die Spuren wie in Abbildung 3.3 getauscht werden.*

Abbildung 3.3: 2-Spuriger Automat, welcher die Relation $R(y, x)$ abbildet.



Nachdem nun bekannt ist, wie synchrone Mehrspurautomaten funktionieren, können automatische Strukturen wie folgt definiert werden:

Definition 3.1.9. *Eine Struktur $\mathcal{A} = (U, R_1^{s_1}, ..., R_m^{s_m})$ (wobei s_i die Stelligkeit der Relation ist) wird automatisch genannt, wenn*

- $U = \Sigma^*$ *und*
- *es für jede s_i-stellige Relation R einen s_i spurigen Automaten $\mathcal{A}_R$ über dem Eingabealphabet $\Sigma_{\square}^{s_i}$ gibt, welcher die Relation definiert.*

3.2 Model Checking

Model Checking für automatische Strukturen funktioniert analog mit dem in Kapitel 2.2 vorgestellten Bottom-Up Verfahren. Ein wichtiger Unterschied dabei ist jedoch, dass es sich nicht um eine normale Struktur handelt, sondern um eine automatische Struktur, welche durch die in Kapitel 3.1 vorgestellten synchronen Mehrspurautomaten beschrieben werden.

Die Herangehensweise ist dabei ähnlich der bei normalen Strukturen. Es wird mithilfe der gegebenen automatischen Struktur und der n-stelligen Formel eine n-stellige Relation konstruiert, welche in diesem Fall aber von einem Automaten definiert wird. Dabei wird für jede Unterformel mit n freien Variablen ein n-Spuren sNFA konstruiert. Dieser akzeptiert genau die n spurigen Wörter, welche auch in der Relation sind, die diese Unterformel definiert, wobei jedes Wort eines Tupels als eine Spur gelesen wird. Dafür wird folgendes Verfahren genutzt:

- $[\![R(x_1...x_k)]\!]^{\mathscr{A}}$:= Ein k-Spuren sNFA, welcher alle Wörter der Relation R akzeptiert, wobei jedes Wort eines Tupels aus R als Spur gelesen wird.
- $[\![x = x']\!]^{\mathscr{A}}$:= Ein 2-Spuren sNFA, welcher nur zwei gleiche Wörter akzeptiert. Dies ist ein Ein-Zustand Automat, welcher nur Transitionsrelationen besitzt bei welchen das Eingabesymbol identisch ist.
- $[\![\neg\varphi]\!]^{\mathscr{A}}$:= Komplement Automat von φ, welcher durch die bekannte Automatentheorie gebildet wird.
- $[\![\varphi \wedge \psi]\!]^{\mathscr{A}}$:= Automat aus dem Schnitt von φ und ψ, welcher durch die bekannte Automatentheorie gebildet wird. Damit dies aber möglich ist, werden beide Automaten vorher auf das selbe Format gebracht, Das bedeutet, dass die Anzahl der

Spuren beider Automaten so vergrößert wird, damit beide Automaten dieselbe Anzahl von Spuren lesen können. Dabei wird eine umgekehrte Projektion durchgeführt. Außerdem werden die Spuren noch anhand ihrer Variablen sortiert (Siehe Definition 3.1.7).

- $[\![\varphi \vee \psi]\!]^{\mathscr{A}}$:= Automat aus der Vereinigung von φ und ψ, welcher durch die bekannte Automatentheorie gebildet wird. Wie auch beim $\wedge$ werden die beiden Automaten zuerst in dasselbe Format gebracht.

- $[\![\exists x_i\ \varphi]\!]^{\mathscr{A}}$:= Projektionsautomat des Automaten für φ, siehe Definition 3.2.

Der Allquantor $\forall$ ist hier nicht aufgeführt, da es schwierig ist einen Automaten anzugeben, welcher diesen abbilden kann. Dieser wird mithilfe des Existenzquantors $\exists$ durch die Umformung $\forall x\varphi \equiv \neg\exists x\neg\varphi$ aufgelöst.

Um nun zu entscheiden, ob die automatische Struktur Modell der Formel ist, muss geprüft werden ob die Sprache des konstruieren Automaten $\mathscr{A}_\varphi$ leer ist.

Satz 3.2.1. *$\mathcal{A}, \vartheta \models \varphi$ gdw. die Sprache des konstruieren Automaten $L(\mathscr{A}_\varphi) \neq \emptyset$.*

Das wird dadurch ersichtlich, ob es einen Pfad vom Anfangszustand zum Endzustand gibt. Dieser Pfad wird berechnet, indem die Menge aller vom Startzustand aus erreichbaren Zustände induktiv berechnet werden. Ein Zustand q wird erreichbar bezeichnet, wenn es einen Zustandsübergang von Zustand p nach q gibt. Befindet sich in der Menge aller erreichbaren Zustände ein Endzustand, dann ist die Sprache des Automaten nicht leer.

Beispiel 3.2.2. *Gegeben ist eine automatische Struktur $\mathscr{A}$ mit den zweistelligen Relationen P und Q und die prädikatenlogische Formel $\varphi = \exists x\exists y(P(x,y) \wedge Q(x,y))$, auf welcher schrittweise das gerade beschriebene Verfahren angewendet wird. Die beiden zweistelligen Relationen werden durch folgende 2-spurige Automaten repräsentiert:*

Abbildung 3.4: Zweispuriger Automat: P

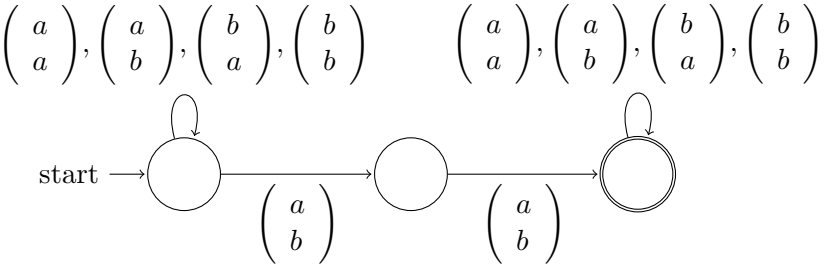
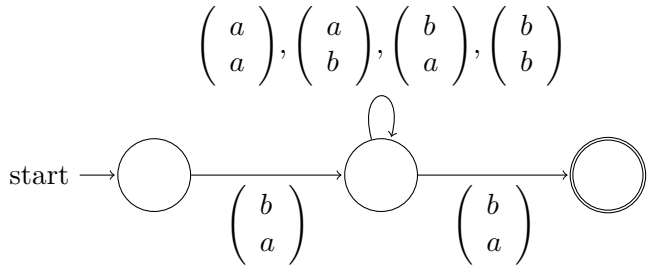
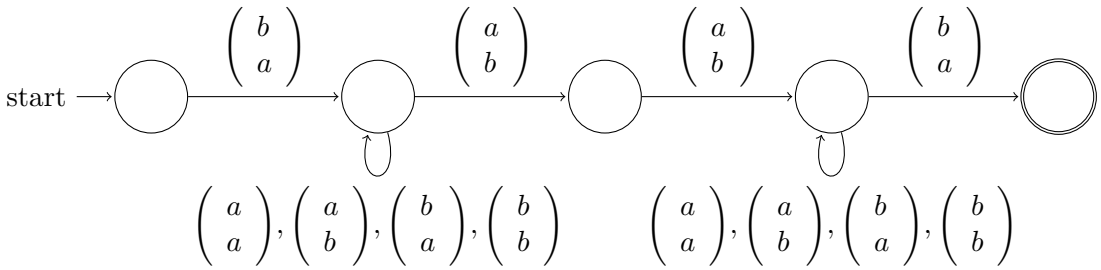


Abbildung 3.5: Zweispuriger Automat: Q



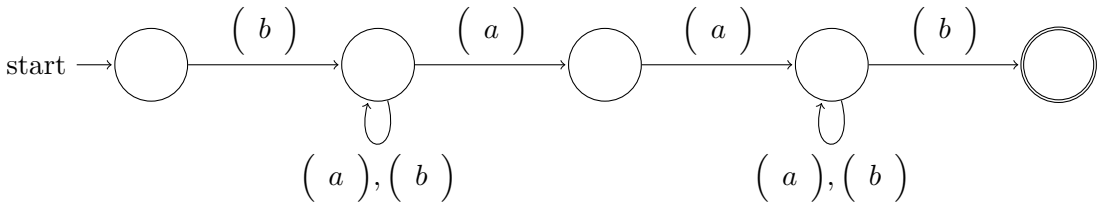
Nun wird der Automat aus dem Schnitt von P und Q gebildet:

Abbildung 3.6: Zweispuriger Schnittautomat: $P(x, y) \wedge Q(x, y)$



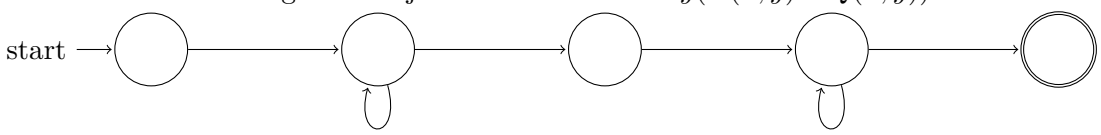
Anschließend wird der Projektionsautomat für $\exists y$ gebildet:

Abbildung 3.7: Einspuriger Projektionsautomat: $\exists y(P(x, y) \wedge Q(x, y))$



Sowie auch für der Projektionsautomat für $\exists x$ gebildet wird:

Abbildung 3.8: Projektionsautomat: $\exists x \exists y(P(x, y) \wedge Q(x, y))$



Abschließend wird Definition 3.2.1 angewendet, durch welche ersichtlich ist dass $\mathscr{A} \models \varphi$.

4 Umsetzung

In diesem Kapitel wird zunächst der technologische Hintergrund sowie die Funktionsweise der bestehenden Implementierung des Model-Checkers von M.Sc. Arno Ehle vorgestellt. Anschließend wird die Funktionsweise des Beweismodus beschrieben, welche die erste Erweiterung des Model-Checkers war. Abschließend wird die Implementierung und Erweiterung um eine Bibliothek beschrieben, welche die Grundlage für das Model Checking sowie den Beweismodus für automatischen Strukturen ist.

4.1 Technologischer Hintergrund

4.1.1 Java

Java ist eine objektorientierte Programmiersprache, welche seit 1995 entwickelt wird und momentan in Version 10 verfügbar ist. Sie gehört zu den sogenannten Hochsprachen, welche deutlich abstrahierter von den Maschinensprachen sind.
Ein Grund, warum Java häufig genutzt wird, ist die Plattformunabhängigkeit. Möglich ist dies durch die Java-Laufzeitumgebung, welche auf der jeweiligen Plattform installiert sein muss. Ein weiterer Grund ist die syntaktische Nähe zu anderen Programmiersprachen, wie C oder C++. Jedoch verzichtet Java komplett auf die bekannte Zeigerarithmetik. Java zeichnet sich vor allem durch eine starke Typisierung aus. Das bedeutet, dass Fehler, wie zum Beispiel eine falsche Variablenzuweisung, direkt beim Kompiliervorgang auffallen. Weiterhin gibt es den sogenannten „Garbage Collector“, welcher dem Entwickler die gesamte Speicherverwaltung abnimmt. Eine weitere Besonderheit stellt die Ausnahmebehandlung dar, welche Möglichkeiten der Fehlerbehandlung zur Laufzeit bietet.
Listing 4.1 zeigt kompilier- und ausführbaren Javacode, welcher „Hello World!“ ausgibt.

Listing 4.1: Java Beispielprogramm - Hello World

```
1 public class Main {
2     public static void main(String[] args) {
3         System.out.println("Hello␣World!");
4     }
5 }
```

4.1.2 JavaFX

JavaFX ist ein Framework zur Erstellung grafischer Benutzeroberflächen (GUI) für Java Applikationen, welche plattformübergreifend eingesetzt werden können. Es wurde entwickelt, weil die damaligen Standartlösungen (AWT und Swing) in die Jahre gekommen waren und dessen Wartungsaufwand zu hoch wurde. Mithilfe von JavaFX ist es möglich eine GUI durch sogenanntes „FXML“ zu beschreiben, welches auf „XML“ basiert. Die zur Ausführung benötigte Laufzeitumgebung ist seit „Java SE Runtime 7 Update 6“ Teil der Java Installation. Listing 4.2 zeigt kompilier- und ausführbaren Javacode, welcher „Hello JavaFX World!“ mithilfe einer GUI ausgibt, die in Abbildung 4.1 zu sehen ist.

