Endbericht

PG AAA

19. Dezember 2006

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Allgemeines

1.1 Motivation

Das Ziel der Projektgruppe 479 AAA („Triple-A", Automatische Analyse mit Automaten) bestand darin, einen Softwarebaukasten zu entwickeln, der verschiedene Automatentypen und darauf basierende Analyseverfahren bereithält. Der Automatenteil sollte als eine auf den allgemeinen, flexiblen und komfortablen Einsatz in beliebigen Softwareprodukten zugeschnittene Bibliothek konzipiert werden, die zudem leicht zu erweitern ist. Bei den Analyseverfahren kann es sich zum Beispiel um Erfüllbarkeitstests für Formeln bestimmter Logiken oder um Model Checking handeln, wobei Automaten als Datenstruktur zur Repräsentation bestimmter Sachverhalte wie etwa Lösungsmengen dienen. Automaten können auch selbst der Gegenstand der Analyse sein, ein Beispiel hierfür ist das Model Checking von Kripkemodellen. Es war nicht vorab spezifiziert, welche Automatentypen oder Analyseverfahren implementiert werden sollten. Ein weiteres Ziel der PG war die Entwicklung einer grafischen Benutzeroberfläche, die alle Elemente des Baukastens zugänglich macht und ebenfalls erweiterbar ist. Insbesondere sollten die Analyseprozesse von der Oberfläche aus mit geeigneten Eingaben gestartet werden können und die Ergebnisautomaten auf Wunsch grafisch dargestellt werden können. Auf die erstellten Automaten sollten die Operationen des jeweiligen Automatentyps angewendet werden können. Außerdem waren Editoren zur manuellen Erstellung von Automaten wünschenswert.

Der Zweck eines solchen Baukastens besteht vor allem darin, dem Anwender die Mächtigkeit von Automatenmodellen zur Verfügung zu stellen. Automaten sind zu den wichtigsten und am gründlichsten erforschten Strukturen der Informatik zu zählen. Für die Standardoperationen wurden möglichst effiziente Algorithmen entwickelt, viele Eigenschaften von Automaten wurden untersucht, und viele Probleme sind durch Automaten modellierbar. Mit der Modellierbarkeit eines Problems als Automat oder Kollektion

von Automaten kann der Lösungsprozess jedoch in vielen Fällen auf eine endlich häufige Anwendung der wohlbekannten und gut beherrschbaren Standardoperationen reduziert werden, anstatt völlig neue Algorithmen zu entwerfen. Natürlich wird mit dieser Vorgehensweise nicht unbedingt die bestmögliche Laufzeit erreicht, aber automatenbasierte Verfahren können durchaus praktikabel sein. In jedem Fall erachtet die Projektgruppe diese spezielle Art von Verfahren als sehr interessant. Einsatzmöglichkeiten der Automatenbibliothek bestehen beispielsweise im Bereich des Model Checking und der Verifikation von Programmen.

Es wurde von Anfang an als eher unwahrscheinlich eingeschätzt, dass im Rahmen der PG ein direkt in kommerziellen Projekten einsetzbares Endprodukt entsteht. Vielmehr sollte das Produkt für den akademischen Bereich von Interesse sein. Neue automatenbasierte Algorithmen können auf recht einfache Weise als weitere Analyse integriert werden, und die Automatenbibliothek ist um neue Typen von Automaten erweiterbar. Alle neuen Komponenten können in standardisierter Weise über die grafische Oberfläche zugänglich gemacht werden. Dort kann dann das Verhalten der Algorithmen auf konkreten Eingaben untersucht werden. Deshalb, und wegen der visuellen Darstellung von Automaten könnte auch ein Einsatz in der Lehre als Werkzeug zur Veranschaulichung in Betracht kommen. Wenn der Baukasten wirklich so flexibel und erweiterbar wird wie geplant, und vor allem die Automatenbibliothek effizient arbeitet, dann ist das Produkt sehr ausbaufähig.

Selbstverständlich existierte bereits ein „Markt“ für Automatenbibliotheken. Nach der Untersuchung einiger Automaten- und Graphenbibliotheken ist die Projektgruppe jedoch zu dem Schluss gekommen, dass keine von ihnen das Potenzial hatte, zu einem Baukasten mit den von uns gewünschten Eigenschaften ausgebaut oder umgebaut zu werden. Die Ergebnisse der Untersuchungen sind in Abschnitt 2.4.2 dokumentiert. So ließen alle betrachteten Bibliotheken ein umfassendes Alphabetkonzept vermissen, sondern verwendeten als Alphabet beispielsweise den Unicode-Zeichensatz. Automatenbasierte Analyseverfahren setzen jedoch die verschiedenartigsten Alphabete voraus, darunter Bitvektoralphabete. Einige Bibliotheken hatten einen eher geringen Funktionsumfang, oder aber die Schwerpunkte entsprachen nicht unseren Interessen.

Aus den genannten Gründen wurde der Beschluss gefasst, eine komplett eigene Automatenbibliothek zu implementieren, die vollständig an unseren Anforderungen ausgerichtet ist. Damit entfällt auch die Problematik fremder Lizenzen. Mehrere Analyseverfahren sollten aufbauend auf unserer Bibliothek realisiert werden, ebenso die grafische Oberfläche. Die Benutzerschnittstelle sollte generisch sein, indem sie ein Framework mit Basisklassen und Standardkomponenten zur Verfügung stellt. Da die in die engere Auswahl aufgenommenen Analyseverfahren (auch) Formeln als Eingaben erwarteten, mussten auch geeignete Parser für die entsprechenden Logiken entwickelt werden.

1.2 Übersicht

Die Projektgruppe begann ihre Arbeit mit einer Seminarphase, um sich die für die Verwirklichung der Ziele erforderlichen theoretischen Hintergründe und praktischen Kenntnisse anzueignen. Es wurden einerseits einige Automatenmodelle und automatenbasierte Analysemethoden vorgestellt, andererseits zählten auch konkrete Werkzeuge und Automatenbibliotheken zu den Themen. Insgesamt wurde mehr Material behandelt, als für die eigentliche Projektarbeit erforderlich war. Allerdings ermöglichte dies die Auswahl besonders geeigneter oder interessanter Gebiete und verhinderte eine zu frühe Beschränkung auf wenige Themenkreise, die sich später als inadäquat hätte erweisen können.

Nach Abschluss der Seminarphase wurde ziemlich schnell deutlich, dass die PG eine komplette Eigenentwicklung anstrebte. Da sie sich damit ein ehrgeiziges Ziel gesetzt hatte, ist im ersten Semester zunächst ein minimales, aber vollständiges System implementiert worden, das nur einen Automatentypen und ein Analyseverfahren sowie einen geeigneten Parser und eine grafische Oberfläche enthält. Es ging in erster Linie darum, ein Gefühl für die Problemstellung zu bekommen, aber bereits hier wurde auch ein Augenmerk auf Erweiterbarkeit gelegt. Es wurde davon ausgegangen, dass die Verwirklichung des Zieles eines universellen Baukastens in einem einzigen Schritt zu kompliziert ist. Für das Minimalprojekt fiel die Wahl auf ein auf endlichen Automaten basierendes Verfahren zur Erfüllbarkeitsanalyse von Formeln der Presburger-Logik. Im Weiteren sollte sich dann herausstellen, wie flexibel und erweiterbar dieses erste Konzept bereits ist. Die PG rechnete damit, dass das System sicherlich als Basis für die weitere Entwicklung dienen kann, auch wenn die gewünschte Flexibilität und Erweiterbarkeit erst noch unter Beweis gestellt werden musste. Eine komplette vorliegende Lösung sollte analysiert und in Frage gestellt werden können, um die Qualität der Lösung abzusichern. Denn es ist typisch, dass bei der Konzeptionalisierung eines Projektes und der Implementierung sich nach einiger Zeit herausstellt, dass gewisse Details noch überdacht werden müssen.

Das Minimalprojekt wurde gegen Ende des ersten Semesters fertiggestellt. Es enthielt eine Automatenbibliothek mit einer abstrakten Automaten-Basisklasse und einer konkreten Unterklasse zur Repräsentation von deterministischen und nichtdeterministischen endlichen Automaten. Die Automaten unterstützten Unicode-Character-Alphabete und Bitvektoralphabete und boten die üblichen Standardoperationen an. Auf dieser Basis war es möglich, Presburger-Formeln in endliche Automaten zu übersetzen, die die Lösungsmenge der jeweiligen Formel repräsentierten. Die Analyse und die Automatenoperationen konnten über die grafische Oberfläche aktiviert werden, und Automaten wurden auch visuell dargestellt, wobei ihr Layout mit dem Fruchterman-Reingold-Algorithmus erstellt wurde. Die grafische Darstellung von Automaten beruhte auf einem flexiblen, selbst entwickel-

ten Framework, das sich auch bei den Erweiterungen im zweiten Semester bewährte. Die Automatenbibliothek musste im zweiten Semester noch einmal verallgemeinert, jedoch nicht komplett neu implementiert werden. Insgesamt wurde das Minimalprojekt von allen Teilnehmern als gelungen akzeptiert.

Zu Beginn des zweiten Semesters musste dann entschieden werden, welche weiteren Automatentypen und Analyseverfahren in unser Projekt aufgenommen werden sollten. Baumautomaten wurden als sehr interessant erachtet, weil sie vielfältige Einsatzmöglichkeiten haben und sich deutlich von den klassischen Automatenmodellen unterscheiden. Die Integration dieses Automatentyps in die Bibliothek würde die Erweiterbarkeitskonzepte der Bibliothek auf eine harte Probe stellen. Allerdings fiel die Entscheidung dann doch zugunsten von Büchiautomaten aus. Büchiautomaten sind endliche Automaten, die nur unendliche Wörter akzeptieren können. Strukturell unterscheiden sie sich nicht von den klassischen endlichen Automaten, allerdings laufen die Operationen nicht identisch ab. Einige bei klassischen Automaten übliche Operationen wie zum Beispiel die Komplementbildung sind hier nicht ausreichend effizient berechenbar, und es kommen weitere Operationen wie zum Beispiel die Degeneralisierung hinzu. Büchiautomaten hatten den Vorteil, dass an das bereits Erreichte angeknüpft werden konnte, statt etwas völlig Neues zu beginnen. Dies war auch im Hinblick auf den im zweiten Semester stark begrenzten Zeitrahmen von Bedeutung. Für das Modelchecking wurden Kripkemodelle implementiert. Auch wenn diese im eigentlichen Sinne keine richtigen Automaten sind, besitzen sie jedoch gemeinsame Strukturen, so dass dies durch Vererbung ausgenutzt wurde. Auf der Basis von Büchiautomaten und der Kripkemodelle war es schliesslich möglich, Verfahren für das Model Checking von Kripkemodellen mit LTL- und CTL-Formeln zu implementieren. Es wurde sogar eine While-Sprache entwickelt, für die sich Kripkemodelle generieren lassen, so dass While-Programme verifiziert und untersucht werden können. Der Automatenbibliothek sind einige Typen von Alphabeten hinzugefügt worden, und vor allem ist die Repräsentation von Transitionen verallgemeinert worden. Einzelne Transitionen können nun mit Mengen von Symbolintervallen oder aber mit OBDDs beschriftet werden. Das flexible Driver-Konzept der grafischen Oberfläche wurde leicht angepasst, um neben der universellen Darstellung auch auf die speziellen Eigenheiten der neuen Automantentypen (bzw. des Kripkemodelles) anwendbar zu sein.

Zunächst wird ein Abriss über die in der Seminarphase behandelten Themen gegeben. Im Anschluss werden dann die Ergebnisse der einzelnen Gruppen vorgestellt. Die Gruppenergebnisse werden nicht grundsätzlich getrennt nach erstem und zweiten Semester dargestellt, dennoch wurde darauf geachtet, dass der Verlauf der Entwicklung nachvollziehbar bleibt. Die Projektgruppe hat sich im ersten Semester in die Teilgruppen „Automaten-Team“ (4 Mitglieder), „Workspace-Team“ (4 Mitglieder), „Transformations-Team“ (1 Mitglied) und „Parser-Team“ (2 Mitglieder) aufgeteilt. Im zweiten Semester bestand das

Workspace-Team nur noch aus 2 Mitgliedern, das nur noch aus einem Mitglied bestehende Parser-Team wurde etwa nach der ersten Hälfte aufgelöst und das Transformations-Team auf 2 Mitglieder aufgestockt. Das Automaten-Team blieb unverändert, wurde jedoch durch zwei weitere Mitglieder unterstützt, die sich der Realisierung der Operationen auf Büchiautomaten widmeten.

Kapitel 2

Seminarphase

Wie in jeder Projektgruppe üblich, fand zu Anfang eine Seminarphase statt. In der Seminarphase erhält jeder Teilnehmer ein Thema, worüber er einen Vortrag hält. Die Seminarphase dient dazu, das Themenfeld der Projektgruppe genauer kennen zu lernen und alle Teilnehmer auf einen gemeinsamen Wissensstand zu bringen.

Das Thema der Projektgruppe umfasste zum Teil Gebiete, welche nicht als selbstverständlich vorausgesetzt werden konnten. Auch diente in unserem Fall die Seminarphase dazu, einen Überblick über den "Markt", also die Vielzahl von Analysemethoden und Programmen, zu erhalten. Die Vorträge "Erfüllbarkeitsprüfung von Formeln der Presburger-Logik mit Hilfe endlicher deterministischer Automaten", "Erreichbarkeitsanalyse für Pushdown-Systeme", "Erreichbarkeitsanalyse paralleler Prozesse mit Baumautomaten" und "Model Checking mit Automaten" befassten sich mit Analysemethoden mittels verschiedener Automatenmodelle. Die Projektgruppe hat sich dann dazu entschieden, zwei solcher Analysemethoden, nämlich zunächst die "Erfüllbarkeit von Formeln der Presburger-Logik mit Hilfe endlicher deterministischer Automaten" und später "Model Checking mit Automaten", zu verwirklichen, weshalb die Kenntnis der entsprechenden Vorträge für das Verständnis und die Implementierung von Nutzen waren. Die Vorträge "Endliche Baumautomaten", "Alternierende Automaten" und "Büchiautomaten" liefern gewissermaßen das theoretische Fundament für die oben genannten Vorträge zu Analyseverfahren. Insbesondere der Vortrag über Büchiautomaten sei dem Leser ans Herz gelegt, da dieses Automatenkonzept dann auch in der Automatenbibliothek verwirklicht worden ist. Die Vorträge über alternierende Automaten und endliche Baumautomaten geben Anreize für mögliche Fortsetzungen des Projekts. Der Vortrag zur Erreichbarkeitsanalyse fällt etwas aus dem Rahmen; er fand im Verlauf des Projekts auch keine weitere Verwendung. Die Vorträge "Mona/Mosel", "jABC and Friends" und "Das Eclipse Project" sollten eine Übersicht über bestehende Werkzeuge und deren mögliche Verwendbarkeit geben. Es wurde keines dieser Tools verwendet. Der Vortrag "Überblick über Automaten-

Bibliotheken“ ist interessant, um einen Einblick in bestehende Automatenbibliotheken zu erhalten. Insbesondere der Automatenbibliothek “dk.brics“ ist Aufmerksamkeit zu schenken, da die Konzeptualisierung unserer Bibliothek darauf zurück geht und gewissermaßen eine Verallgemeinerung dieser Bibliothek darstellt. “Erfüllbarkeitsprüfung von Formeln der Presburger-Logik mit Hilfe endlicher deterministischer Automaten“ wurde in der ersten Phase der Projektgruppe als Analysemethode implementiert.

2.1 Automatenmodelle

2.1.1 Büchiautomaten

2.1.1.1 Endliche Automaten auf endlichen Wörtern

Hier soll ein Automatentyp vergestellt werden, der Wörter von unendlicher Länge erkennen kann. Zuerst wird aber auf endliche Automaten auf endlichen Wörtern eingegangen:

Definition (endlicher Automat): Ein *endlicher Automat* A ist ein Tupel $(\Sigma, Q, \Delta, Q^0, F)$. Dabei ist Σ das endliche Eingabealphabet und Q die endliche Menge an Zuständen. $Q^0 \subseteq Q$ ist die Menge an Anfangszuständen (meistens einelementig). Bei F handelt es sich um die akzeptierende Zustände. Die Δ-Relation stellt die möglichen Transitionen (Zustandsübergänge) des Automaten dar. Es gilt:

$$\Delta \subseteq \underbrace{Q}_{Alter\ Zustand} \times \underbrace{\Sigma}_{Eingabe} \times \underbrace{Q}_{Neuer\ Zustand}$$

Definition (Durchlauf): Sei das Wort $v \in \Sigma^*$ mit Länge $|v|$ gegeben. Dabei versteht man unter dem *Durchlauf* (oder *Lauf*) eine Folge von Zuständen $\rho_i \in Q$ der Länge $|v| + 1$.

- $\rho_0 \in Q^0$
  d. h. dass das erste Element der Folge aus einem Startzustand besteht (der Durchlauf fängt in einem Startzustand an).
- $0 \leq i < |v| : (\rho_i, v(i), \rho_{i+1}) \in \Delta$
  Für alle Elemente des Durchlaufs gilt, dass die Übergänge vom aktuellen Zustand ρ_i zum nächsten Zustand ρ_{i+1} erlaubte Transitionen sind.

Das bedeutet, dass man sich den Durchlauf von A über v als endliche Zustandsfolge oder Pfad innerhalb des Graphen vorstellen kann (vgl. [**?**] S. 121).

2.1.1.1.1 Determinismus Endliche Automaten auf endlichen Wörtern kann man unterteilen in deterministische (DFA) und nichtdeterministische (NFA). Bei einem NFA hat der Automat die Wahl zwischen mehreren erlaubten Transitionen.

2.1.1.2 Endliche Automaten auf unendlichen Wörtern

NFAs und DFAs sind so definiert, dass sie beim erreichen eines akzeptierenden Zustandes akzeptieren und halten. Endliche Automaten auf unendlichen Wörtern sind so definiert,

dass sie akzeptieren, sobald mindestens ein akzeptierender Zustand unendlich oft in einem Lauf vorkommt. Dies bedeutet, dass sie nie halten (wenn der Automat als eine reelle Maschine betrachtet wird). Das ist besonders interessant, da viele Systeme, bei denen man Model Checking anwenden will, auch nicht terminieren. Das wirft aber auch die Frage auf, wie man überhaupt überprüfen soll, ob ein solcher Automat akzeptiert. Als direkte Folgerung aus den o.g. Argumenten ergibt sich, dass Wörter, die von einem solchen Automaten erkannt werden, sich jeweils aus einem endlichen Anfangsstück und einem sich unendlich wiederholenden Endstück endlicher Länge zusammensetzen. Spätestens das Endstück schließt also einen Kreis. Die akzeptierten Wörter v haben also die formale Form: $v \in \Sigma^\omega$ (wobei ω die unendliche Entsprechung des endlichen regulären Operators $*$ ist). Deshalb nennt man die Automaten auch ω-Automaten, bzw die Sprachen, die sie erkennen, ω-Sprachen (vgl. [**?**]).

2.1.1.2.1 Büchiautomaten *Büchiautomaten (BA)* gehören zu den einfachsten Vertretern der ω-Automaten. Obwohl sie genauso aussehen wie endliche Automaten über endlichen Wörtern, sind F nicht mehr Zustände, deren einmaliges Erreichen schon zur Akzeptanz führt, sondern Zustände die unendlich oft durchlaufen werden müssen damit das Wort akzeptiert werden kann. Sie werden benannt nach Julius Richard Büchi (1924-1984) (vgl. [**?**, **?**]).

2.1.1.2.2 Akzeptanzprüfung bei Büchiautomaten Aber wie akzeptiert ein BA ein unendliches Wort, wenn er nicht anhalten kann? Man kann das Problem mit Hilfe des Durchlaufs in den Griff bekommen.
Definition(inf(ρ)): inf(ρ) ist die Menge aller Zustände die unendlich oft im Durchlauf ρ vorkommen. Ein Durchlauf von A über v gilt als akzeptierend, genau dann, wenn gilt:

$$\inf(\rho) \cap F \neq \emptyset$$

Das bedeutet, dass A v genau dann akzeptiert, wenn ein akzeptierender Zustand (also ein Zustand aus der Menge F) unendlich oft auf dem Pfad vorkommt.

Natürlich bleibt das Problem, dass man weder einen unendlichen Pfad auf unendliches Vorkommen von Elementen aus F testen kann, noch ihn zum Testen unendlich lange rechen lassen kann. Es handelt sich dabei eher um eine Art Grenzwertbetrachtung.

Eine Sprache, die durch einen BA erkannt werden kann, wird Büchi-erkennbar (büchirecognizable) genannt (vgl. [**?**] S. 120f).

2.1.1.2.3 Nichtdeterministische Büchiautomaten Bei Büchiautomaten ist, so wie bei endlichen Automaten für endliche Wörter, die Δ-Relation nicht zwingend als

deterministisch angenommen (NFA). Das bedeutet, dass gelten kann:

$$(q, a, l), (q, a, l') \in \Delta, \text{ mit } l \neq l'.$$

Es existieren also von einem Zustand zwei Übergänge mit dem selben gelesenen Zeichen zu verschiedenen Zuständen. Deterministische Büchiautomaten bekommen im Allgemeinen das Kürzel DBA, nichtdeterministische das Kürzel NBA.

2.1.1.2.4 Transformation eines NBAs zu einem DBA NFAs kann man in DFAs umwandeln mit Hilfe der Potenzmengenkonstruktion (auch wenn sie ε-Transitionen beinhalten). Das funktioniert, indem jeder Zustand des deterministischen Automaten für die Zustandmenge steht, in der sich der nichtdeterministische befinden kann.

Etwas Derartiges ist für Büchiautomaten nicht möglich, da Nichtdeterminismus hier zusätzliche Ausdruckskraft bedeutet. Z. B. die Sprache $\{v \in \Sigma^\omega | v = \Sigma^* b^\omega\}$ über dem Alphabet $\Sigma = \{a, b\}$ wird von einem nichtdeterministischen Büchiautomaten erkannt. Es existiert aber kein DBA dafür.

Folglich ist die Menge der Sprachen, die DBAs erkennen können, in der der NBAs enthalten, aber nicht andersherum. Folglich sind sie auch nicht gleich und es existiert keine allgemeine Transformation NBA $\rightarrow$ DBA (vgl. [**?**],[**?** S. 127).

2.1.1.3 Operationen auf Büchiautomaten

Büchiautomaten, genauer gesagt deren Sprachen, sind abgeschlossen gegenüber Komplement, Schnitt und Vereinigung.