Listing 4.2: JavaFX Beispielprogramm - Hello JavaFX World!

```
1 import javafx.application.Application;
2 import javafx.scene.Scene;
3 import javafx.scene.control.Label;
4 import javafx.scene.layout.StackPane;
5 import javafx.stage.Stage;
6
7 public class Main extends Application {
8
9     @Override
10    public void start(Stage primaryStage) {
11        Label label = new Label("Hello␣JavaFX␣World!");
12
13        StackPane root = new StackPane();
14        root.getChildren().add(label);
15
16        Scene scene = new Scene(root, 400, 300);
17
18        primaryStage.setTitle("Hello␣JavaFX␣World!");
19        primaryStage.setScene(scene);
20        primaryStage.show();
21    }
22
23    public static void main(String[] args) {
24        launch(args);
25    }
26
27 }
```

Abbildung 4.1: Screenshot des JavaFX Beispielprogramms

4.1.3 Generic Types

Generics Types (Generische Elemente) sind eines der komplexeren Sprachmittel in Java. Mithilfe dieser werden Klassen oder Methoden um eine Variable für einen Typ erweitert. Dieser Typ ist während der Implementierung nicht bekannt und wird erst beim nutzen durch einen konkreten Typ ersetzt. Ziel ist es doppelte Implementierungen zu vermeiden, ohne aber die Typsicherheit zu verlieren. Ein bekanntes Beispiel ist eine „ArrayList“. Gewohnte Funktionen wie „add“, „remove“ oder „get“ bieten unabhängig vom Typ die selbe Funktionalität. Listing 4.3 zeigt eine beispielhafte Nutzung von Generics.

Listing 4.3: Generic Beispiel anhand einer ArrayList in Java

```
1 ArrayList<String> list1 = new ArrayList<>();
2 list1.add("First");
3 list1.add("Second");
4
5 ArrayList<Integer> list2 = new ArrayList<>();
6 list2.add(1);
7 list2.add(2);
8
9 for (String value : list1) {
10     System.out.println(value);
11 }
12
13 for (Integer value : list2) {
14     System.out.println(value);
15 }
```

4.1.4 Maven

Maven ist ein auf Java basierendes Build-Management Tool. Es versucht den gesamten Prozess der Softwareentwicklung abzubilden. Dieser ist dabei in Schritte aufgeteilt, wie das Validieren des Projekts (validate), das Kompilieren des Projekts (compile) oder auch das Erstellen einer ausführbaren Datei (package). Informationen, wie den Namen des Projekts oder die Autoren werden in einer Datei names „pom“ (Project Object Model) im XML Format abgelegt. Dort können auch Abhängigkeiten zu anderen Projekten oder Bibliotheken aufgelistet werden. Diese versucht Maven dann aufzulösen, indem es die benötigen Ressourcen im lokalen oder im Internet zur Verfügung gestellten Repositories sucht.

4.1.5 Externe Bibliotheken

Java und JavaFX bieten zwar ein gutes Grundset an Funktionen und Möglichkeiten, jedoch reichen diese nicht immer aus oder sind in der Bedienung etwas umständlich. Aus diesem Grund werden externe Bibliotheken eingesetzt, welche von Entwicklern erstellt, gepflegt und veröffentlicht werden. Das Veröffentlichen geschieht in der Regel über Maven oder andere Build-Management Tools, da diese die Handhabung und Versionierung, im Vergleich zum manuellen einbinden von „jar“ Files, stark vereinfachen.

ControlsFX

ControlsFX ist ein Open Source Projekt für JavaFX, welches hochwertige UI Elemente bereit stellt. Es beinhaltet eine Menge Komponenten, welche nicht in JavaFX enthalten sind oder deren Nutzung verbessert wurde. So gibt es beispielsweise die Möglichkeit der Validierung von Eingabefeldern oder auch der Anzeige von Notifications und Popovers, sowie viele weitere, welche in der Dokumentation [Gil] gefunden werden können.

javatuples

Javatuples ist eine Bibliothek, welche die aus anderen Programmiersprachen bekannten „Tuples“ in Java verfügbar macht. Ein Tupel ist eine Sequenz von Objekten (auch verschiedenen Typen), welche nicht zwangsläufig etwas miteinander zu tun haben, aber im selben Kontext benötigt werden. So ist es beispielsweise möglich bei einer Funktion, welche nur ein Element zurückgeben kann, ein Tupel als Rückgabe-Typ zu nutzen um mehrere Elemente zurückzugeben. Weitere Informationen sind in der Dokumentation [Fer] ersichtlich.

4.2 Zugrundeliegende Implementierung

Die zugrundeliegende Implementierung dieser Arbeit ist ein Model Checking Tool für endliche Strukturen von M.Sc. Arno Ehle. Es besteht aus den drei Modulen: Formelparser, visuellem Graphentool sowie Model Checker. In einer früheren Projektarbeit wurde dieses Tool von mir um ein weiteres Modul, einen Beweismodus für endliche Strukturen, erweitert. Dieser bildet in Verbindung mit den anderen Modulen die Ausgangsbasis meiner Implementierung.

Für die Integration von automatischen Strukturen mussten Teile der Implementierung dieser Module angepasst und erweitert werden, weshalb deren Aufbau und Funktionsweise anschließend kurz erläutert wird.

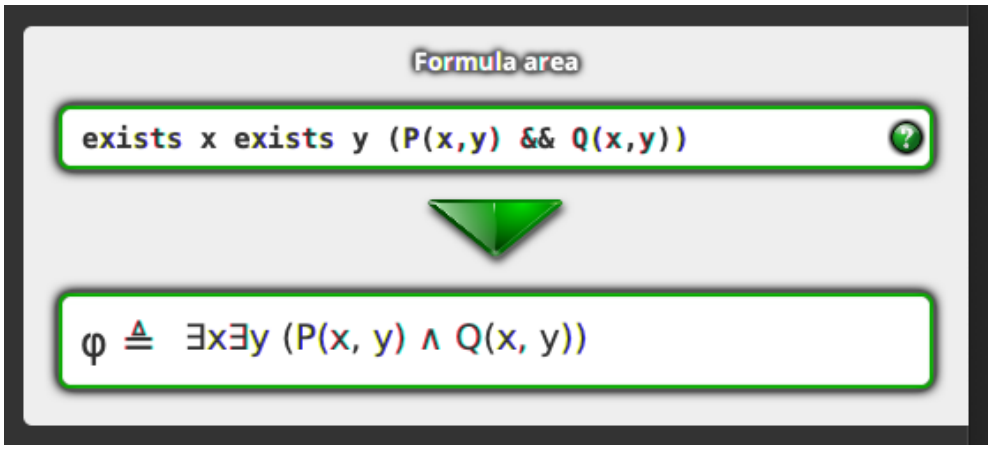
Der Formelparser dient dazu, eine vom Nutzer in Schriftform (als String) eingegebene Formel in ein für der Programm nutzbares Format zu parsen. Dafür wurde folgende Zuordnung definiert, um alle Bestandteile einer Formel ohne Umweg eingegeben zu können.

Wort	Zuordnung
!	$\neg$
tt	`tt`
ff	`ff`
||	$\vee$
&&	$\wedge$
->	$\rightarrow$
<->	$\leftrightarrow$
exists	$\forall$
forall	$\exists$

Funktion, Prädikate, Konstanten und Variablen können wiederum wie gewohnt eingegeben werden.

Sobald der Nutzer den Button zum parsen betätigt, wird die als String eingegebene Formel schrittweise von vorne nach hinten gelesen und in die Datenstruktur `FOLFormula` überführt. Dabei wird der Type der Formel (Prädikat, Und-Verknüpfung, Existenzquantor, ...) erkannt und gespeichert, um anhand dessen beim Model Checking zu bestimmen welche Operation durchgeführt werden muss. Zusätzlich wird jede Formel mit den nachfolgenden Subformeln (und umgekehrt) in einer Eltern-Kind-Beziehung gespeichert, um Abhängigkeiten einfacher handhaben zu können. Sollten beim Parsen Fehler auftreten, werden diese dem Nutzer in einem Dialog angezeigt. Anderenfalls wird das Resultat des parsens dem Nutzer mit den entsprechenden Symbolen unter der eingegebenen Formel angezeigt.

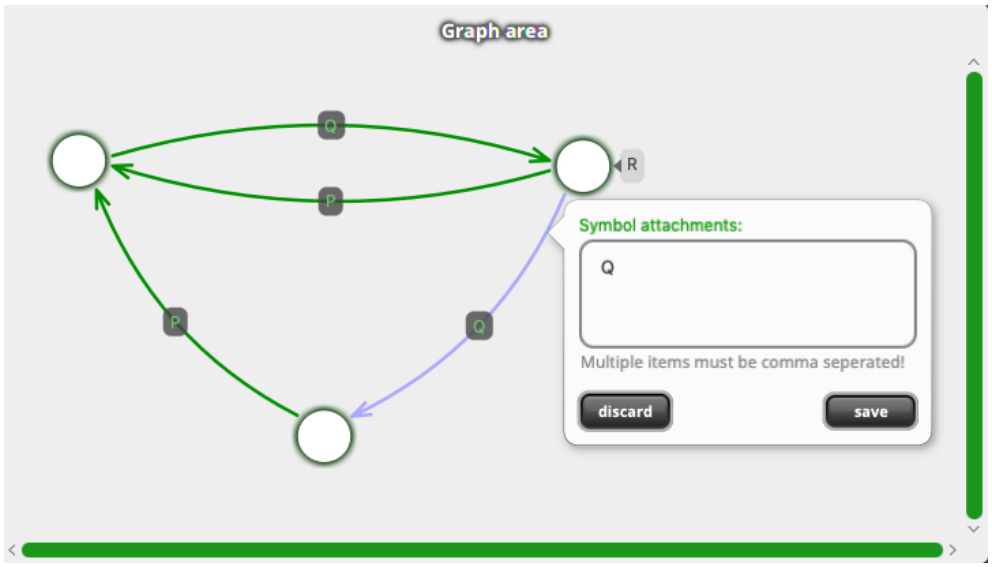
Abbildung 4.2: Parsen einer Beispielformel



Als Eingabemöglichkeit einer endlichen Struktur wird ein visuelles Graphentool bereitgestellt. Da die Eingabe einer automatischen Struktur (aufgrund der bekannten Gegebenheiten) mit diesem Tool für endliche Graphen nicht möglich ist, sei dieses nur der Vollständigkeit halber erwähnt.

Mit dem Tool ist es dem Nutzer möglich Knoten und Kanten zu erstellen sowie bestimmte Eigenschaften an diesen zu definieren. Die Knoten (in der Implementierung als `Vertex` Objekt definiert) und Kanten (in der Implementierung als `Edge` Objekt definiert) sind dabei streng bidirektional verbunden. Sprich, jeder Knoten kennt alle eingehenden und ausgehenden Kanten und jede Kante kennt den Start- und Ziel-Knoten.

Abbildung 4.3: Visuelles Graphentool mit Beispielgraph



4.2.1 Model Checker

Der Model Checker ist das Kernmodul der Anwendung. Dieser arbeitet mit der geparsten `FOLFormula` und der `Vertex-Edge` Datenstruktur, um eine Aussage zu treffen ob die definierte Struktur Model der eingegebenen Formel ist.

Betätigt der Nutzer den Button für das Model Checking, werden zunächst einige Bedingungen überprüft (z.B. ob es eine korrekt geparste Formel sowie Struktur gibt) und führt ein initales Setup des Model Checkers aus. Dazu gehört jeweils das Erstellen und Füllen einer HashMap für einstellige- bzw. zweistellige Prädikatsymbole. In der HashMap der einstelligen Prädikatsymbole wird für jedes Symbol, welches in der Struktur an einem `Vertex` definiert ist, einem Set aller `Vertex` hinterlegt welche auch dieses Symbol haben. Selbiges wird auch für zweistellige Prädikatsymbole durchgeführt, aber mit einem Set von `Edge` anstelle `Vertex`.