2.1.1.3.1 Schnitt von büchierkennbaren Sprachen Es soll der Schnitt der erkannten Sprachen der Automaten B_1 und B_2 berechnet werden. Die Grundidee ist, dass wir einen Zähler verwenden, der am Anfang auf 0 steht. Wir warten, bis der Automat B_1 in einem akzeptierenden Zustand ist, dann setzen wir den Zähler auf 1 und warten bis B_2 akzeptiert. Wenn das geschehen ist, setzen wir den Zähler auf 0 zurück. Wenn das unendliche Wort wirklich aus dem Schnitt der Sprachen stammt, dann muss unendlich oft B_1 und B_2 akzeptiert werden. Der neue Schnittautomat akzeptiert nur, wenn er von der 1 wieder in die 0 wechselt. Formal sieht das folgendermaßen aus (vgl.[**?**]):

Seien $B_1 = \langle \Sigma, Q_1, \Delta_1, Q_1^0, F_1 \rangle$ und $B_2 = \langle \Sigma, Q_2, \Delta_2, Q_2^0, F_2 \rangle$ gegeben. Der Automat den wir konstruieren sieht wie folgt aus:

$$B_1 \cap B_2 = \left\langle \Sigma, Q_1 \times Q_2 \times \{0, 1\}, \Delta, Q_1^0 \times Q_2^0 \times \{0\}, Q_1 \times F_2 \times \{1\} \right\rangle.$$

Die Transitionen werden wie folgt gebildet:

$(\langle r_i, q_j, x\rangle, a, \langle r_m, q_n, y\rangle) \in \Delta \Leftrightarrow$ die folgenden beiden Bedingungen sind erfüllt:

1. $(r_i, a, r_m) \in \Delta_1 \wedge (q_j, a, q_n) \in \Delta_2$ (Transitionen der Automaten übernehmen)
2. Es gilt:
   - $x = 0 \wedge r_m \in F_1 \Rightarrow y = 1$ (B_1 hat akzeptiert)
   - $x = 1 \wedge q_n \in F_2 \Rightarrow y = 0$ (B_2 hat akzeptiert und Zähler wieder auf 0 $\Rightarrow$ Schnittautomat akzeptiert)
   - sonst $y = x$.

Abbildung 2.1 geschnitten mit Abbildung 2.2 liefern als Ergebnis Abbildung 2.3.

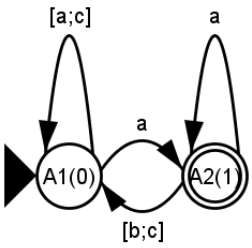


Abbildung 2.1: Automat A für Schnitt

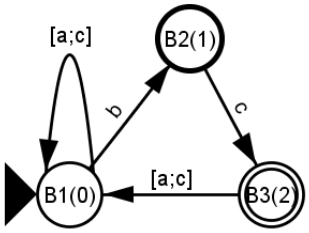


Abbildung 2.2: Automat B für Schnitt

2.1.1.3.2 Vereinigung von büchierkennbaren Sprachen Die Vereinigung ist für Model Checking nicht besonders interessant. Deshalb wird sie hier kaum behandelt. Die Idee ist, dass man wie bei NFAs einfach einen neuen Initialzustand mit ε-Transitionen zu den Initialzuständen der beiden Automaten erzeugt und die ε-Kanten danach einsinken lässt.

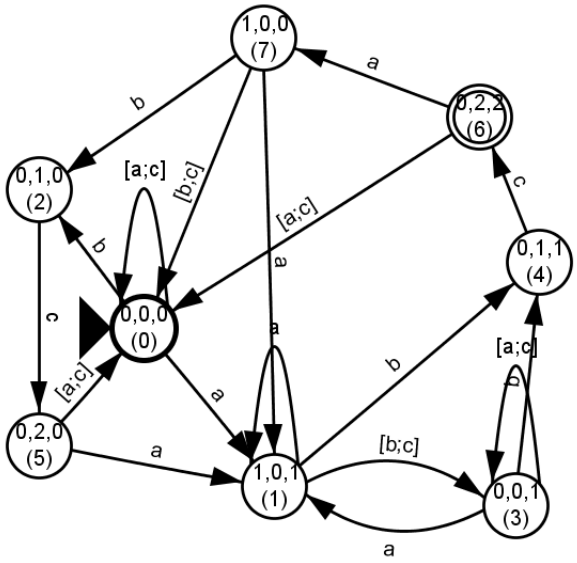


Abbildung 2.3: Ergebnis-Automat für Schnitt

2.1.1.3.3 Leerheitstest von büchierkennbaren Sprachen Diese Operation wird hier besprochen, da sie sich beim Model Checking als hilfreich erweist. Man kann die Leerheit beweisen oder widerlegen durch den Nachweis ihres Komplements (der Nichtleerheit). Um ein Wort zu akzeptieren, muss der Lauf ρ auf dem Wort v unendlich oft einen akzeptierenden Zustand durchlaufen haben. D. h., dass er eine stark zusammenhängende Komponente mit einem akzeptierenden Zustand bilden muss, die vom Initialzustand erreichbar ist. Wenn es eine solche Komponente gibt, muss folglich mindestens ein Wort akzeptiert werden und die Sprache kann damit nicht leer sein. Anders formuliert, wird ein erreichbarer akzeptierender Zustand gesucht, der Teil eines Kreises ist. Da die Existenz eines einzigen akzeptierten Wortes bereits ausreicht, kann man das Problem zum Beispiel mit Hilfe von zwei DFS's vom Initialzustand aus lösen: siehe Algorithmus 1 (vgl. [**?**] S.

129f).

```
procedure emptiness
    for all q_0 ∈ Q^0 do
    dfs1(q_0);
    terminate(false);
end procedure
procedure dfs1(q)
    local q';
    hash(q);
    for all successors q' of q do
        if q' not the hash table then dfs1(q');
    if accept(q) then dfs2(q);
end procedure
procedure dfs2(q)
    local q';
    flag(q);
    for all successor q' of q do
        if q' on dfs1 stack then terminate (True)
        else if q' not flaggt then dfs2(q');
        end if;
end procedure
```

Algorithmus 1 : Leerheitstest für BAs

2.1.1.3.4 Komplement von büchierkennbaren Sprachen Ein Vorgehen wie bei einem endlichen Automaten für endliche Wörter ist hier nicht möglich. Grob gesagt kann man dort einfach die Akzeptanz-Eigenschaft jedes einzelnen Zustands negieren und erhält auf diese Weise einen Automaten, der $\overline{L}$ akzeptiert. Man kann zwar einen BA konstruieren, der das Komplement der Sprache erkennt, hat allerdings das Risiko eines exponentiellen Blowups. Die Konstruktion oder der Beweis der Eigenschaft ist für diesen Text zu umfangreich. Es funktioniert zum Beispiel mit der Konstruktion von Safra (vgl. [**?**] S. 125).

2.1.1.3.5 Verallgemeinerte Büchiautomaten Verallgemeinerte (oder generalisierte) Büchiautomaten können mehrere Klassen an akzeptierenden Zuständen haben:

$P_i \subseteq F$ für $0 \leq i \leq n$

Dieser Automat akzeptiert erst, wenn mindestens ein akzeptierender Zustand je Klasse unendlich oft durchlaufen wurde. Das bedeutet, er akzeptiert, genau dann, wenn gilt:

$$\forall P_i \subseteq F : \inf(\rho) \cap P_i \neq \emptyset.$$

Das heißt aber auch, dass der Automat akzeptiert, wenn gilt: $F = \emptyset$. In diesem Fall würde jedes Wort über Σ^ω akzeptiert. Trotzdem ist der verallgemeinerte Büchiautomat nicht ausdrucksstärker, was sich auch in der Transformierbarkeit zu einem Büchiautomaten zeigt.

Nehmen wir an, dass gilt: $F = \{P_1, .., P_n\}$. Dann konstruiert man den Büchiautomaten $B' = \langle \Sigma, Q \times \{1, .., n\}, \Delta', Q^0 \times \{1\}, P_n \times \{n\} \rangle$. Dabei wird Δ' definiert als $(\langle q, x \rangle, a, \langle q', y \rangle) \in \Delta'$, wenn $(q, a, q') \in \Delta$ und x und y folgende Bedingungen erfüllen:

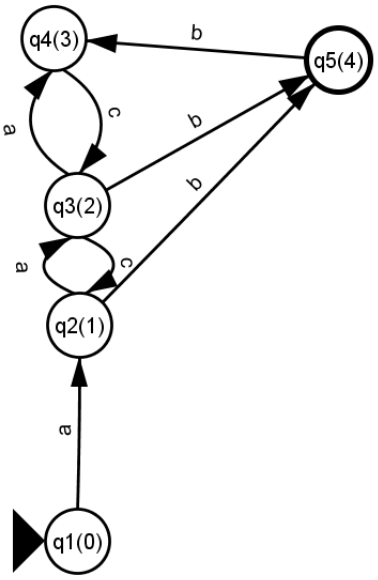


Abbildung 2.4: Ursprungsautomat für Degeneralisierung

- $q \in P_i \wedge x = i \Rightarrow y = \begin{cases} i+1 & \text{falls } i \neq n \\ 1 & \text{falls } i = n \end{cases}$
- Sonst: $x = y$.

Die Konstruktion bedeutet als Graph betrachtet, dass man den Automaten so oft kopiert, wie es Akzeptanzmengen gibt. Allerdings wird jede Transition in einem akzeptierenden Zustand derjenigen Akzeptanzklasse, für die die Kopie steht, so umgebogen, dass sie in den gleichen Zustand der nächsten Kopie geht. Die letzte Kopie verweist aber wieder zurück auf die erste Kopie. Nur die akzeptierenden Zustände der ersten Kopie sind auch im neuen Automaten akzeptierend. So kann das Wort nur erkannt werden, wenn der Automat alle Zustände erreicht hat und durch alle Kopien gelaufen ist. Wie man leicht vermuten kann, entstehen nicht gerade übersichtliche und redundanzfreie Automaten.

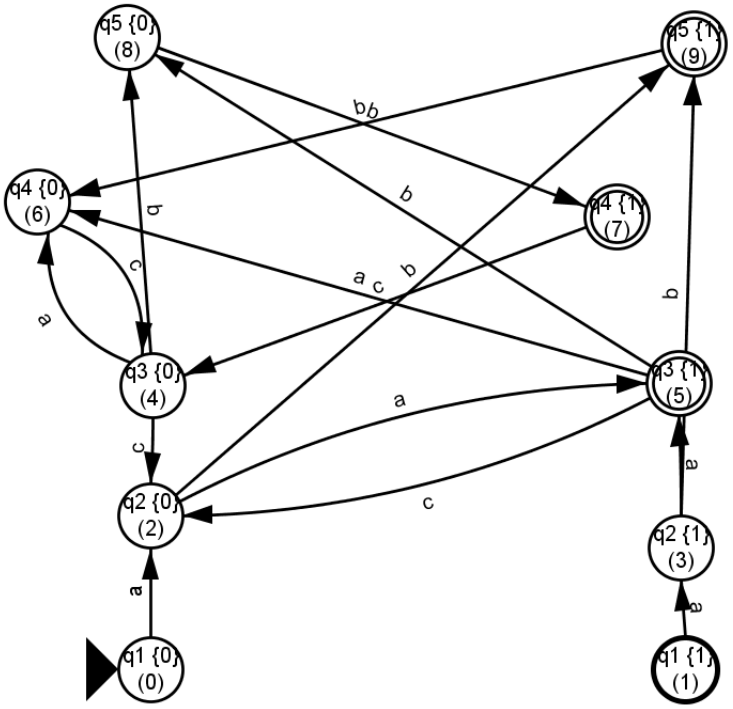


Abbildung 2.5: Ergebnis der Degeneralisierung

Dass der generalisierte BA auch nicht weniger erkennen kann, ist trivial, da ein BA als ein generalisierter BA mit nur einer Klasse von akzeptierenden Zuständen aufgefasst werden kann (vgl. [**?**] S. 128).

Für ein Beipiel siehe Abbildung 2.4 (Akzeptanzmengen: $\{q_2, q_5\}, \{q_3, q_4, q_5\}$) und 2.5.

2.1.2 Endliche Baumautomaten

Automaten sind in der Informatik ein weit verbreitetes Konzept, das oft Gegenstand theoretischer Forschung ist. Es hat sich jedoch herausgestellt, dass kaum ein theoretisches Teilgebiet der Informatik einen derart großen praktischen Nutzen hat wie die Automatentheorie. Im Gegensatz zu herkömmlichen Automaten, die nur Wörter erkennen, gibt es eine besondere Klasse von Automaten, die Baumautomaten, die herkömmliche Automaten erweitern, so dass sie in der Lage sind, baumartige Strukturen zu erkennen. Konkret heißt das, dass Baumautomaten Sprachen wie Boolesche Terme erkennen können.

Im folgenden Text werden die Grundlagen für endliche Baumautomaten dargestellt. Dabei liegt der Schwerpunkt auf den Bottom-Up Baumautomaten, die Top-Down Baumautomaten werden nur kurz angesprochen. Die Grundlage für diesen Text ist [**?**].

2.1.2.1 Bottom-Up Baumautomaten

Endliche Baumautomaten arbeiten auf endlichen geordneten Bäumen, die durch Terme dargestellt werden. Bottom-Up Baumautomaten starten ihren Lauf an den Blättern und bewegen sich von dort aus nach oben zur Wurzel.

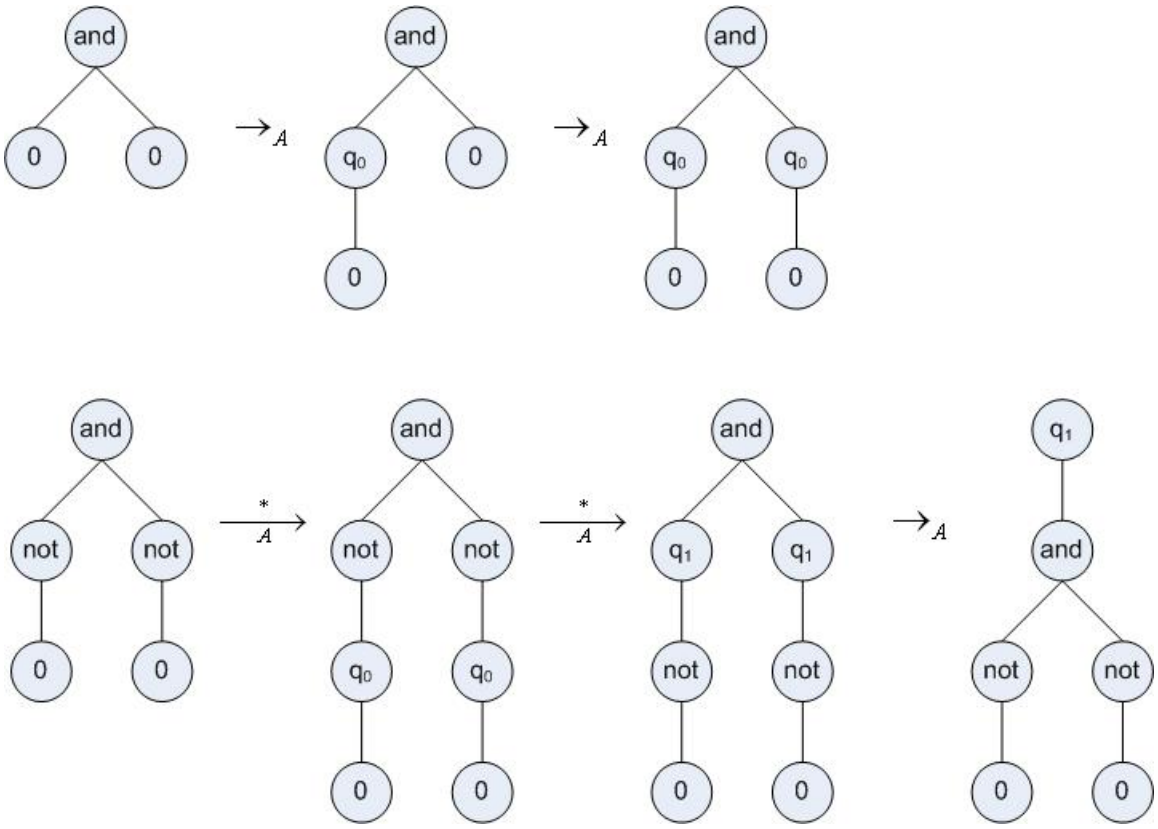
2.1.2.1.1 NFTA (Nondeterministic Finite Tree Automata) Die Menge $T(F, \chi)$ von Termen über dem geordneten Alphabet F und der Menge χ von Variablen ist die kleinste Menge, die definiert ist durch:
$F_0 \subseteq T(F,\chi) \wedge \chi \subseteq T(F,\chi)$ und, wenn $p \geq 1 \wedge f \in F_p \wedge t_1, ..., t_p \in T(F,\chi)$, dann ist $f(t_1, ..., t_p) \in T(F,\chi)$ (mit F_0 den 0-stelligen Zeichen, d. h. Konstanten und F_p den p-stelligen Zeichen). Wenn χ leer ist, dann schreibt man auch $T(F)$. Terme aus $T(F)$ heißen Grundterme.
Ein nichtdeterministischer endlicher Baumautomat über einem endlichen Alphabet F ist ein Tupel
$A = (Q, F, Q_f, \triangle)$, wobei Q eine Menge von Zuständen, $Q_f \subseteq Q$ eine Menge von akzeptierenden Zuständen, χ die Menge von Variablen und $\triangle$ eine Menge von Zustandsübergangsfunktionen von folgendem Typ ist:
$f(q_1(x_1), ..., q_n(x_n)) \rightarrow q(f(x_1, ..., x_n))$, wobei $n \geq 0$, $f \in F_n$ (mit F_n den n-stelligen Funktionen),
$q, q_1, ..., q_n \in Q$, $x_1, ..., x_n \in \chi$.

2.1.2.1.1.1 Move-Relation Um herauszufinden, ob ein Term von einem Baumautomaten akzeptiert wird, kann man Zustände in einen Baum über den Blättern einfügen, die man dann immer weiter nach oben verschiebt und aktualisiert. Ein Term/Baum wird dann akzeptiert, wenn sich zum Schluss an der Wurzel ein akzeptierender Zustand befindet.

Definition Sei $A = (Q, F, Q_f, \triangle)$ ein NTFA über F. Die Move-Relation $\overset{A}{\rightarrow}$ ist definiert durch:
seien $t, t' \in T(F \cup Q)$, mit $t \overset{A}{\rightarrow} t'$. Es existiert eine Substitution, so dass t gemäß der Regeln aus $\triangle$ zu t' substituiert werden kann.
Wir schreiben $\overset{A,*}{\rightarrow}$, wenn wir $\overset{A}{\rightarrow}$ endlich oft anwenden.

Beispiel Sei $F = \{and(,), not(), 0\}$.Wir betrachten den Automaten $A = (Q, F, Q_f, \triangle)$ definiert durch:
$Q = \{q_0, q_1\}$, $Q_f = \{q_1\}$ und
$\triangle = \{0 \rightarrow q_0(0), not(q_1(x)) \rightarrow q_0(not(x)), not(q_0(x)) \rightarrow q_1(not(x))\} \cup$

$\{and(q_1(x), q_1(y)) \rightarrow q_1(and(x, y))\}$



2.1.2.1.1.2 Lauf Man kann auch anders als durch die Move-Relation herausfinden, ob ein Term von einem Baumautomaten akzeptiert wird. Anstatt Zustände in den Baum einzufügen und sie dann zu verschieben, kann man auch von den Blättern angefangen jeden Teilterm oder jede Konstante aus einem Knoten auf einen Zustand abbilden.

Definition Sei t ein Grundterm und A ein NTFA. Ein Lauf r auf A über t ist eine Abbildung $r : Pos(t) \to Q$ (wobei $Pos(t)$ die Position von t bezeichnet, welches sowohl eine Konstante als auch eine mehrstellige Funktion sein kann) passend zu $\triangle$.

Der NTFA beginnt an den Blättern und bewegt sich nach oben, wobei er entlang seines Weges jede Position des Terms, die er bei einem Knoten findet, auf einen Zustand abbildet, bis zur Wurzel. Der Baum, bzw. der Term, wird akzeptiert, wenn nach dem Lauf an der Wurzel ein akzeptierender Zustand steht.

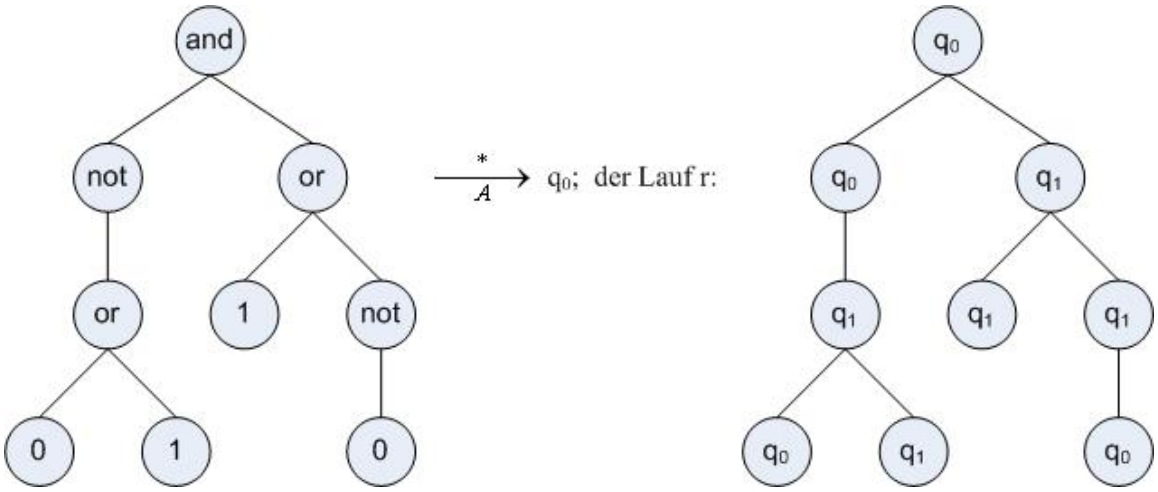
Beispiel Sei $F = \{or(,), and(,), not(), 0, 1\}$.Wir betrachten den Automaten $A = (Q, F, Q_f, \triangle)$ definiert durch:
$Q = \{q_0, q_1\}$, $Q_f = \{q_1\}$ und
$\triangle = \{0 \to q_0, 1 \to q_1, not(q_0) \to q_1, not(q_1) \to q_0, and(q_0, q_0) \to q_0, and(q_0, q_1) \to q_0\} \cup$

$\{and(q_1, q_0) \to q_0, and(q_1, q_1) \to q_1, or(q_0, q_0) \to q_0, or(q_0, q_1) \to q_1\} \cup$

$\{or(q_1, q_0) \to q_1, or(q_1, q_1) \to q_1\}$

Gegeben sei der Grundterm $t = and(not(or(0, 1)), or(1, not(0))$, mit dem dazugehörigen

Lauf, gegeben als ein Baum



Die Baumsprache, die A erkennt, ist die Menge aller Booleschen Ausdrücke über F, die true sind. Der Baumautomat ist ein vollständiger und minimaler DFTA.