Listing 4.4: Pseudocode: HashMap der Einstelligen- und Zweistelligen-Prädikatsymbole

```
1 private HashMap<String, Set<Vertex>> oneArySymbolTable = [
2     "P": [Vertex1, Vertex2],
3     "Q": [Vertex2, Vertex3]
4 ];
5 private HashMap<String, Set<Edge>> twoArySymbolTable = [
6     "L": [Edge1,Edge2,Edge3],
7     "N": [Edge4]
8 ];
```

Anschließend beginnt der Algorithmus und arbeitet sich, je nach Formel Type, rekursiv durch die `FOLFormula` und gibt ein Booleschen Werte zurück, je nachdem ob die Struktur model der Formel ist oder nicht. Wie sich der Algorithmus je nach Type verhält, wird in nachfolgender Liste beschrieben.

!	Die Subformel wird in den Algorithmus gegeben, das Ergebnis jedoch negiert zurückgegeben.
||	Die beiden Subformeln werden in den Algorithmus gegeben, die Disjunktion der beiden Ergebnisse wird zurückgegeben.
&&	Die beiden Subformeln werden in den Algorithmus gegeben, die Konjunktion der beiden Ergebnisse wird zurückgegeben.
->	Die beiden Subformeln werden in den Algorithmus gegeben, die Implikation der beiden Ergebnisse wird zurückgegeben.
<->	Die beiden Subformeln werden in den Algorithmus gegeben, die Äquivalenz der beiden Ergebnisse wird zurückgegeben.
exists	Für jeden `Vertex` in der Struktur wird die Subformel in den Algorithmus gegeben. Zuvor wird der `Vertex` aber an die quantifizierte Variable gebunden. Es wird `TRUE` zurückgegeben, falls mindestens ein Ergebnis `TRUE` zurückgeliefert hat.
forall	Für jeden `Vertex` in der Struktur wird die Subformel in den Algorithmus gegeben. Zuvor wird der `Vertex` aber an die quantifizierte Variable gebunden. Es wird `TRUE` zurückgegeben, falls alle Ergebnisse `TRUE` zurückgeliefert hat.
Prädikat (einstellig)	Zunächst wird in der Liste der einstelligen Prädikatsymbole (`oneArySymbolTable`) die unter dem Namen der aktuellen `FOLFormula` gespeicherte Liste von `Vertex` Objekten herausgesucht. Anschließend wird `TRUE` zurückgegeben, falls sich in dieser Liste der `Vertex` befindet, welcher zu dem Zeitpunkt auch an Variable des Prädikats gebunden ist.
Prädikat (zweistellig)	Zunächst wird in der Liste der zweistelligen Prädikatsymbole (`twoArySymbolTable`) die unter dem Namen der aktuellen `FOLFormula` gespeicherte Liste von `Edge` Objekten herausgesucht. Anschließend wird `TRUE` zurückgegeben, falls sich in dieser Liste ein `Edge` Objekt befindet, dessen Start- und Ziel-`Vertex` identisch ist mit denen, welche zu diesem Zeitpunkt an die erste bzw. zweite Variable des Prädikats gebunden sind.

4.3 Beweismodus

Der Beweismodus dient als didaktisches Mittel, um Lernenden spielerisch bei dem Verständnis vom Model Checking zu helfen. Dieser basiert auf der Spielsemantik, beschrieben von Martin Otto [Ott]. Dabei gibt es zwei Spieler, den Verifizierer, welcher gewinnt, wenn die Struktur Modell der Formel ist, und den Falsifizierer, welcher gewinnt, wenn die Struktur kein Modell der Formel ist, sowie eine Struktur $\mathcal{A}$ und eine Formel φ, welche bespielt werden. Das Spiel beginnt mit der ersten Subformel ψ von φ in Negationsnormalform. Je

nach Typ darf der Verifizierer oder Falsifizierer eine dieser Aktionen ausführen:

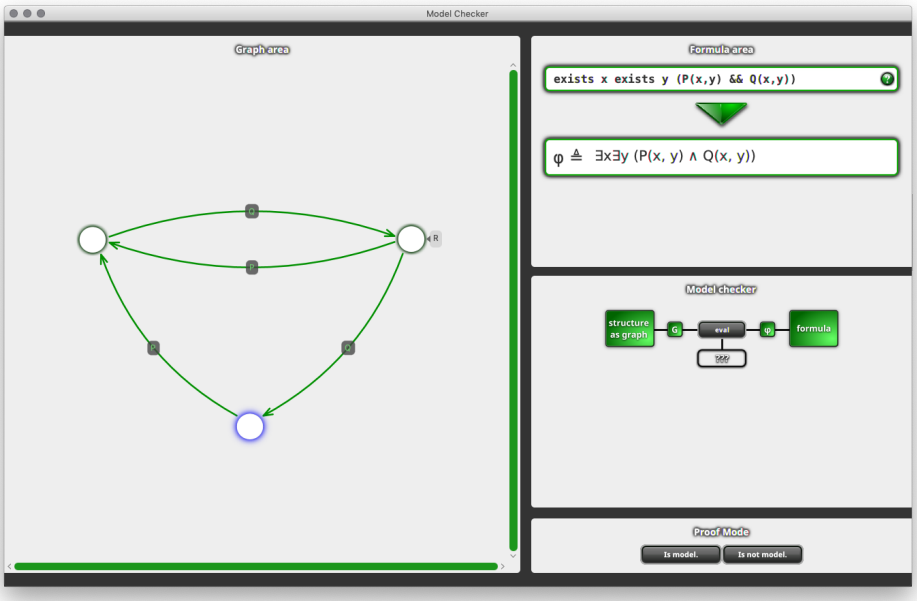
$\psi = \psi_1 \wedge \psi_2$ Falsifizierer wählt ψ_1 oder ψ_2
$\psi = \psi_1 \vee \psi_2$ Verifizierer wählt ψ_1 oder ψ_2
$\psi = \forall x_i \psi_0$ Falsifizierer wählt für x_i einen Knoten aus der Struktur $\mathcal{A}$
$\psi = \exists x_i \psi_0$ Verifizierer wählt für x_i einen Knoten aus der Struktur $\mathcal{A}$

Danach wird die nächste Subformel bespielt, solang bis das Spiel mit einer atomaren oder negiert atomaren Subformel endet.

Anders als in der Spielsemantik muss die Formel φ im Beweismodus nicht in Negationsnormalform gebracht werden, da eine umgeformte Formel den Nutzer verwirren und zu sehr vom Hauptproblem ablenken könnte. Um dies zu ermöglichen, werden im Falle eine Negation die Rollen von Verifizierer und Falsifizierer vertauscht. Ab einem solchen Zeitpunkt muss beispielsweise der Nutzer, welcher zuvor die Rolle des Verifizierers inne hatte, die des Falsifizieres übernehmen und seine Entscheidungen so setzen, dass die Struktur kein Modell des folgenden negierten Formelbaums ist, um seine anfängliche Rolle als Verifizierer zu bestätigen.

Um den Beweismodus nahtlos in das bestehende Programm zu integrieren wurden unterhalb des Model Checkers zwei Buttons integriert, mithilfe derer der Nutzer, nach Eingabe einer Formel und Struktur, den Beweismodus als Verifizierer (Struktur ist Modell) oder als Falsifizierer (Struktur ist kein Modell) starten kann.

Abbildung 4.4: Model Checker mit zusätzlichem Beweismodus

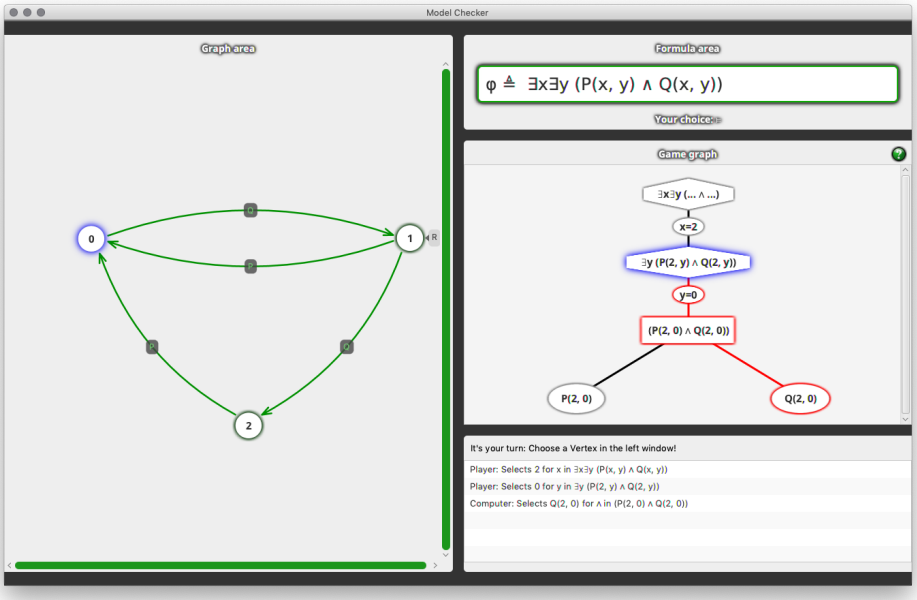


Der Computer übernimmt daraufhin die Rolle des Gegenspielers und probiert den Nutzer vom Gegenteil der gewählten Antwort zu überzeugen. Dafür wird beim Spielbeginn intern

ein vollständiger Spielbaum erzeugt. Für jeden Quantor wird dabei für jeden Knoten in der Struktur ein Ast generiert und für jeden Operator alle Seiten bis zum Ende generiert, sodass der Computer für jeden Schritt im Spielgraph alle Möglichen Endszenarien kennt und seinen Schritt dementsprechend wählen kann. Im Falle eines, für den Computer, aussichtslosen Spiels wählt dieser zufällig.

Abbildung 4.5 zeigt ein laufendes Spiel mit ausstehender Nutzerentscheidung.

Abbildung 4.5: Beweismodus mit ausstehender Nutzerentscheidung

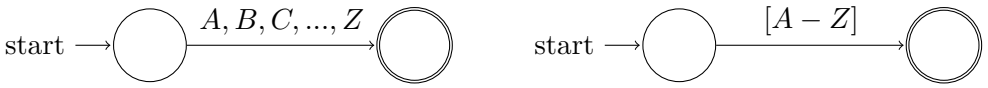


4.4 Automaten Bibliothek

Um das Model Checking von automatischen Strukturen möglich zu machen, muss zunächst die Möglichkeit geschaffen werden automatische Strukturen mithilfe von mehrspurigen Automaten zu implementieren. Als Basis wird dafür die „Symbolic Automata“ Bibliothek von Loris D'Antoni [D'A] eingesetzt. Diese Bibliothek ist unter anderem in der Lage auf sogenannten (einstelligen) SFAs (Symbolic finite automaton) die Standard Operationen, Vereinigung, Schnitt und Komplement durchzuführen. Als Alphabet für diese Automaten wird dabei standardmäßig der Unicode Zeichensatz (U+0000 bis U+FFFF) verwendet. Für eine einfachere Verständlichkeit wird dieses aber in einigen nachfolgenden Beispielen eingeschränkt. Ein wichtiger und, bei dieser Alphabet größe notwendiger, Schritt ist das Nutzen von Intervallen innerhalb der Zustandsübergänge (Transitionen). Dadurch ist es beispielsweise möglich, eine Transition mit der Bedingung „[A-Z]“ zu definieren, ohne alle Symbole einzeln aufzählen zu müssen. Grafisch ist dies wie in Abbildung 4.6 vorstellbar. Diese Designentscheidung kommt beim Anwenden von Operationen vor allem der Speichernutzung und Laufzeit zugute, welche im Fall von mehrspurigen Automaten noch viel

wichtiger ist.

Abbildung 4.6: Nutzung von Intervallen innerhalb von Transitionen



Im Folgenden werden die Algorithmen und Implementierungen der Bibliothek beschrieben, um anschließend die notwendigen Anpassungen für mehrspurige Automaten aufzuzeigen und deren Implementierung sowie Probleme währenddessen zu erläutern. Implementationsdetails werden dabei mittels Pseudocode oder Ablauf-Diagrammen anschaulich dargestellt. Ein gewisses Grundwissen über Datenstrukturen und Datentypen aus Java wird vorausgesetzt.