2.1.2.1.2 DFTA (Deterministic Finite Tree Automata) Deterministische Baumautomaten sind ein Spezialfall von NFTAs. Jede Sprache, die von einem NFTA erkannt wird, kann auch von einem DFTA erkannt werden.
Ein Baumautomat $A = (Q, F, Q_f, \triangle)$ ist deterministisch, wenn es keine zwei Regeln gibt, die die gleiche linke Seite haben (und keine ϵ-Regeln). In einem DFTA gibt es höchstens einen Lauf für jeden Grundterm.

Definitionen Ein NFTA ist **vollständig**, wenn es mindestens eine Regel $f(q_1, ..., q_n) \rightarrow q \in \triangle$, $\forall n \geq 0$, $f \in F_n$, $q_1, ..., q_n \in Q$. In einem vollständigen DFTA gibt es genau einen Lauf für jeden Grundterm.
Ein Zustand q ist **erreichbar**, wenn ein Grundterm t existiert, so dass $t \xrightarrow{A} q$. Ein NFTA A heißt **reduziert**, wenn alle Zustände erreichbar sind.

2.1.2.1.3 Baumhomomorphismen Baumhomomorphismen sind Transformationen, die die Baumstruktur erhalten. Sie sind eine Verallgemeinerung der Homomorphismen für Wörter (einstellige Terme). Beim Wortproblem ist bekannt, dass die regulären Mengen unter Homomorphismen und inversen Homomorphismen abgeschlossen sind, beim Baumproblem ist das anders. Falls erkennbare Baumsprachen unter inversen Homomorphismen abgeschlossen sind, sind sie nur unter einer Subklasse von Homomorphismen abgeschlossen.

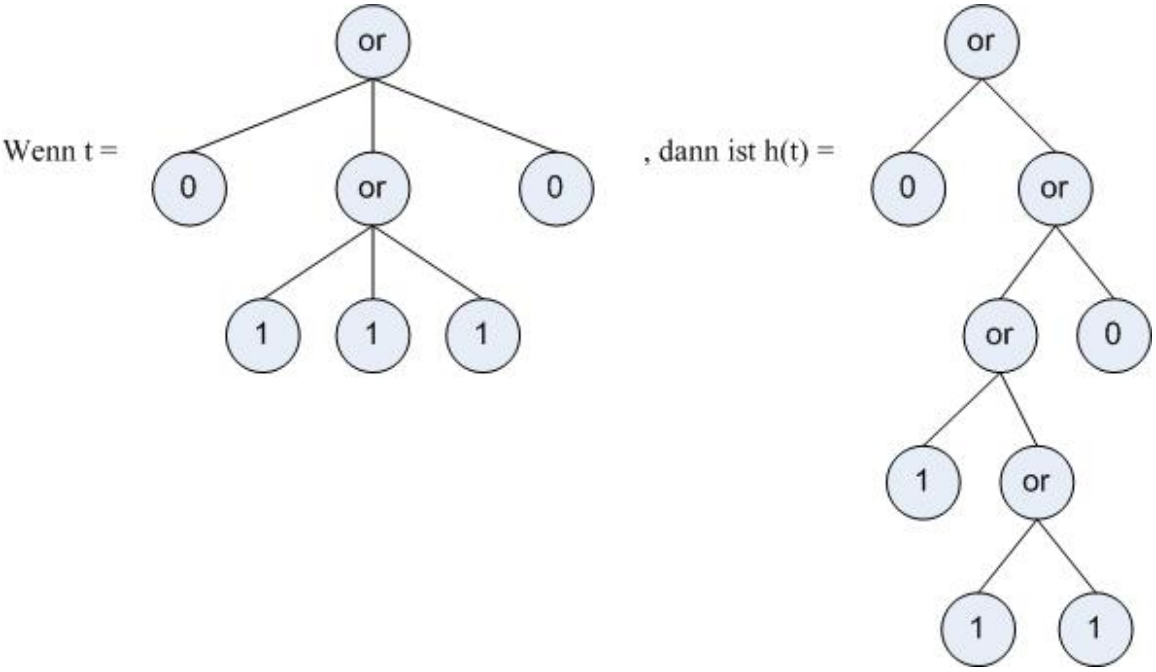
Definition Seien F und F' zwei Mengen von Funktionssymbolen. Für jedes $n > 0$, so dass F ein Symbol der Stelligkeit n enthält, definieren wir eine Menge von Variablen $\chi_n = \{x_1, ..., x_n\}$ disjunkt von F und F'. Sei h_F eine Abbildung, die mit $f \in F$ von Stelligkeit

n einen Term $t_f \in T(F', \chi_n)$ verbindet. Der Baumhomomorphismus $h : T(F) \to T(F')$, bestimmt durch h_F, ist folgendermaßen definiert:

- $h(a) = t_a \in T(F')$ gilt für jedes $a \in F_0$,
- $h(f(t_1, ..., t_n)) = t_f \{x_1 \leftarrow h(t_1), ..., x_n \leftarrow h(t_n)\}$,

wobei $t_f \{x_1 \leftarrow h(t_1), ..., x_n \leftarrow h(t_n)\}$ das Resultat der Anwendung der Substitution $\{x_1 \leftarrow h(t_1), ..., x_n \leftarrow h(t_n)\}$ auf den Term t_f ist.

Beispiel Sei $F = \{or(,,), 0, 1\}$ und $F' = \{or(,), 0, 1\}$. Wir betrachten den Baumhomomorphismus h bestimmt durch h_f und definiert durch: $h_f(or(x_1, x_2, x_3)) = or(x_1, or(x_2, x_3))$, $h_f(0) = 0$ und $h_f(1) = 1$. Wir können also einen Ternärbaum in einen Binärbaum umwandeln:



Bei der Anwendung von Baumhomomorphismen wird die Erkennbarkeit nicht immer erhalten.

2.1.2.1.4 Abschlusseigenschaften Die Klasse der erkennbaren Baumsprachen ist unter Vereinigung, Komplement und Schnitt abgeschlossen.

2.1.2.1.4.1 Vereinigung Für die Konstruktion der Vereinigung wählen wir den Produktautomaten, da dieser Determinismus bewahrt. Wenn die zwei gegebenen Automaten deterministisch sind, ist der Produktautomat auch deterministisch.

Konstruktion Seien A_1 und A_2 vollständig. Dann sei der FTA $A = (Q, F, Q_f, \triangle)$ definiert durch:
$Q = Q_1 \times Q_2$, $Q_f = Q_{f1} \times Q_2 \cup Q_1 \times Q_{f2}$ und $\triangle = \triangle_1 \times \triangle_2$, wobei
$\triangle_1 \times \triangle_2 = \{f((q_1, q_1'), ..., (q_n, q_n')) \rightarrow (q, q') \mid f(q_1, ..., q_n) \rightarrow q \in \triangle_1, f(q_1', ..., q_n') \rightarrow q \in \triangle_2\}$.
$L(A)$ und $L(A_1) \cup L(A_2)$ sind gleich. Wichtig ist, dass A_1 und A_2 vollständig sind.

2.1.2.1.4.2 Komplement Sei L eine erkennbare Baumsprache und sei $A = (Q, F, Q_f, \triangle)$ ein vollständiger DFTA, so dass $L(A) = L$. Nun bilden wir das Komplement der akzeptierenden Zustände, um L^C zu erkennen. Also sei $A^C = (Q, F, Q_f^C, \triangle)$ mit $Q_f^C = Q/Q_f$, so dass der DFTA A^C das Komplement der Menge L in $T(F)$ erkennt.
Möchte man das Komplement für einen NFTA bilden, muss man ihn zuerst in einen DFTA umwandeln und dann das Komplement der akzeptierenden Zustände bilden.

2.1.2.1.4.3 Schnitt Geschlossenheit unter Schnitt folgt aus der Geschlossenheit unter Vereinigung und Komplement, denn $L_1 \cap L_2 = \neg(\neg L_1 \cup \neg L_2)$.

2.1.2.1.5 Entscheidungsprobleme Im Folgenden sind drei wichtige Entscheidungsprobleme mit ihrer Komplexität kurz aufgeführt. Die Beweise finden sich in der angegebenen Literaturquelle, wo außerdem weitere Entscheidungsprobleme, wie Uniform Membership, nicht leerer Schnitt, Leerheit des Komplements, Singleton Set Property und grundiertes Schnittproblem besprochen werden.

2.1.2.1.5.1 Leerheit Ein Baumautomat ist leer, wenn die Sprache, die er erkennt, leer ist. Dies kann in linearer Zeit entschieden werden.

2.1.2.1.5.2 Endlichkeit Ein Baumautomat ist genau dann endlich, wenn die Sprache, die er erkennt, endlich ist. Dies ist in polynomieller Zeit entscheidbar.

2.1.2.1.5.3 Äquivalenz Zwei Baumautomaten sind genau dann äquivalent, wenn sie dieselbe Sprache erkennen. Dieses Problem ist bei zwei deterministischen Baumautomaten A_1 und A_2 für einen Algorithmus in $O(||A_1|| \times ||A_2||)$ entscheidbar, für nichtdeterministische brauchen wir einen exponentiellen Algorithmus.

2.1.2.2 Top-Down Baumautomaten

Top-Down Automaten starten an der Wurzel und bewegen sich nach unten, wobei sie auf ihrem Weg induktiv jeden Zustand mit einem Subterm verbinden, wodurch ein Term gebildet wird. Damit ist die Baumsprache $L(A)$, die von einem Top-Down Baumautomaten A erkannt wird, die Menge aller Grundterme t, für die ein Startzustand q in I existiert, so dass $q(t) \stackrel{A,*}{\rightarrow} t$ gilt.

Definition Ein nichtdeterministischer endlicher Top-Down Baumautomat über F ist ein Tupel $A = (Q, F, Q_f, \triangle)$ mit Q einer Menge von Zuständen (Zustände sind einstellige Symbole), $I \subseteq Q$ einer Menge von Startzuständen und $\triangle$ einer Menge von Abbildungsregeln vom folgenden Typ:
$q(f(x_1, ..., x_n)) \rightarrow f(q_1(x_1), ..., q_n(x_n))$, wobei $n \geq 0$, $f \in F_n$, $q, q_1, ..., q_n \in Q$, $x_1, ..., x_n \in \chi$. Wenn $n = 0$ ist, d. h. wenn das Symbol ein Konstantensymbol a ist, dann ist die Zustandsübergangsfunktion bei einem Top-Down NFTA von der Form $q(a) \rightarrow a$.

Die Ausdrucksstärke von Top-Down und Bottom-Up Baumautomaten ist gleich. Deterministische Top-Down Baumautomaten sind allerdings nicht so mächtig wie nichtdeterministische, d. h. es existiert eine erkennbare Baumsprache, die von keinem Top-Down DFTA akzeptiert wird, und darum sind sie auch nicht so mächtig wie Bottom-Up Baumautomaten (Baumeigenschaften, die von einem deterministischen Top-Down Baumautomat beschrieben werden, können nur von Wegeigenschaften abhängen.). Ein endlicher Top-Down Baumautomat $(Q, F, Q_f, \triangle)$ ist deterministisch, wenn es genau einen Startzustand gibt und keine zwei Regeln mit derselben linken Seite existieren.

2.1.3 Alternierende Automaten

Die Verifizierung von abstrakten Modellen ist auf die Gültigkeit von Formeln reduzierbar, die gewünschte Eigenschaften beschreiben. Insbesondere im Zusammenhang mit der Informatik interessiert die Verifizierung einer Formel. Dies trifft auf Datenbankanfrageauswertungen, FS Programmverifikation (siehe spätere Abschnitte) oder Wissensrepräsentation zu.

Obwohl die Gültigkeit und Verifizierung einer Formel eng miteinander zusammenhängen, sind algorithmische Lösungen oft nicht zusammenhängend. Für Temporallogiken ist aber eine einheitliche Lösung möglich und zwar durch die Nutzung alternierender Automaten.

Ursprünglich wurden in der Literatur nichtdeterministische Automaten zur Übersetzung von temporallogischen Formeln in Automaten genutzt. Diese Übersetzung weist zwei Nachteile auf: Die Übersetzung ist nicht trivial und es ergibt sich ein exponentieller

Größenzuwachs, so dass Algorithmen nicht effizient sind.

Temporallogiken sind Logiken, die sich mit der Beschreibung von der zeitlichen Ordnung von Ereignissen beschäftigen. Sie können sich beispielsweise zur Prüfung von nebenläufigen Programmen nutzen lassen.

FS Programme können als Übergangssystem modelliert werden. Daher können sie durch eine Menge von Booleschen Atomen beschrieben werden. Dies lässt sich mit Temporallogiken verbinden. Somit kann die Verifizierung eines bestimmten Programmverhaltens geschehen, indem das gewünschte Verhalten als temporallogische Formel spezifiziert wird und dann lediglich geprüft wird, ob das Programm ein Modell der Formel ist. Daher leitet sich der Ausdruck „Model Checking“ ab.

Diese Ausarbeitung basiert auf [?]. Ferner wurden [?], [?] und [?] herangezogen um das Thema auszuarbeiten.

2.1.3.1 Automatentheorie

2.1.3.1.1 Wörter und Bäume Sei Σ eine nichtleere endliche Zeichenmenge, d. h. ein Alphabet. Ein (un)endliches Wort ist eine (un)endliche Zeichenfolge aus Σ^* (bzw. Σ^ω). Ein (un)endlicher Baum t ist ein (un)endlicher zusammenhängender (gerichteter) Graph, mit einem Wurzelknoten ϵ, jeder andere Knoten hat einen eindeutigen Elternknoten. Die Stelligkeit $arity(v)$ eines Knotens v ist die Zahl der Kinder von v, die Tiefe $|v|$ ist der Abstand zur Wurzel ϵ.

Sei N eine Teilmenge positiver ganzer Zahlen (bzw. natürlicher Zahlen). Ein Baum τ über N ist eine Teilmenge von N^*, so dass wenn $v \cdot i \in \tau$ mit $v \in N^*$, $i \in N$, dann ist $v \in \tau$ und es gibt eine Kante von v nach $v \cdot i$ und für $i > 0$ gilt ebenfalls $v \cdot (i-1) \in \tau$. Man beachte, dass „$\cdot$“ die Konkatenation ausdrückt. Daher ist ϵ die Wurzel eines solchen Baumes.

Beispiel: $N = \{0, 1\}$, $\tau = \{\epsilon, 0, 1, 00, 01\}$

Für eine Untermenge $D \subseteq N$ heißt ein Baum τ über N ein D-Baum gdw. $\forall x \in \tau : arity(x) \in D$. Für eine einelementige Menge $D = \{k\}$ heißt τ uniform und wird als k-Baum bezeichnet. Ein Baum heißt blätterlos, wenn alle seine Knoten mindestens einen Kindknoten besitzen. Somit ist ein unendliches Wort ein blätterloser 1-Baum.

Ein (un)endlicher Zweig $\beta = v_0, v_1, ...$ ist eine maximale Knotenfolge, so dass $\forall i \geq 0 : v_i$ der Elternknoten von v_{i+1} ist, mit v_0 als Wurzel. Ein Σ-beschrifteter Baum ist ein Paar (τ, T) mit einem Baum τ und einer zugehörigen Abbildung $T : \tau \to \Sigma$, die jedem Knoten ein Zeichen aus einem endlichen Alphabet Σ zuweist. Ein Zweig von (τ, T) definiert ein Wort $T(\beta) = T(v_0), T(v_1), ...$ durch Anwendung von T.

2.1.3.1.2 Nichtdeterministische Automaten auf unendlichen Wörtern Ein nichtdeterministischer Büchiautomat (NBA) A auf (unendlichen) Wörtern ist ein Tupel $(\Sigma, Q, q_0, \delta, F)$, mit einer endlichen Zustandsmenge Q, einem Startzustand q_0, einer endlichen Menge akzeptierender Zustände F und einer Zustandsübergangsfunktion $\delta : Q \times \Sigma \to 2^Q$ (Nichtdeterminismus, da δ auf eine mögliche Zustandsübergangsmenge abbildet).

Ein Lauf r von A auf einem unendlichen Wort $w = a_0, a_1, ...$ über Σ ist eine Folge $q_0, q_k, ...$ (mit $k > 0$), so dass $\forall i \geq 0 : q_{i+1} \in \delta(q_i, a_i)$ gilt. Dann bezeichne $lim(r)$ die Menge aller Zustände, die in r unendlich oft auftreten. Da Q endlich ist, ist $lim(r)$ folglich nicht leer.

Ein Lauf r ist akzeptierend, wenn $lim(r) \cap F \neq \emptyset$. Ein unendliches Wort w wird von A akzeptiert, wenn es einen akzeptierenden Lauf r von A auf w gibt. Die Menge der unendlichen, vom Automaten A akzeptierten Wörter wird mit $L_\omega(A)$ bezeichnet.

Lemma 1. *NBAs sind gegenüber Schnittbildung abgeschlossen, diese ist in linearer Zeit durchführbar.*

Ob ein NBA A der Größe n eine nichtleere Sprache akzeptiert, ist in linearer Zeit mit Platzbedarf $O(log^2 n)$ entscheidbar. Dieses Problem ist wichtig, um zu entscheiden, ob ein Automat überhaupt interessant ist. Sonst gibt es einen äquivalenten Automaten, der kein Wort akzeptiert.

2.1.3.1.3 Alternierende Automaten auf unendlichen Wörtern Nichtdeterminismus gibt einer Maschine existenzielle Wahlmöglichkeit, ihr Komplement dagegen folglich eine universelle Wahlmöglichkeit. Daher macht die Betrachtung von Maschinen Sinn, die beide Wahlmöglichkeiten vereinen. Diese Maschinen heißen alternierend.

Für eine Menge X sei $B^+(X)$ eine Menge positiver Boolescher Formeln über X, d. h. Elemente aus X sowie $true/false$ induktiv verknüpft mit „$\wedge$“, „$\vee$“. Sei $Y \subseteq X$. Y erfüllt eine Formel $\Theta \in B^+(X)$, wenn Θ genau für die positive Belegung der Elemente aus Y erfüllt ist. D. h. Y ist ein positives Modell für Θ; es enthält keine negierten Ausdrücke, z.B. für $A \vee B$ ist $\{A\}$ ein positives Modell.

Sei A ein nichtdeterministischer Automat. Die Übergangsfunktion δ lässt sich als Formel ausdrücken (bzw. als Klauselmenge sehen): $\delta(q, a) = \{q_1, q_2, q_3\} \Leftrightarrow \delta(q, a) = q_1 \vee q_2 \vee q_3$

Ein alternierender Automat (ABA) A ist ähnelt von der Definition einem NBA, mit dem Unterschied, dass die Übergangsfunktion δ partiell auf $B^+(Q)$ abbildet. Bedingt durch die mögliche Universalwahl ist ein Lauf r eines ABA A auf einem unendlichen Wort $w = a_0, a_1, a_2, ...$ ein Q-beschrifteter Baum (ρ, r), mit Wurzel $r(\epsilon) = q_0$, so dass gilt: Wenn $|v| = i$, $r(v) = q$ und $\delta(q, a_i) = \Theta$ dann

- gibt es k Kinder $v_1, ..., v_k$ (mit $k \leq |Q|$) von v
- und $\{r(v_1), ..., r(v_k)\}$ erfüllt Θ.

Ein Lauf kann endliche Zweige besitzen. Für den Fall $\Theta = true$ kann v auch kinderlos sein. Jedoch $\Theta = false$ ist in jedem Fall unerfüllbar. Außerdem darf $\delta(q, a_i)$ nicht undefiniert sein, d. h. δ muss im Zustand q beim Lesen von a_i auf einen Folgezustand abbilden. Ein Lauf r ist akzeptierend, wenn alle unendlichen Zweige in r unendlich viele Beschriftungen aus F enthalten. D. h. ein Zweig trifft auf $\Theta = true$ oder unendlich oft auf einen akzeptierenden Zustand.

Lemma 2. *Ein ABA der Größe n kann in einen NBA der Größe $2^{O(n)}$ umgewandelt werden.*

Es folgt, dass der Leerheitstest für einen ABA in exponentieller Zeit bzw. mit quadratischem Platzbedarf entschieden werden kann.

2.1.3.1.4 Nichtdeterministische Automaten auf unendlichen Bäumen Ein nichtdeterministischer Automat auf (unendlichen) Bäumen (NTA) A ist identisch zu einem NBA, mit dem Unterschied, dass er um eine endliche Menge $D \subset \mathbb{N}$ von Stelligkeiten ergänzt wird und die Übergangsfunktion $\delta : Q \times \Sigma \times D \to 2^{Q*}$ ist.

Im Zustand q beim Lesen eines k-stelligen Knotens v wird aus einer Menge von k-Tupeln eines aus der Übergangsfunktion der Form $q_1, ..., q_k$ (nichtdeterministisch) gewählt, der Automat „kopiert sich selbst“ k mal und setzt seine Läufe in den Kindern von v in den jeweiligen Folgezuständen fort.

Ein Lauf r von A auf einem unendlichen Σ-beschrifteten D-Baum (τ, T) ist ein Q-beschrifteter D-Baum (τ, r) mit $r : \rho \to Q$, so dass $r(\epsilon) = q_0$ gilt, und für jeden Knoten v mit $arity(v) = k$ ist
$\langle r(v \cdot 1), r(v \cdot 2), ..., r(v \cdot k)\rangle \in \delta(r(v), T(v), k)$.

Ein Lauf ist akzeptierend, wenn $lim(r(\epsilon)) \cap F \neq \emptyset$ für jeden Zweig $\beta(= v_0, v_1, ...)$ von τ gilt. D. h. in jedem Zweig β kommt unendlich oft ein Knoten v_i vor, für den $r(v_i) \in F$ gilt. Die Menge der von A akzeptierten Bäum wird mit $T_\omega(A)$ bezeichnet.

Lemma 3. *Der Leerheitstest auf einem NTA ist in quadratischer Zeit entscheidbar.*

2.1.3.1.5 Alternierende Automaten auf unendlichen Bäumen Ein alternierender Automat auf (unendlichen) Bäumen (ATA) A ist identisch zu einem NTA mit einer anderen partiellen Übergangsfunktion $\delta : Q \times \Sigma \times D \to B^+(N \times Q)$.

Hierbei drückt $c \in N$ (zur Erinnerung $D \subseteq N$) eine Richtung (zu dem nächsten Kind) aus, in die ein Automat in einem Zustand $q \in Q$ beim Lesen eines Knotens fortfährt. So drückt $\delta(q_0, a, 2) = ((1, q_1) \wedge (2, q_2))$ aus, dass der Automat A am Anfang beim Lesen von a eine Kopie von A im Zustand q_1 in Richtung 1 und eine Kopie von A im Zustand q_2 in Richtung 2 startet.