Um einen Automaten in der Implementierung abzubilden, wird dieser SFA als Klasse definiert. Diese Klasse besitzt einen initialen Zustand, eine Liste von Endzuständen, eine Liste von Transitionen und eine boolesche Algebra. Zustände haben dabei keinen eigenen Typ, sondern werden durch einen Integer Wert mit einer ID versehen und entsprechend hochgezählt. Anders als in der Definition eines Automaten, wird die Menge der Zustände nicht explizit angegeben, sondern kann aus den gegebenen Zuständen und der Liste der Transitionen gebildet werden. Das Eingabealphabet wiederum wird durch die Implementierung einer boolesche Algebra gegeben, in diesem Fall der Unicode Zeichensatz.

Eine Transition (SFAInputMove) wird auch als eigenständiges Objekt angesehen. Dieses hat neben einem Startzustand und einem Endzustand eine Bedingung, im Folgenden `guard` genannt, welche erfüllt werden muss, damit diese Transition durchgeführt werden kann. Werden die ausgehenden Transitionen eines Zustands benötigt, können diese mit der ID des Zustands über den Automaten abgefragt werden. Dieser iteriert wiederum durch die Liste der Transitionen und liefert die entsprechenden Transitionen zurück.
Als guard kommt in diesem Fall ein sogenanntes Character Predicate (CharPred) zum Einsatz, welches die Logik der Intervalle definiert. Dieses besitzt eine Liste von Paaren, welche jeweils die Grenzen eines Intervalls definieren. Beispielsweise '1' bis '2', '4' bis '4' und '7' bis '9'. Neben dieser Liste beinhaltet die Implementierung auch noch die Logik zur Validierung (beispielsweise ist ein `CharPred` mit den Intervallen (9,0) unzulässig, da die '0' in der Unicode Definition vor der '9' kommt) und Auswertung ob z.B. ein `CharPred` mit dem Intervall (0,5) von einem Character '7' erfüllt wird.

Die Implementierung einer booleschen Algebra stellt einen wichtigen Punkt in dem Automatenkonstrukt dar. In dem Basisfall kommt ein `UnaryCharIntervalSolver` zum Einsatz, welcher speziell für den Einsatz mit intervall-basierten, einspurigen `CharPred` implementiert wurde. Die Algebra hat dabei die Aufgabe bekannte Logikoperationen (And, Or, Not) für entsprechende guards der Transitionen bereitzustellen.

Die Idee des Algorithmus für das Komplement eines `CharPreds` besteht darin, die Lücken zwischen den Intervallen zu schließen. Durch Betrachten des `CharPred` mit den Intervallen (2,3) und (6,7) ist ersichtlich dass das Komplement mindestens das Intervall (4,5) beinhalten muss. Zusätzlich die Intervalle vom kleinsten Element des Alphabets (0,1) sowie bis zum größten Element des Alphabets (8,9) . Listing 4.5 zeigt den Algorithmus im Pseudocode.

Listing 4.5: Operation Not - Komplement eines CharPreds

```
1 func not(ncp) -> CharPred {
2     List resultIntervals = new List
3
4     // Step 1.
5     // If the first interval doesn't start at the minimum,
6     // add the corresponding interval.
7     Character currentBottom = ncp.intervals.first.left;
8     if (CharPred.MIN_CHAR < currentBottom) {
9         newIntervals.add(Pair.of(
10            CharPred.MIN_CHAR,
11            (currentBottom - 1)
12        ));
13    }
14    // Step 2.
15    // Add the spaces between each interval.
16    for interval in ncp.intervals from 1 to ncp.intervals.size {
17        currentBottom = interval.left;
18        char newLow = previousTop + 1;
19        char newHigh = currentBottom - 1;
20        if (newLow <= newHigh) {
21            newIntervals.add(Pair.of(
22                newLow,
23                newHigh
24            ));
25        }
26        previousTop = interval.right;
27    }
28    // Step 3.
29    // If previousTop doesn't end at the maximum,
30    // add the corresponding interval.
31    if (previousTop < CharPred.MAX_CHAR) {
32        newIntervals.add(Pair.of(
33            (previousTop + 1),
34            CharPred.MAX_CHAR
35        ));
36    }
37
38    return CharPred(resultIntervals)
39 }
```

Der Schnitt zweier `CharPreds` wird mithilfe der mathematischen „Min“ und „Max“ Funktionen realisiert. Es wird das Maximum der beiden unteren (linken) Grenzen und das Minimum der beiden oberen (rechten) Grenzen gebildet. Ein valides Schnittintervall liegt vor, falls der errechnete untere Wert kleiner oder gleich dem errechneten oberen Wert ist. Ist der errechnete untere Wert größer als der errechnete oberen Wert, ist der Schnitt leer. Das Schnittintervall von den Intervallen $1-3$ mit $2-4$ ist demnach $2-3$. Listing 4.6 zeigt den Algorithmus im Pseudocode.

Listing 4.6: Operation And - Schnitt zweier CharPreds

```
1 func and(ncp1, ncp2) -> CharPred {
2     List result = new List
3
4     for interval1 in ncp1 {
5         for interval2 in ncp2 {
6
7             // 1. Step
8             // Create new Pair
9             pair = Pair.of(
10                Math.max(interval1.left,  interval2.left),
11                Math.min(interval1.right, interval2.right)
12            );
13
14            // 2. Step
15            // If the result is valid, add new interval element.
16            // It's not valid if:
17            // * There's a pair with the wrong order (e.g  '6-4')
18            // * List 'result' contains the new element.
19            if pair.valid() {
20                result.add(pair);
21            }
22        }
23    }
24    return CharPred(result)
25 }
```

Beispiel 4.4.1. *Gegeben sind die beiden CharPreds ncp1 und ncp2 mit:*
$ncp1 = [(0,3),(6,9)]$
$ncp2 = [(0,2),(5,8)]$
Gesucht ist der Schnitt beider CharPreds

Es wird schrittweise Algorithmus 4.6 angewendet:

1. Iterationsschritt:	2. Iterationsschritt:	3. Iterationsschritt:	4. Iterationsschritt:
$interval1 = (0,3)$	$interval1 = (0,3)$	$interval1 = (6,9)$	$interval1 = (6,9)$
$interval2 = (0,2)$	$interval2 = (5,8)$	$interval2 = (0,2)$	$interval2 = (5,8)$
$pair = (0,2)$	$pair = (5,3)$	$pair = (6,2)$	$pair = (6,8)$
$result = [(0,2)]$	$result = [(0,2)]$	$result = [(0,2)]$	$result = [(0,2),(6,8)]$

Der Schnitt von ncp1 und ncp2 ist ein CharPred mit den Intervallen $[(0,2),(6,8)]$.

Die Logikoperationen für Or wird durch die bekannte Operation $!(!ncp1 \wedge !ncp2!)$ realisiert und bedarf damit keinem zusätzlichen Algorithmus.

Nachdem nun bekannt ist wie ein Automat implementiert wird und wie Logik Operationen auf guards funktionieren, wird im Folgenden die Implementierungen der Operationen auf Automaten aufgezeigt. Die Bibliothek ist in der Lage das Komplement eines Automaten sowie den Schnitt und die Vereinigung zweier Automaten zu bilden. Die Implementierung folgt dabei den bekannten Verfahren und Algorithmen. Abbildung 4.7 zeigt den Ablauf um das Komplement eines Automaten zu bilden, Abbildung 4.8 zeigt den Ablauf um den Schnitt zweier Automaten zu bilden und Abbildung 4.9 zeigt den Ablauf um die Vereinigung zweier Automaten zu bilden.

Abbildung 4.7: Implementierung des Komplements von einem Automaten A.

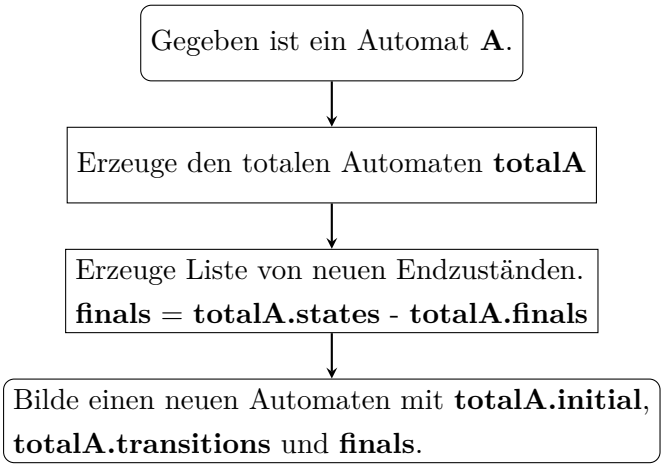


Abbildung 4.8: Implementierung des Schnitts von zwei Automaten A1 und A2.

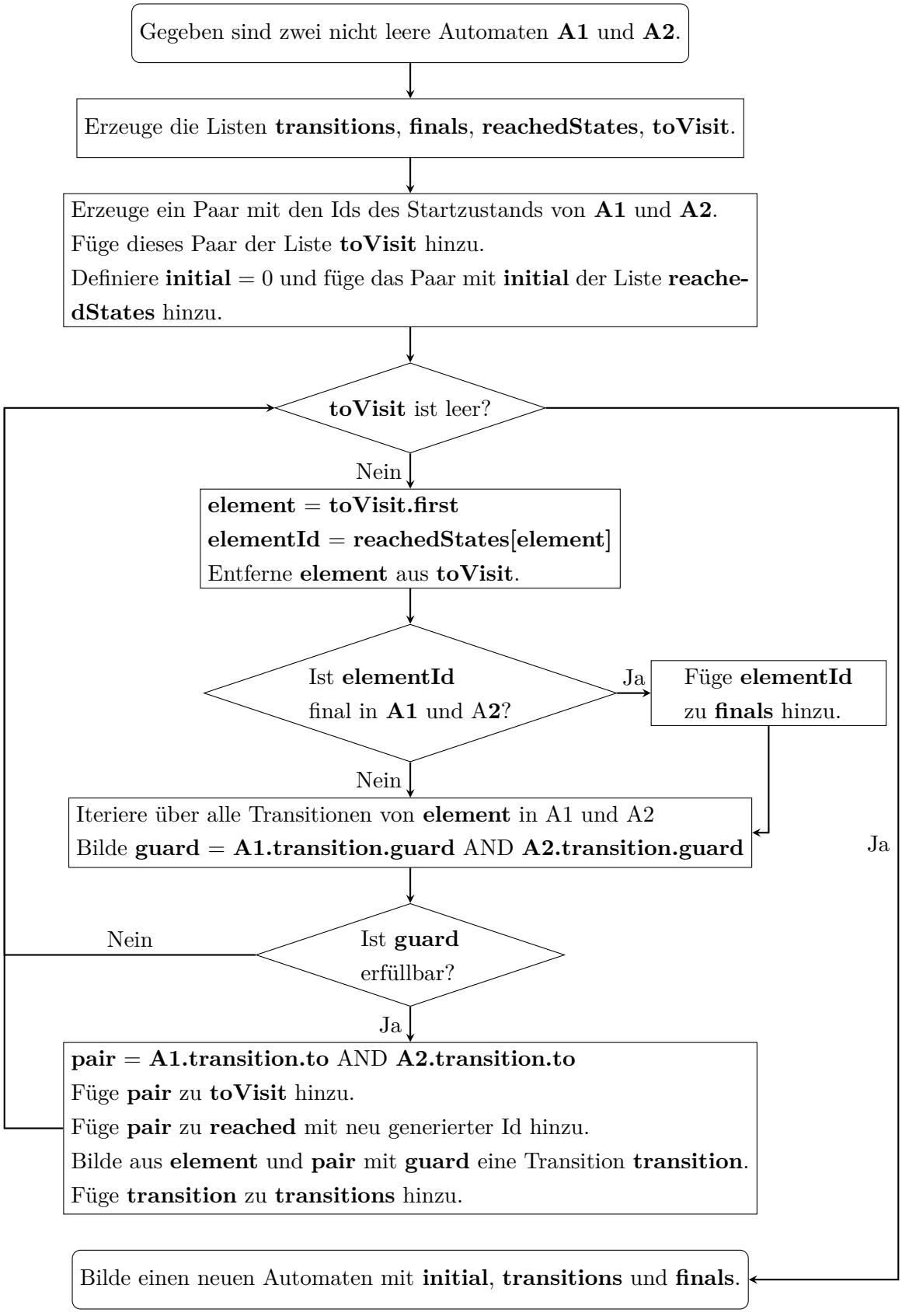
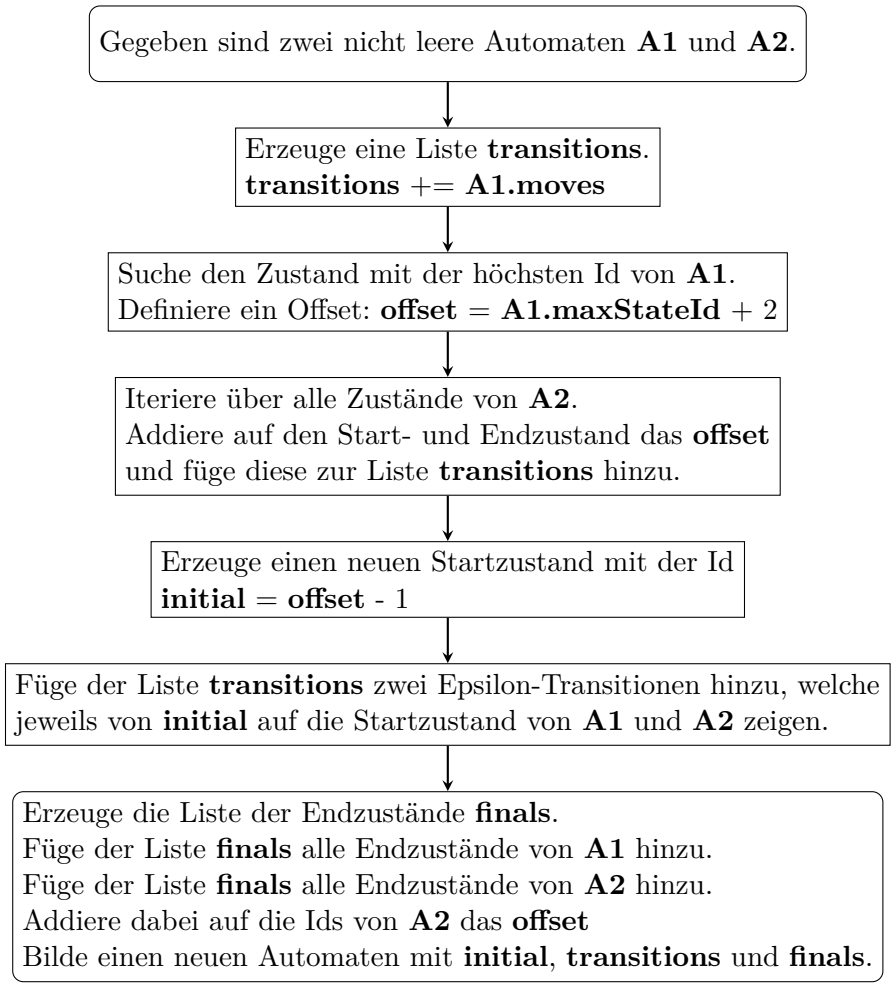


Abbildung 4.9: Implementierung der Vereinigung von zwei Automaten A1 und A2.



4.5 Erweiterung der Bibliothek

Da nun bekannt ist wie ein Automat und dessen Operationen implementiert sind, kann begonnen werden einen n-Spuren Automat zu definieren, um Komponenten zu finden, welche erweitert werden müssen. Der erste Schritt besteht darin, eine Liste von Transitionen zu erstellen. Wie im vorherigen Kapitel 4.4 beschrieben wird dafür ein Ausgangszustand, ein Folgezustand und in diesem Fall ein n-spuriges guard benötigt. Da das bestehende `CharPred` nur eine Spur abbilden kann, musste die Bibliothek um ein guard Implementierung erweitert werden, welches n Spuren abbilden kann. Im Folgenden `NCharPred` genannt.

Um zu verstehen, wie die Struktur des `NCharPred` definiert werden muss, wird mit einem einspurigen guard mit den Intervallen '0' - '2' und '5' - '6' begonnen. Die Elemente des guards sind demnach 0,1,2,5 und 6. Dieses guard soll nun so erweitert werden, dass es die Elemente $(0,1)$, $(0,2)$, $(0,3)$, $(1,1)$, $(1,2)$, $(1,3)$, $(2,1)$, $(2,2)$, $(2,3)$, $(5,4)$ und $(6,4)$ beinhaltet. Dafür wird für das erste Intervall '0' - '2' eine zweite Spur mit '1' - '3', sowie für das zweite Intervall '5' - '6' eine Spur mit '4' - '4' hinzugefügt. Damit die Datenstruktur

dies abbilden kann, wird eine Liste von Intervallen benötigt, welche für jedes Intervall wiederum eine Liste von Spuren mit den jeweiligen Paaren beinhaltet, wie in Abbildung 4.7 zu sehen.

Listing 4.7: Intervall Struktur eines NCharPred mit zwei Spuren

```
1 // Liste von Intervalllisten
2 [
3     // Liste von n Paaren des ersten Intervalls
4     [
5         (0,2),(1,3)
6     ]
7     // Liste von n Paaren des zweiten Intervalls
8     [
9         (5,6),(4,4)
10    ]
11 ]
```

Die Intervalle der jeweiligen Spuren gehören dabei zwingend zueinander. Betrachtet an Beispiel 4.7, bedeutet dies, dass das erst Element im ersten Intervall $(0, 2)$ zu dem zweiten Element im ersten Intervall $(1, 3)$ gehört und damit folgende Symbole eines 2-spurigen Wortes definiert:

$$\left(\begin{array}{c}0\\1\end{array}\right),\left(\begin{array}{c}0\\2\end{array}\right),\left(\begin{array}{c}0\\3\end{array}\right),\left(\begin{array}{c}1\\1\end{array}\right),\left(\begin{array}{c}1\\2\end{array}\right),\left(\begin{array}{c}1\\3\end{array}\right),\left(\begin{array}{c}2\\1\end{array}\right),\left(\begin{array}{c}2\\2\end{array}\right),\left(\begin{array}{c}2\\3\end{array}\right)$$

Das zweite Intervall analog.

Daraus folgt, dass die Anzahl der Paare innerhalb eines Intervalle immer identisch sein müssen, da diese die Anzahl der Spuren definieren.

Die Implementierung der Transition (SFAInputMove) bedient sich Generic Types, dadurch kann das neu implementierte `NCharPred` direkt genutzt werden, ohne dass die Transition für n-spurige Automaten neu implementiert werden muss.

Da die im Basisfall genutzte Boolesche Algebra `UnaryCharIntervalSolver` auch speziell für die Nutzung mit `CharPreds` implementiert wurde, muss die Bibliothek um eine Algebra erweitert werden, welche mit `NCharPreds` umgehen kann. Im folgenden `NCharIntervalSolver` genannt. Diese muss in der Lage sein, die Standardoperationen „And“, „Not“ und „Or“ auf `NCharPreds` durchzuführen. Ein naiver Ansatz wäre, einfach die im Intervall definierten guards vollständig als Liste zu generieren und auf diesen dann die Operationen mithilfe der Java Standard Funktionen auf Listen durchzuführen (z.B. könnte für „Not“ eine vollständige Liste unter dem gegebenen Alphabet generiert und dann die entsprechenden Elemente entfernt werden), um anschließend aus den verbleibenden Elementen die neuen Intervalle zu generieren. Dieser Ansatz ist zwar weniger

fehleranfällig, aber wirkt sich drastisch auf die Performance aus. Aus diesem Grund wird für jede Operation ein spezieller Algorithmus benötigt, welcher sich die Datenstruktur des `NCharPreds` zunutze macht.

Der Algorithmus für das Schneiden zweier `NCharPreds` orientiert sich sehr stark an dem für `CharPreds`, welcher in Listing 4.6 beschrieben ist. Da `NCharPred` mehrere Spuren besitzen, müssen diese dabei beachtet werden, weshalb der Algorithmus beim Berechnen der jeweiligen Schnittintervalle zusätzlich über alle Spuren iteriert, wie in Listing 4.8 dargestellt.

Listing 4.8: Operation And - Schnitt zweier NCharPreds

```
1 func and(ncp1, ncp2) -> NCharPred {
2     // Lanes count if ncp1 and ncp2 are equal
3     Int lanes = ncp1.lanes
4     List resultIntervals = new List
5
6     // Iterate over the intervals
7     for i1 in ncp1 {
8         for i2 in ncp2 {
9
10            Liste lanes = new List
11
12            // 1. Step
13            // Create new Pairs for each lane
14            for i in lanes {
15                lanes.add(Pair.of(
16                    Math.max(i1[i].left,  i2[i].left),
17                    Math.min(i1[i].right, i2[i].right)
18                ));
19            }
20
21            // 2. Step
22            // If the result is valid, add new interval element.
23            // It's not valid if:
24            // * There's a pair with the wrong order (e.g  '6-4')
25            // * List 'resultIntervals' contains the new element.
26            if lanes.valid() {
27                resultIntervals.add(lanes);
28            }
29        }
30    }
31    return NCharPred(resultIntervals)
32 }
```

Beispiel 4.5.1 zeigt die schrittweise Durchführung des Algorithmus mit den `NCharPreds` $ncp1 = \begin{pmatrix} 1-2 \\ 1-3 \end{pmatrix}, \begin{pmatrix} 4-5 \\ 4-6 \end{pmatrix}$ und $ncp2 = \begin{pmatrix} 2-3 \\ 2-4 \end{pmatrix}, \begin{pmatrix} 5-6 \\ 5-7 \end{pmatrix}$.

Beispiel 4.5.1. *Gegeben sind die beiden 2-Spurigen NCharPreds ncp1 und ncp2 mit:*
$ncp1 = [[(1,2),(1,3)],[(4,5),(4,6)]]$
$ncp2 = [[(2,3),(2,4)],[(5,6),(5,7)]]$
Gesucht ist der Schnitt beider NCharPreds.

Es wird schrittweise Algorithmus 4.8 angewendet:

1. Iterationsschritt:	2. Iterationsschritt:
$i1 = [(1,2),(1,3)]$	$i1 = [(1,2),(1,3)]$
$i1 = [(2,3),(2,4)]$	$i2 = [(5,6),(5,7)]$
$lanes = [(2,2),(2,3)]$	$lanes = [(5,2),(5,3)]$
$result = [[(2,2),(2,3)]]$	$result = [[(2,2),(2,3)]]$
3. Iterationsschritt:	**4. Iterationsschritt:**
$i1 = [(4,5),(4,6)]$	$i1 = [(4,5),(4,6)]$
$i2 = [(2,3),(2,4)]$	$i2 = [(5,6),(5,7)]$
$lanes = [(4,3),(4,4)]$	$lanes = [(5,5),(5,6)]$
$result = [[(2,2),(2,3)]]$	$result = [[(2,2),(2,3)],[(5,5),(5,6)]]$

Der Schnitt von den beiden NCharPreds ncp1 und ncp2 ist ein NCharPred mit den Intervallen $[(2,2),(2,3)],[(5,5),(5,6)]]$.

Für das Bilden des Komplements eines `NCharPred` kann leider nicht der bestehende Algorithmus des `CharPred` genutzt werden, da das Iterieren über alle Spuren zu falschen Ergebnissen führen würde. Aus diesem Grund wurde ein neuer Algorithmus entwickelt. Die Idee dabei war es, mit einem neuen, unter dem Alphabet vollständigen Intervall zu beginnen, welches dieselbe Spurenanzahl besitzt wie das `NCharPred` $ncp1$, für welches die Operation durchgeführt werden soll. Anschließend wird über alle Intervalle von $ncp1$ iteriert und es werden Spurweise diese Bereiche aus dem gerade erstellen Intervall entfernt. Algorithmus 4.9 zeigt die Implementierung dieser Operation.