Ein Lauf r eines ATA auf einem blätterlosen D-Baum $\langle \tau, T \rangle$ ist ein $(N^* \times Q)$-beschrifteter Baum (ρ, r), bei dem jedem Knoten p aus (ρ, r) nicht eindeutig ein Knoten v aus τ entspricht. Im Gegensatz dazu ist diese Zuordnung bei einem NTA eindeutig. Ein Knoten (v, q) in r entspricht dem Automaten, der den Knoten v aus τ im Zustand q liest. Ein Lauf $(\rho, r) := \langle \tau_r, T_r \rangle$ ist ein Σ_r-beschrifteter Baum (mit $\Sigma_r := (N^* \times Q)$), es gilt:

1. $T_r(\epsilon) = (\epsilon, q_0)$
2. Sei $v_r \in \tau_r$, $T_r(v_r) = (v, q)$, $arity(v) = k$, $\delta(q, T(v), k) = \Theta$.
   Dann gibt es eine Menge $M = \{(c_1, q_1), ..., (c_n, q_n)\} \subseteq \{1, ...k\} \times Q$ so dass:

   M erfüllt Θ

   $1 \leq i \leq n$, $(v_r \cdot i) \in \tau_r$, $T_r(v_r \cdot i) = (v \cdot c_i, q_i)$

Lemma 4. *Ein ATA der Größe n kann in einen NTA der Größe $2^{O(n)}$ umgewandelt werden.*

Lemma 5. *Somit ist der Leerheitstest für einen ATA in exponentieller Zeit entscheidbar.*

2.1.3.1.6 Reduktion des Leerheitstests Der Leerheitstest für NTAs ist reduzierbar auf denselben Test mit nur $\{a\}$ als Beschriftung. Ein NTA A wird in einen NTA A' überführt, man ersetzt $\Sigma := \{a\}$ und $\delta := \delta'$. Hierbei ist $\delta'(q, a, k) := \bigcup_{b \in \Sigma} \delta(q, b, k)$. A akzeptiert irgendeinen Baum (τ, T) gdw. A' einen a-beschrifteten Baum (τ', T) akzeptiert. D. h. A' errät eine Σ-Beschriftung und verfährt dann wie A.

Diese Reduktion ist nicht auf ATA anwendbar. (A' akzeptiert (τ, T)) $\Rightarrow$ (A akzeptiert (τ, T)) gilt nicht i. A., d.h. $L(A') \subseteq L(A)$. Denn zwei verschiedene Kopien von A, die auf demselben Teil von (τ, T) laufen, müssten immer die gleiche Beschriftung „erraten“.

Lemma 6. *Der Leerheitstest für einen ABA über einem einelementigen Alphabet ist in quadratischer Zeit entscheidbar. Dies ist äquivalent zu dem Leerheitstest eines ATA auf uniformen (blätterlosen) 1-Bäumen mit $\{a\}$ als Alphabet.*

2.1.3.2 Temporallogik und alternierende Automaten

Temporallogik ist ein Zweig der Modallogik. Man unterscheidet zwei Typen von Temporallogiken: lineare und verzweigende. In der linearen Temporallogik (LTL) hat jeder

zukünftige Zeitpunkt einer *Berechnung* eine eindeutige mögliche Zukunft. In der verzweigenden Temporallogik (CTL: computational tree logic) kann sich – vereinfacht dargestellt – ein Zeitpunkt dagegen verzweigen, es ist nicht mehr nur eine lineare Berechnung wie bei LTL.

Beide Typen lassen sich mit der Theorie von Automaten auf unendlichen Wörtern bzw. auf unendlichen Bäumen verbinden, die genau alle Berechnungen akzeptieren, für die eine Formel gilt.

2.1.3.2.1 Lineare Temporallogik (LTL) Eine LTL-Formel φ wird induktiv über einer Menge $Prop$ von Atomen gebildet. φ kann eine atomare Formel sein. Oder wenn φ' und φ'' LTL-Formeln sind, dann sind $\neg\varphi'$ und $(\varphi' \wedge \varphi'')$ sowie $X\varphi'$ und $\varphi' U \varphi''$ ebenfalls LTL-Formeln. X und U sind dabei unäre bzw. binäre Temporalkonnektoren. LTL-Formeln werden über Berechnungen interpretiert. Eine Berechnung ist eine Funktion $\pi : \omega \rightarrow 2^{Prop}$, die zu einem gegebenen Zeitpunkt den Atomen aus $Prop$ Wahrheitswerte zuordnet.

Eine LTL-Formel φ gilt in einer Berechnung π zum Zeitpunkt $i \in \omega$, $(\pi, i) \models \varphi$, gdw.:

- $(\pi, i) \models p$ mit $p \in Prop$ gdw. $p \in \pi(i)$
- $(\pi, i) \models \phi \wedge \psi$ gdw. $(\pi, i) \models \phi$ und $(\pi, i) \models \psi$
- $(\pi, i) \models \neg\phi$ gdw. $(\pi, i) \models \phi$ gilt nicht
- $(\pi, i) \models X\phi$ gdw. $(\pi, i+1) \models \phi$
- $(\pi, i) \models \phi U \psi$ gdw. $\exists j \geq i : (\pi, j) \models \psi$ und $\forall k : i \leq k \leq j : (\pi, k) \models \phi$

φ gilt in einer Berechnung π, notiert als $\pi \models \varphi$, gdw. $(\pi, 0) \models \varphi$ gilt. Ein Programm P über einer Menge von Atomen ist ein Tupel (W, w_0, R, V), mit einer Zustandsmenge W (z. B. Speicherabbild), einer Erreichbarkeitsrelation $R \subseteq W^2$ (daher Nichtdeterminismus) und einer Abbildung $V : W \rightarrow 2^{Prop}$, die den Zuständen Wahrheitswerte zuweist. Der Einfachheit halber wird R als total angenommen (aRb oder bRa gilt immer). Ein Programm P heißt Finite State (FS) Programm, wenn W endlich ist.

Für ein endliches W heißt P ein endliches Zustandsprogramm. Ein Pfad $u = w_0, w_1, ...$ in P ist eine Folge, so dass $\forall i \geq 0 : w_i R w_{i+1}$ gilt. Die Folge $V(w_0), V(w_1), ...$ ist eine Berechnung von P.

Eine Formel φ gilt in P, wenn φ in allen Berechnungen von P gilt. φ ist gültig, gdw. φ in allen Programmen P gilt. D. h. φ ist gültig, wenn φ in allen Berechnungen gilt. Also können Berechnungen als unendliche Wörter über dem Alphabet 2^{Prop} gesehen werden.

Satz 1. *Zu einer gegeben LTL-Formel φ lässt sich ein ABA A_ω konstruieren mit $\Sigma = 2^{Prop}$, $|Q| = O(|\varphi|)$ so dass $L_\omega(A_\varphi)$ genau die Menge der Berechnungen ist, in denen φ gilt.*

Beweis. Es sei Q die Menge aller Teilformeln von φ und deren Negationen, $q_0 := \varphi$ und F sei die Menge aller Formeln in Q der Form (pUq). Für die Übergangsfunktion δ gilt:

1. $\delta(p, a) = true$ gdw. $p \in a$ (d. h. $\delta(p, a) = false$ für $p \notin a$)
2. $\delta(\phi \wedge \psi, a) = \delta(\phi, a) \wedge \delta(\psi, a)$
3. $\delta(\neg\phi, a) = \overline{\delta(\phi, a)}$ (gemeint: die duale Formel)
4. $\delta(X\phi, a) = \phi$
5. $\delta(\phi U \psi, a) = \delta(\psi, a) \vee (\delta(\phi, a) \wedge (\phi U \psi))$

□

Korollar 1. *Zu einer gegebenen LTL-Formel φ lässt sich ein NBA A_ω konstruieren mit $\Sigma = 2^{Prop}$, $|Q| = 2^{O(|\varphi|)}$, so dass $L_\omega(A_\varphi)$ genau die Menge der Berechnungen ist, in denen φ gilt.*

2.1.3.3 Model Checking (Truth and Validity Checking)

In diesem Abschnitt werden nun bisherige Ergebnisse verwendet, um die Gültigkeit und Korrektheit einer temporallogischen Formel entscheiden zu können.

2.1.3.3.1 LTL Eine LTL-Formel φ ist gültig

- $\Leftrightarrow$ alle Berechnungen von A_φ akzeptiert werden.
- $\Leftrightarrow L_\omega(A_\varphi) = \Sigma^\omega$ (mit $\Sigma = 2^{Prop}$)
- $\Leftrightarrow \Sigma^\omega - L_\omega(A_\varphi) = \emptyset$
- $\Leftrightarrow L_\omega(A_{\neg\varphi}) = \emptyset$

Somit kann die Gültigkeitsprüfung einer Formel auf den Leerheitstest reduziert werden.

Satz 2. *Die Gültigkeit einer gegebenen LTL-Formel φ kann in $O(2^{O(|\varphi|)})$ mit Platzbedarf $O(|\varphi|^2)$ entschieden werden.*

Seien ein FS Programm P und eine LTL-Formel φ gegeben, dann kann man wie folgt entscheiden, ob φ in P gilt. P kann als NBA A_P gesehen werden mit $Q := W$, $q_0 := w_0$, $F := V$, $\Sigma := 2^{Prop}$ und $w' \in \delta(w, a)$ gdw. wRw', $a = V(w)$. Dann ist $L_\omega(A_P)$ die Menge aller Berechnungen von P.

Der Korrektheitstest (d. h. der Test, ob eine Formel bei einem gegebenen, unterliegenden Programm erfüllt ist) lässt sich nun reduzieren. φ gilt in P wenn $L_\omega(A_P) \subseteq L_\omega(A_\varphi)$ gilt. D. h.:

$$\boxed{L_\omega(A_P) \subseteq L_\omega(A_\varphi) \Leftrightarrow L_\omega(A_P) \cap L_\omega(\overline{A_\varphi}) = \emptyset \Leftrightarrow L_\omega(A_P) \subseteq L_\omega(A_{\neg\varphi}) = \emptyset}$$

Satz 3. *Die Korrektheit einer LTL-Formel φ in einem FS Programm P kann in $O(|P| \cdot 2^{O(|\varphi|)})$ mit Platzbedarf $O((|\varphi| + log|P|)^2)$ entschieden werden.*

2.2 Allgemeine Analyse

2.2.1 Erreichbarkeitsanalyse für Pushdown-Systeme

Pushdown-Systeme sind Pushdown-Automaten ohne Eingabe. Ihre (nichtdeterministischen) Übergänge hängen nur vom obersten Symbol auf dem Stack und dem aktuellen Kontrollpunkt ab. Diese Art von Automat verfügt über einen unendlichen Raum von Zuständen (bei Pushdown-Systemen als Konfigurationen bezeichnet), weil der Stack beliebig wachsen kann. Pushdown-Systeme eignen sich sehr gut zur Modellierung von Programmen, denn sie repräsentieren nicht nur die Übergänge zwischen den endlich vielen Belegungen der globalen Variablen, sondern auch den für rekursive und nicht-rekursive Prozeduraufrufe verwendeten Stack des Programms, der zumindest theoretisch beliebig wachsen kann. Offensichtlich können schon recht einfache Programme einen unendlichen Zustandsraum haben, wodurch eine Modellierung durch endliche Automaten inadäquat wird. Die hier vorgestellten Techniken zur Berechnung von Vorgängern ermöglichen eine Erreichbarkeitsanalyse für Pushdown-Systeme. Ein grundlegendes Erreichbarkeitsproblem ist es z. B., zu entscheiden, ob ein Pushdown-System gemäß seiner Transitionsregeln von einer Konfiguration s in eine Konfiguration t übergehen kann. Die Erreichbarkeitsanalyse führt zu neuen Algorithmen für das Model Checking von Pushdown-Systemen und zu neuen Datenflussanalyse-Algorithmen für Flussgraphen. Die hier dargestellten Verfahren wurden [**?**] entnommen. Dort werden auch die Anwendungsmöglichkeiten der Erreichbarkeitsanalyse beim Model Checking detailliert vorgestellt. Weitere Anwendungen sind in [**?**] zu finden.