Listing 4.9: Operation Not - Komplement eines NCharPred

```
1 func not(ncp1) -> NCharPred {
2     Int lanes = ncp1.lanes
3     List resultIntervals = new List
4
5     // Add initial min-max pairs for each lane
6     List initial = new List
7     for i in 0...lanes {
8         initial.add(Pair.of(MIN, MAX));
9     }
10    resultIntervals.add(initial);
11
12    // Call the function to remove all ncp1 intervals
13    // from our min-max interval
14    resultIntervals = not(resultIntervals, ncp1.intervals)
15
16    return NCharPred(resultIntervals)
17 }
18
19 // This function is called recursively,
20 // to remove each interval from 'sourceIntervals'
21 func recursiveNot(resultIntervals,
22                   sourceIntervals) -> IntervalList {
23     if !sourceIntervals.empty {
24         // Get the first interval from list to remove
25         toRemove = sourceIntervals[0]
26
27         // Get a list of intervals which need to be
28         // modified to remove 'intervalToRemove'
29         // These are intervals which intersect with 'toRemove'
30         intervalsToModify = getIntersections(resultIntervals,
31                                              toRemove)
32
33         for interval in intervalsToModify {
34             result = removeInterval(toRemove,
35                                     interval,
36                                     new List,
37                                     0)
38             // Remove the intersecting interval
39             // and replace it with the intervals
40             // without 'toRemove'
41             resultIntervals.remove(interval)
42             resultIntervals.addAll(result)
43         }
44
45         sourceIntervalls.remove(toRemove)
46
47         // Recursive call
```

```
48          return recursiveNot(resultIntervals, sourceIntervalls)
49      } else {
50          return resultIntervals
51      }
52 }
53
54 // This function is called to remove an interval from
55 // a given intersectingInterval.
56 // It's called recursively for each lane.
57 func removeInterval(removeInterval,
58                     intersectingInterval,
59                     resultIntervals,
60                     laneIndex) -> IntervalList {
61     Int lanes = intersectingInterval.lanes
62
63     left = intersection(removeInterval,
64                         intersectingInterval,
65                         .left)
66     resultIntervals.add(left)
67
68     right = intersection(removeInterval,
69                          intersectingInterval,
70                          .right)
71     resultIntervals.add(right)
72
73     middle = interval(left, right)
74     resultIntervals.add(middle)
75
76     if (laneIndex + 1) < lanes {
77         return removeInterval(removeInterval,
78                               middle,
79                               resultIntervals,
80                               (laneIndex + 1))
81     } else if middle == removeInterval {
82         resultIntervals.remove(removeInterval)
83         return resultIntervals
84     }
85 }
```

Beispiel 4.5.2. *Gegeben ist das 3-Spurige NCharPred ncp1 mit:*
$ncp1 = [[(2,3),(1,4),(7,9)]]$
Gesucht ist der Schnitt beider CharPreds

Es wird schrittweise Algorithmus 4.9 angewendet:

Begonnen wird mit einem vollständigen 3-spurigen Intervall: $[(0,9),(0,9),(0,9)]$

*Dieses wird neben dem Intervall von ncp*1 *in die Funktion recursiveNot gegeben.*
Das Intervall $[(0,9),(0,9),(0,9)]$ *schneidet sich mit* $[(2,3),(1,4),(7,9)]$ *und wird mit dem Spurindex 1 in die Funktion removeInterval gegeben.*

Es wird $(0-9)-(2-3)=(0-1),(4-9)$ *berechnet, wodurch das ursprüngliche Intervall* $[(0,9),(0,9),(0,9)]$ *in den linken und rechten Teil* $[(0,1),(0,9),(0,9)]$ *und* $[(4,9),(0,9),(0,9)]$*, sowie den mittleren Teil* $[(2,3),(0,9),(0,9)]$ *aufgeteilt wird.*
In der Ergebnisliste wird $[(0,9),(0,9),(0,9)]$ *gegen die aufgeteilten Intervalle* $[(0,1),(0,9),(0,9)],[(2,3),(0,9),(0,9)],[(4,9),(0,9),(0,9)]$ *ausgetauscht. Der mittlere Teil wird samt der Ergebnisliste in die Funktion removeInterval gegeben, wobei der Spurindex auf 2 angehoben wird.*

Es wird $(0-9)-(1-4)=(0-0),(5-9)$ *berechnet, wodurch das ursprüngliche Intervall* $[(2,3),(0,9),(0,9)]$ *in den linken und rechten Teil* $[(2,3),(0,0),(0,9)]$ *und* $[(2,3),(5,9),(0,9)]$*, sowie den mittleren Teil* $[(2,3),(1,4),(0,9)]$ *aufgeteilt wird.*
In der Ergebnisliste wird $[[(2,3),(0,9),(0,9)]$ *gegen die aufgeteilten Intervalle* $[(2,3),(0,0),(0,9)],[(2,3),(1,4),(0,9)],[(2,3),(5,9),(0,9)]$ *ausgetauscht. Der mittlere Teil wird samt der Ergebnisliste in die Funktion removeInterval gegeben, wobei der Spurindex auf 3 angehoben wird.*

Es wird $(0-9)-(7-9)=(0-6)$ *berechnet, wodurch das ursprüngliche Intervall* $[(2,3),(1,4),(0,9)]$ *in den linken Teil* $[(2,3),(1,4),(0,6)]$*, sowie den (theoretischen) mittleren Teil* $[(2,3),(1,4),(7,9)]$ *aufgeteilt wird.*
In der Ergebnisliste wird $[(2,3),(1,4),(0,9)]$ *gegen die aufgeteilten Intervalle* $[(2,3),(1,4),(0,6)],[(2,3),(1,4),(7,9)]$ *ausgetauscht. Da der Spurindex gleich der Anzahl der Spuren ist befindet sich der Algorithmus am Ende. Der zuletzt berechnete mittlere Teil gleich dem Intervall aus ncp*1 *und wird aus der Ergebnisliste entfernt.*

Die Ergebnisliste beinhaltet $[(0,1),(0,9),(0,9)],[(2,3),(0,0),(0,9)],[(2,3),(1,4),(0,6)]$, $[(2,3),(5,9),(0,9)],[(4,9),(0,9),(0,9)]$*, welche die Intervalle des Komplements von ncp*1 *sind.*

Da nun bekannt ist wie ein mehrspuriger Automat definieren werden kann und wie die Operation auf diesem funktionieren, wird nun wieder der Beispielautomaten P aus Abbildung 3.4 betrachtet. Das folgende Listing 4.10 zeigt die Implementierung mithilfe der Funktionalität der Bibliothek. Es werden vier Transitionen bestimmt, mit den jeweiligen „guards“. Die entsprechenden Zustände der Transitionen werden automatisch registriert. Abschließend wird der Startzustand 0, die Liste der Endzustände [2], die genutzte Implementierung der Algebra und eine Spurzuweisung definiert.

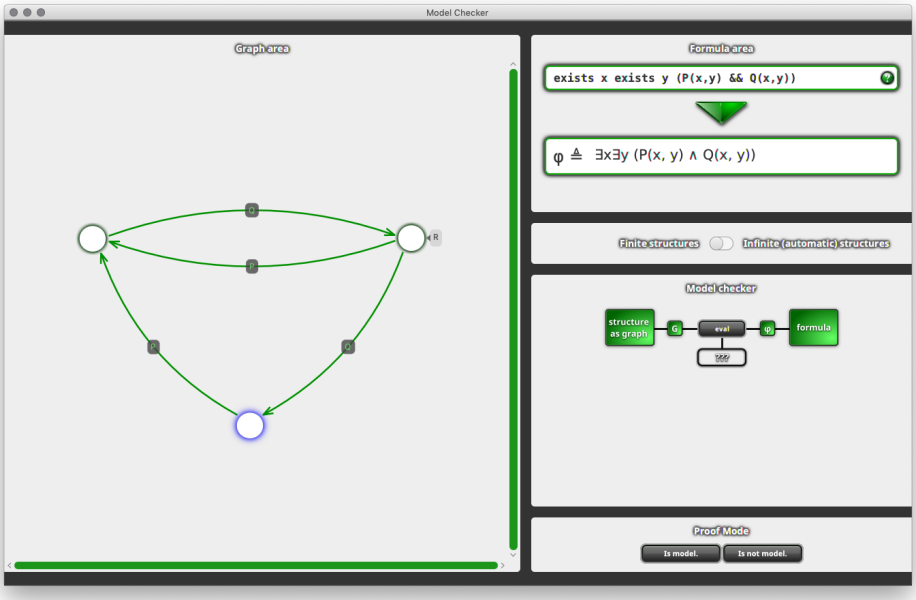
Listing 4.10: Zweispuriger Automat: P

```
1 // a-a
2 // b-b
3 private NCharPred ab = new NCharPred.Builder()
4         .element('a')
5         .element('b')
6         .build();
7
8 // a-b
9 // a-b
10 private NCharPred any = new NCharPred.Builder()
11         .element(ImmutablePair.of('a', 'b'), 2)
12         .build();
13
14 private NSFA<NCharPred, List<Character>>
15                 nsfa(List<String> laneAssignment) {
16
17     Collection<SFAMove<NCharPred, List<Character>>>
18         tra = new LinkedList<>();
19
20     tra.add(new SFAInputMove<>(0, 0, any));
21     tra.add(new SFAInputMove<>(0, 1, ab));
22     tra.add(new SFAInputMove<>(1, 2, ab));
23     tra.add(new SFAInputMove<>(2, 2, any));
24
25     NSFA<NCharPred, ArrayList<Character>> mkSFA
26         = NSFA.MkNSFA(tra,
27                       0,
28                       Arrays.asList(2),
29                       algebra,
30                       laneAssignment);
31
32         return mkSFA;
33 }
```

4.6 Model Checker für automatische Strukturen

Im vorherigen Kapitel wurde aufgezeigt wie die wesentlichen Funktionen innerhalb der Automaten Bibliothek implementiert wurden, um automatische Strukturen handhaben zu können. Dieses Kapitel beschäftigt sich mit der Integration der angepassten Automaten Bibliothek in die bestehende Implementation des Model Checkers und des Beweismodus.

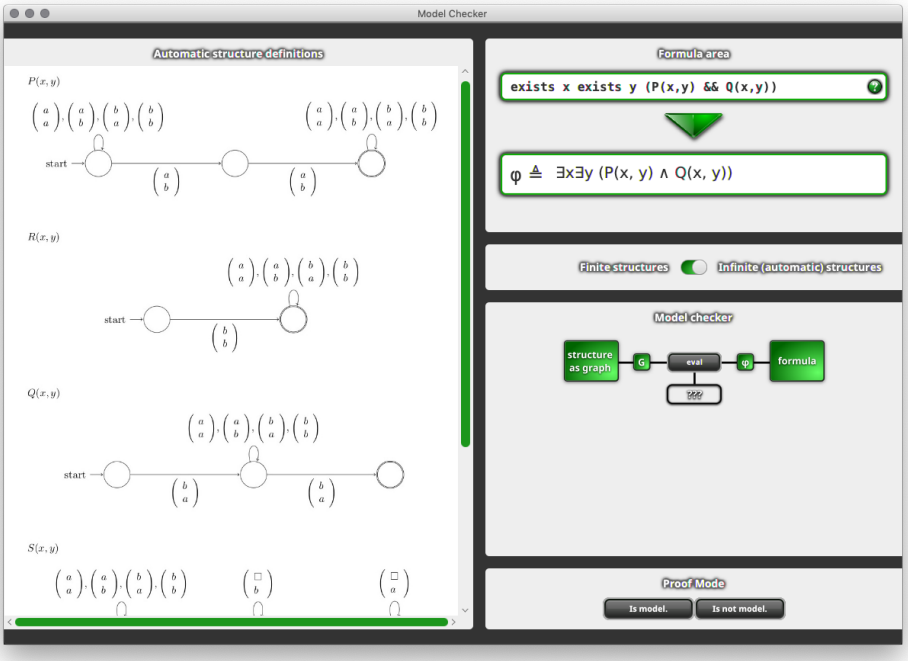
Abbildung 4.10: Model Checker mit Auswahl für endliche Strukturen



Zum einen musste die Benutzeroberfläche erweitert werden, um zu entscheiden, ob eine endliche oder unendliche Struktur genutzt werden soll, da die bestehende Implementierung weiterhin funktionieren soll. Dafür wurde ein Switch unterhalb der Formel integriert, wie in Abbildung 4.10 zu sehen. Durch Betätigung dieses Switches werden unendliche Strukturen ausgewählt. Zusätzlich wird auch die „Graph area“ durch eine Abbildung der implementierten automatischen Strukturen ausgetauscht, wie in Abbildung 4.11 zu sehen.

Die Implementierung des Model Checkers für automatische Strukturen unterscheidet sich im Aufbau nicht sehr stark von der für endliche Strukturen, welche in Kapitel 4.2.1 erläutert wurde. Da es sich nicht mehr um eine endliche Struktur handelt gibt es dementsprechend auch keine `Vertex-Edge` Datenstruktur, sondern eine Liste von mehrspurigen Automaten mit entsprechender Benennung für die jeweilige Relation, welche sie abbilden. Betätigt der Nutzer den Button für das Model Checking wird die selbe Setup-Routine, wie in Kapitel 4.2.1 beschrieben, durchlaufen. Anschließend beginnt auch dieser Algorithmus und arbeitet sich rekursiv, je nach Formeltype, durch die `FOLFormula`. Im Unterschied zum bestehenden Algorithmus für endlichen Strukturen wird dabei als Resultat kein `Boolean` zurückgegeben, sondern ein Automat. Wie sich der Algorithmus je nach Type verhält wird in nachfolgender Liste beschrieben.

Abbildung 4.11: Model Checker mit Auswahl für unendliche Strukturen



!	Die Subformel wird in den Algorithmus gegeben, der Ergebnis Automat jedoch negiert zurückgegeben.
||	Die beiden Subformeln werden in den Algorithmus gegeben, die Disjunktion der beiden Ergebnis Automaten wird zurückgegeben.
&&	Die beiden Subformeln werden in den Algorithmus gegeben, die Konjunktion der beiden Ergebnis Automaten wird zurückgegeben.
->	Die Formel `A -> B` wird als `!A || B` ausgewertet.
<->	Die Formel `A <-> B` wird als `(A && B) || (!A && !B)` ausgewertet.
exists	Die Subformel wird in den Algorithmus gegeben, der Ergebnis Automat wird zurückgegeben, aber vorher die gebundene Variable des Quantors herausprojiziert.
forall	Die Formel `forall A` wird als `!exists !A` ausgewertet.
Prädikat	Der Automat mit dem Namen des Prädikats wird zurückgegeben.

Um abschließend eine Aussage über ist Modell bzw. ist kein Modell treffen zu können kommt die Definition 3.2.1 zum Einsatz.

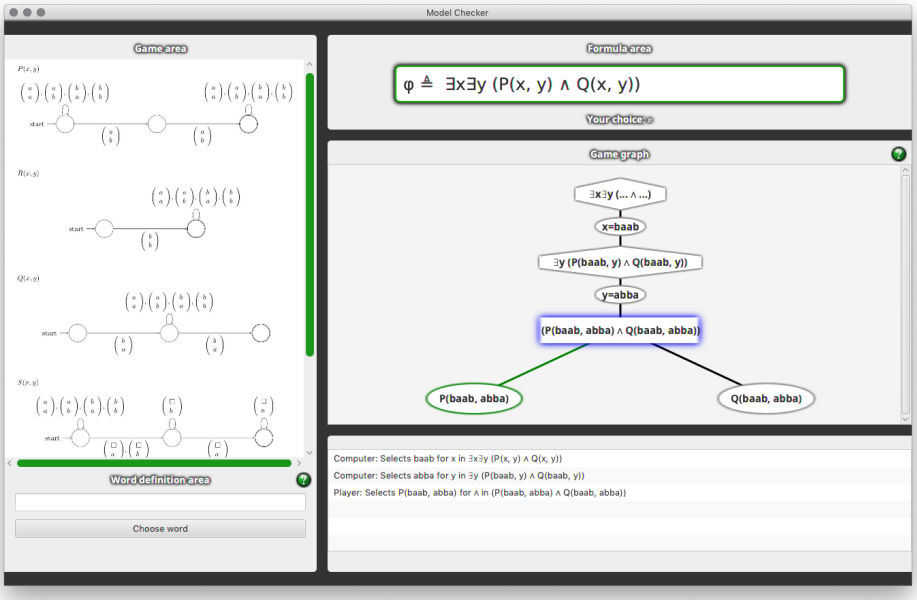
4.7 Beweismodus für automatische Strukturen

Die Implementierung des Beweismodus für automatische Strukturen basiert auch auf der bestehenden Logik, beschrieben in Kapitel 4.3. Wie im vorherigen Kapitel beschrieben,

gibt es auch hier keine `Vertex-Edge` Datenstruktur. Somit kann der Nutzer an einem Quantor nicht direkt einen Knoten in der Datenstruktur auswählen, sondern muss eine entsprechende Zeichenkette als Wort eingeben.

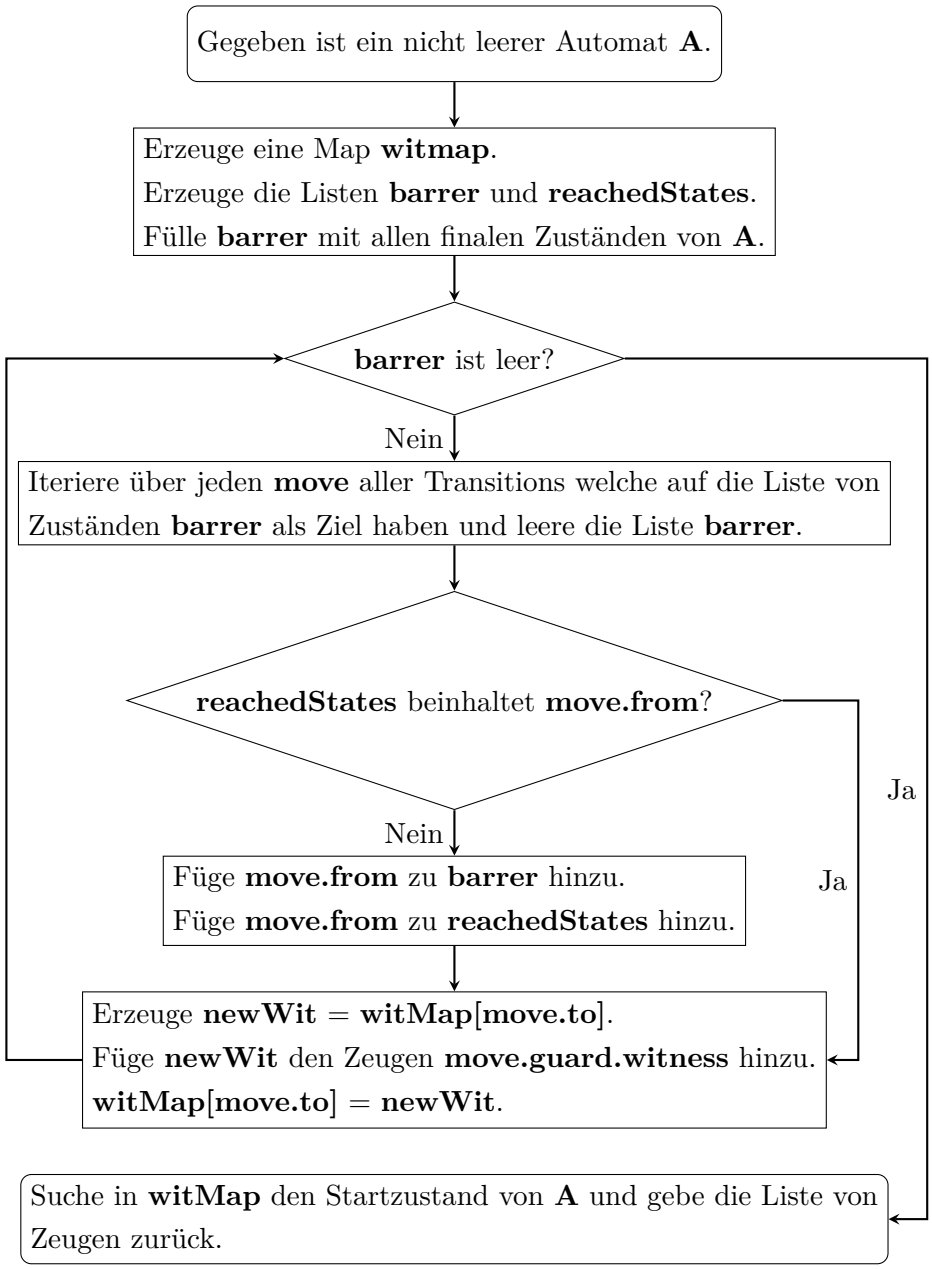
Dafür musste die Benutzeroberfläche für den Beweismodus bei unendlichen Strturen angepasst werden. Der linke Teil zeigt nun wie beim Model Checker auch die implementierten Automatischen Strukturen und beinhaltet zusätzlich noch ein Textfeld zum Eingeben eines Wortes, wie in Abbildung 4.12 ersichtlich.

Abbildung 4.12: Beweismodus für unendliche Strukturen mit ausstehender Nutzerentscheidung



Ein weiterer Unterschied ist das Spielverhalten des Computer-Gegners. Dieser kann in diesem Fall keinen vollständigen Spielgraphen nutzen, da ein solcher bei automatischen Strukturen unmöglich zu generieren ist. Aus diesem Grund generiert jeder Knoten bei seiner Auswahl dynamisch den oder die Folgeknoten. Damit der Computer aber weiterhin einen für seine Rolle korrekten Spielzug durchführen kann, nutzt dieser für die Auswahl von `&&` sowie `||` Operatoren die vorhandene Implementation des Model Checkers. Für Quantoren, bei welchen eine Zeichenkette aus dem Alphabet eingegeben werden muss, bedarf es einer Lösung welche eine Zeichenkette generiert, die zur Rolle des Computers passt.
Die Basis dafür bringt die Bibliothek bereits mit. Eine Methode, `getWitness`, welche eine einstellige Zeichenkette (im folgenden Zeuge genannt) generiert, welche den Automat erfüllt. Der Automat wird dabei vollständig vom Endzustand rückwärts durchlaufen und dabei die einzelnen Zeugen der Transitionen gespeichert. Am Ende wird die Liste an Zeugen zurückgegeben, welche am Startzustand beginnen. Abbildung 4.13 zeigt den Ablauf des Algorithmus im Detail.

Abbildung 4.13: Implementierung des Generieren eines Zeugen, welcher Automat A erfüllt.



Da die Implementierung dieser Methode bereits Generic Types nutzt, bedarf diese keiner Anpassung und kann direkt für mehrspurige Automaten verwendet werden. Jedoch eignet sich diese Methode nur, wenn der Computer eine Zeichenketten für die erste Quantor-Variable auswählen muss. Sollten bereits eine oder mehrere Quantor-Variablen Zeichenketten (ob vom Nutzer oder Computer) zugeordnet bekommen haben, müssen diese beim generieren eines Zeugen mit beachtet werden.

Der dafür entwickelte Algorithmus ist in Abbildung 4.14 sowie 4.15 abgebildet. Die Idee dahinter ist es, den Automaten mit dem bekannten Zeichenketten zu durchlaufen und alle anderen Spuren vorerst zu ignorieren. Dabei werden sich die jeweiligen Zustandsübergänge

gemerkt. Gibt es am Ende eine Liste von Zustandsübergängen, welche am Startzustand beginnt und am Endzustand endet, so kann diese durchlaufen werden und anhand des jeweiligen `guard` eine Liste von Zeugen generiert werden.

Abbildung 4.14: Suchen einer Liste von Transitionen für teilweise bekannte Eingabe.

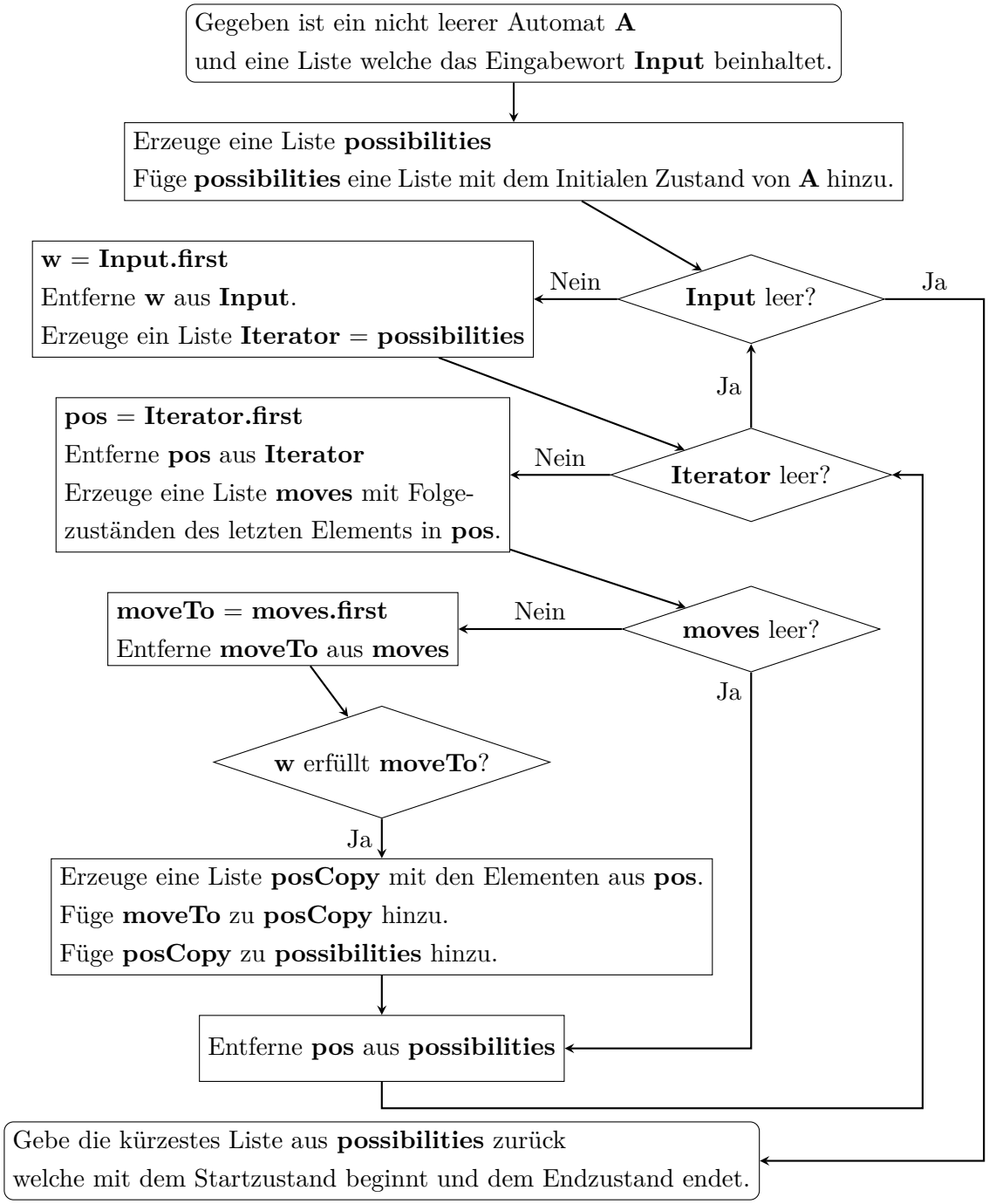
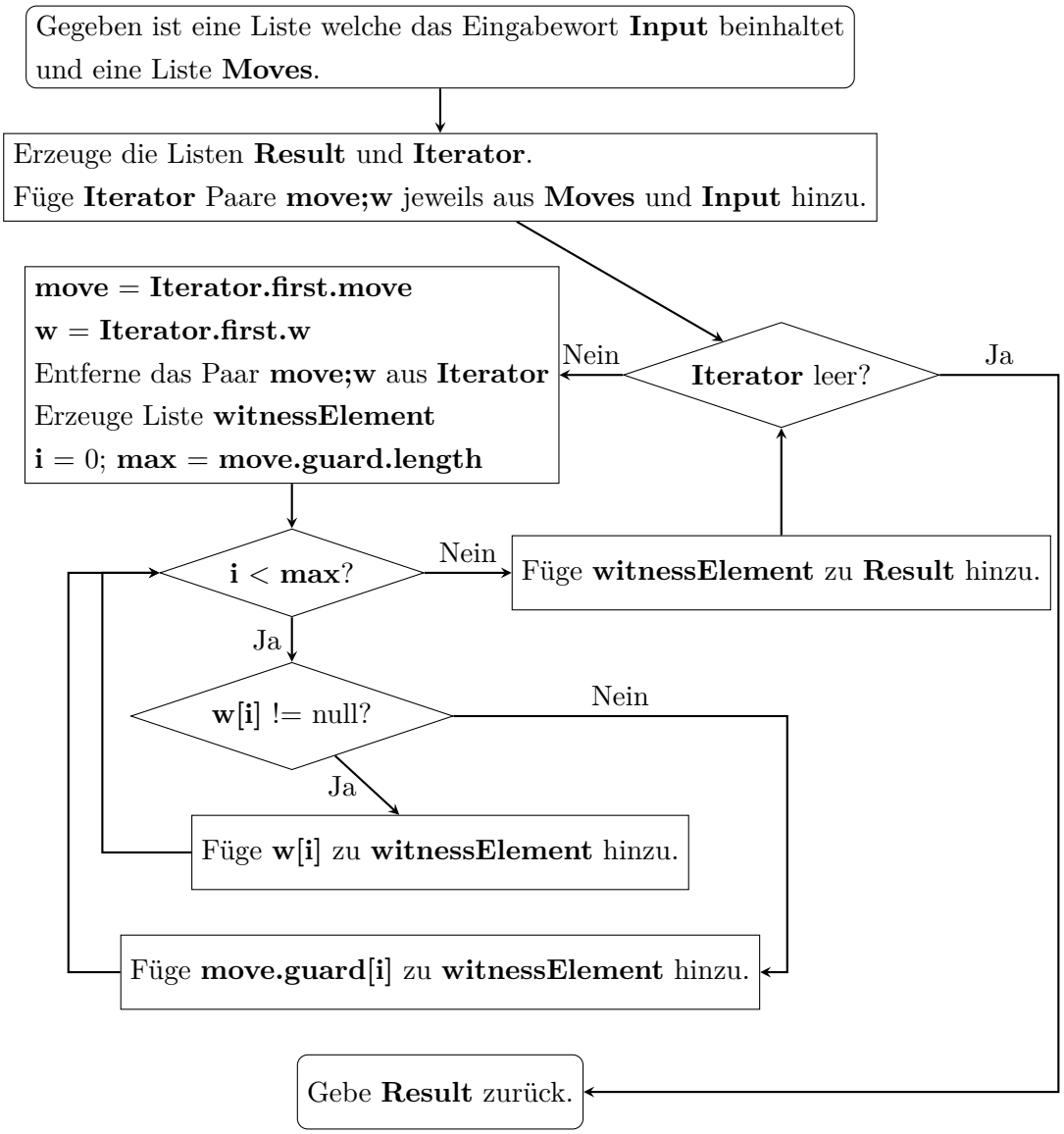


Abbildung 4.15: Generieren eines Zeugen für eine gegebene Liste von Zustandsübergängen und teilweise bekannte Eingabe.



Nachdem die Möglichkeit geschaffen wurde das Nutzer sowie Computer-Gegner Zeichenketten für die Quantoren eingeben und bereitstellen können wurde noch die Implementierung der Spiellogik angepasst. Durch die vorherige Implementierung gab es sehr starre Verknüpfungen zwischen der `Vertex-Edge` und der Implementierung der Spiellogik. Diese starren Abhängigkeiten wurden entfernt und durch Generics Types ersetzt.

Abschließend musste noch eine maximale Versuchsanzahl für Quantoren definiert werden. Ohne diese würde ein Beweis sonst unter bestimmten Umständen niemals enden. Beispielsweise beim Beweisen eines Existenz-Quantor einer Formel für welche die Struktur jedoch kein Modell ist.

5 Fazit und Ausblick

5.1 Fazit

Abschließend kann gesagt werden, dass die Integration von automatischen Strukturen in die bestehende Implementierung des Model Checkers sowie Beweismodus gut möglich war. Der wichtigste Teil dabei war, das Entwickeln der Definitionen und Abläufe von synchronen Mehrspurautomaten, wie beispielsweise das Projizieren oder Wechseln von Spuren. Anhand dieser Definitionen wurden die Algorithmen innerhalb der eingesetzten Bibliothek implementiert. Die Bibliothek war dabei jedoch nur bedingt hilfreich und konnte nur einige wenige Funktionen und Konzepte beisteuern. Durch den von der Bibliothek definierten Rahmen war die Implementierung der Algorithmen zum Teil schwierig und teilweise eingeschränkt.

Die in der Einleitung gesetzten Ziele der Integration von automatischen Strukturen und der Beibehaltung der vorherigen Funktionalität wurden erfüllt. Weiterhin wurde im Rahmen dieser Arbeit auch die Möglichkeit geschaffen, die Implementierung des Model Checkers sowie Beweismodus um gegebenenfalls weitere Strukturen zu ergänzen.

5.2 Ausblick

Da sich diese Arbeit im Grundsatz mit der Möglichkeit der Integration von automatischen Strukturen beschäftigt, wurden einige vordefinierte automatische Strukturen fest in der Implementierung hinterlegt. Im Rahmen weiterer Arbeiten wäre es vorstellbar, sich mit einer zur Laufzeit möglichen Erzeugung solcher automatische Strukturen zu beschäftigen. Die Automaten innerhalb des Model Checkers könnten mithilfe einer Benutzeroberfläche definiert, oder diese durch einen Regulären Ausdruck eingelesen und in einen Automaten überführen werden.
Zusätzlich könnte eine Visualisierung für automatische Strukturen geschaffen werden, durch welche sich der Nutzer navigieren kann, um das Verständnis weiter zu fördern.

6 Literaturverzeichnis

[BG00] Blumensath, Achim ; Grädel, Erich: Automatic Structures. In: *Proceedings of 15th IEEE Symposium on Logic in Computer Science LICS 2000*, 2000, 51–62

[D'A] D'Antoni, Loris: *Symbolic automata*. `https://github.com/lorisdanto/symbolicautomata`. – (abgerufen am 13.05.2019)

[Fer] Fernández, Daniel: *javatuples*. `https://www.javatuples.org`. – (abgerufen am 01.04.2019)

[Fur12] Furia, Carlo A.: A Survey of Multi-Tape Automata. In: *CoRR* abs/1205.0178 (2012). `http://arxiv.org/abs/1205.0178`

[Gil] Giles, Jonathan: *ControlsFX*. `http://fxexperience.com/controlsfx/`. – (abgerufen am 01.04.2019)

[HMU06] Hopcroft, John E. ; Motwani, Rajeev ; Ullman, Jeffrey D.: *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 2006. – ISBN 0321455363

[KNRS07] Khoussainov, Bakhadyr ; Nies, André ; Rubin, Sasha ; Stephan, Frank: Automatic Structures: Richness and Limitations. In: *CoRR* abs/cs/0703064 (2007). `http://arxiv.org/abs/cs/0703064`

[Kum] Kumar, K. N.: *Automata, Logic, Games and Algebra*. `http://www.cmi.ac.in/~kumar/words/lecture07.pdf`. – (abgerufen am 10.03.2019)

[Lan15] Lange, Martin: Grundlagen der Programmsicherheit. (2015). – Unveröffentlichtes Skript, Universität Kassel

[Lan18] Lange, Martin: Theoretische Informatik: Logik. (2017/2018). – Unveröffentlichtes Skript, Universität Kassel

[Ott] Otto, Martin: *Notizen zur Logik erster Stufe*. https://moodle.informatik.tu-darmstadt.de/pluginfile.php/7066/course/summary/FOnotes15.pdf. – Technische Universität Darmstadt (abgerufen am 03.05.2019)