2.2.1.1 Pushdown-Systeme

Ein Pushdown-System (PDS) ist ein Tripel $\mathcal{P} = (P, \Gamma, \Delta)$, wobei P eine endliche Menge von Kontrollpunkten, Γ ein endliches Stack-Alphabet und $\Delta \subseteq (P \times \Gamma) \times (P \times \Gamma^*)$ eine endliche Menge von Transitionsregeln ist. Eine Konfiguration von $\mathcal{P}$ ist ein Paar $\langle p, w \rangle$ mit $p \in P$ und $w \in \Gamma^*$. Die Semantik einer Konfiguration $\langle p, w \rangle$ ist, dass sich $\mathcal{P}$ aktuell an Kontrollpunkt p befindet und den Stackinhalt $w = \gamma_1 \cdots \gamma_k$ hat, wobei γ_1 an der Spitze liegt. Für $((q, \gamma), (q', w)) \in \Delta$ schreiben wir $(q, \gamma) \hookrightarrow (q', w)$. Eine solche Regel ermöglicht für alle $w' \in \Gamma^*$ den Übergang von der Konfiguration $\langle q, \gamma w' \rangle$ in die Konfiguration $\langle q, ww' \rangle$. Das PDS geht also von q nach q' über, nimmt γ vom Stack und schreibt das Wort w wieder darauf. Nichtdeterminismus ist durch diese Definition nicht ausgeschlossen.

Die Erreichbarkeitsrelation $\Rightarrow$ sei der reflexive und transitive Abschluss der Direkter-Nachfolger-Relation. Ein Lauf von $\mathcal{P}$ ist eine endliche Sequenz $c_1 c_2 \cdots c_k$ von Konfigura-

tionen c_i von $\mathcal{P}$, wobei für $i = 1, \cdots, k-1$ die Konfiguration c_{i+1} direkter Nachfolger von c_i ist und auf c_k keine Regel mehr anwendbar ist (für oberstes Zeichen gibt es keine Regel, oder der Stack ist leer).

2.2.1.2 Das Problem der Vorgänger-Berechnung für Pushdown-Systeme

Für alle Mengen $C \subseteq P \times \Gamma^*$ von Konfigurationen von $\mathcal{P}$ werden folgende Vorgängerfunktionen definiert:

- $pre_{\mathcal{P}}(C) = \{c \in P \times \Gamma^* \mid \exists c' \in C : c' \, ist \, direkter \, Nachfolger \, von \, c\}$
- $pre^*_{\mathcal{P}}$ = reflexiver und transitiver Abschluss von $pre_{\mathcal{P}}$
- $pre^+_{\mathcal{P}} = pre_{\mathcal{P}} \circ pre^*_{\mathcal{P}}$

$pre_{\mathcal{P}}(C)$ ist die Menge aller Konfigurationen von $\mathcal{P}$, von denen aus in genau einem Schritt eine Konfiguration $c \in C$ erreichbar ist. Im Unterschied dazu enthält $pre^*_{\mathcal{P}}(C)$ genau diejenigen Konfigurationen von $\mathcal{P}$, von denen aus in einer beliebigen Zahl (auch 0) von Schritten eine Konfiguration $c \in C$ erreichbar ist. Für $pre^+_{\mathcal{P}}(C)$ gilt ähnliches, jedoch muss mindestens ein Schritt gemacht werden.

Das Problem der Vorgänger-Berechnung besteht nun darin, für ein gegebenes Pushdown-System $\mathcal{P}$ und eine beliebige Menge C von Konfigurationen von $\mathcal{P}$ die Menge $pre^*_{\mathcal{P}}(C)$ zu berechnen. Ein naiver Ansatz besteht darin, $X := C$ zu setzen und solange die Anweisung $X := X \cup pre_{\mathcal{P}}(X)$ auszuführen, bis X stationär wird. Dies setzt einen Algorithmus für $pre_{\mathcal{P}}$ voraus. Nach Terminierung des Algorithmus gilt offensichtlich $X = pre^*_{\mathcal{P}}(C)$. Das Problem ist jedoch, dass der Algorithmus divergieren kann. Dazu betrachte man das PDS $\mathcal{P} = (P, \Gamma, \Delta)$ mit $P = \{p\}$ und $\Gamma = \{\gamma\}$ sowie $\Delta = \{((p, \gamma), (p, \varepsilon))\}$. Wir wenden den naiven Algorithmus auf $C = \{\langle p, \varepsilon\rangle\}$ an. Wegen $pre_{\mathcal{P}}(C) = \{\langle p, \gamma\rangle\}$ wird in der ersten Iteration $X := \{\langle p, \varepsilon\rangle, \langle p, \gamma\rangle\}$ gesetzt. Da $pre_{\mathcal{P}}(\{\langle p, \varepsilon\rangle, \langle p, \gamma\rangle\}) = \{\langle p, \gamma\rangle, \langle p, \gamma\gamma\rangle\}$ ist, wird in der zweiten Iteration $X := \{\langle p, \varepsilon\rangle, \langle p, \gamma\rangle, \langle p, \gamma\gamma\rangle\}$ berechnet. Offensichtlich wird für alle $i \geq 1$ in der i-ten Iteration $X := \{\langle p, \gamma^j\rangle \mid 0 \leq j \leq i\}$ berechnet, X wird also niemals stationär. Eine weitere Schwierigkeit ist, dass X mit unendlichen Mengen belegt werden kann (C kann schon unendlich sein), was sofort die Frage nach der Repräsentation im Speicher aufwirft. Der hier zu entwickelnde Algorithmus zur Vorgänger-Berechnung wird ebenso wie der naive Algorithmus Folgen von Vorgänger-Mengen berechnen, dabei jedoch dank größerer Schritte stets nach endlich vielen Schritten $pre^*_{\mathcal{P}}(C)$ erreichen. Auch dieser

Algorithmus verarbeitet potentiell unendliche Mengen. Wir benötigen also eine Datenstruktur zur endlichen Repräsentation unendlicher Mengen. Weil z. B. Algorithmen für das Model Checking oft mit den Mengen rechnen, sollte die Datenstruktur abgeschlossen sein unter den Operationen Schnitt, Vereinigung und Komplement und diese Operationen auch möglichst effizient, aber unbedingt mit polynomiellem Rechenaufwand ausführen können. Natürlich sollte es auch in polynomieller Zeit entscheidbar sein, ob eine gegebene Konfiguration Element der repräsentierten Menge ist. Schliesslich fordern wir noch die Abgeschlossenheit unter $pre^*_{\mathcal{P}}$, um die Vorgänger-Berechnung immer durchführen zu können. Die von uns gewählte Datenstruktur wird all diese Anforderungen erfüllen, wird aber nicht in der Lage sein, beliebige Teilmengen des Konfigurationsraums darzustellen.

2.2.1.3 Multi-Automaten (MA)

Die endliche effiziente Repräsentation unendlicher Konfigurationsmengen wird hier durch sog. Multi-Automaten realisiert.

Sei $\mathcal{P} = (P, \Gamma, \Delta)$ ein Pushdown-System mit $P = \{p^1, \cdots, p^m\}$. Ein $\mathcal{P}$-Multi-Automat ($\mathcal{P}$-MA) ist ein Tupel $\mathcal{A} = (\Gamma, Q, \delta, I, F)$, wobei Q eine endliche Menge von Zuständen ist, $\delta \subseteq Q \times \Gamma \times Q$ eine Menge von Transitionen ist, $I = \{s^1, \cdots, s^m\} \subseteq Q$ die initialen Zustände von $\mathcal{A}$ definiert und $F \subseteq Q$ die Menge der akzeptierenden Zustände ist. Auch hier ist Nichtdeterminismus erlaubt. Die Transitionsrelation $\rightarrow \subseteq Q \times \Gamma^* \times Q$ ist die kleinste Relation mit

- wenn $(q, \gamma, q') \in \delta$, dann $q \xrightarrow{\gamma} q'$
- $q \xrightarrow{\varepsilon} q$ für alle $q \in Q$
- $q \xrightarrow{w} q''$ und $q'' \xrightarrow{\gamma} q'$ dann $q \xrightarrow{w\gamma} q'$

Für $w = \gamma_1 \cdots \gamma_k \in \Gamma^*$ heißt $s^i \xrightarrow{\gamma_1} q_1 \cdots \xrightarrow{\gamma_k} q_k$ w-Lauf von $\mathcal{A}$. $\mathcal{A}$ akzeptiert $\langle p^i, w \rangle$ genau dann wenn $s^i \xrightarrow{w} q$ für ein $q \in F$. Es sei $Conf(\mathcal{A})$ die Menge der von $\mathcal{A}$ akzeptierten Konfigurationen.

2.2.1.4 Vorgänger-Berechnung für Pushdown-Systeme

Seien ein PDS $\mathcal{P} = (P, \Gamma, \Delta)$ und eine reguläre Menge $C \subseteq P \times \Gamma^*$ von Konfigurationen gegeben. Weil C regulär ist, gibt es einen $\mathcal{P}$-MA $\mathcal{A} = (\Gamma, Q, \delta, I, F)$ mit $Conf(\mathcal{A}) = C$. $\mathcal{A}$ enthalte o. B. d. A. keine Transitionen nach I. Der nicht-naive Algorithmus zur Berechnung von $pre^*_{\mathcal{P}}(C)$ konstruiert wie der naive Algorithmus eine Folge $Y = (Y_i)_{i \geq 0}$ von

Konfigurationsmengen. Die durch den naiven Algorithmus erzeugten Zwischenergebnisse seien durch die Folge $X = (X_i)_{i \geq 0}$ bezeichnet ($X_0 = C$). Dann hat Y die folgenden Eigenschaften:

- (P1). $\exists i \geq 0 : Y_{i+1} = Y_i$
- (P2). $\forall i \geq 0 : X_i \subseteq Y_i$
- (P3). $\forall i \geq 0 : Y_i \subseteq \bigcup_{j \geq 0} X_j$

(P1) sichert die Terminierung des Algorithmus. Für alle i hängt Y_{i+1} nur von Y_i ab. Falls $Y_{i+1} = Y_i$ ist, muss also für alle $j \geq i$ sogar die Gleichung $Y_j = Y_i$ gelten. Dies bedeutet, dass die Folge nach einer endlichen Zahl von Schritten stationär wird. Wegen (P2) werden keine Elemente vergessen, die der naive Algorithmus berücksichtigen würde. Oder anders ausgedrückt enthalten sowohl X_i als auch Y_i alle Vorgänger der Menge C, die (gemessen am kürzesten Weg) nicht mehr als i Schritte von der Menge C entfernt sind. Allerdings kann Y_i noch weitere Elemente enthalten, nämlich vorweggenommene Elemente, die in der X-Folge erst später vorkommen. Während X_{i+1} nur endlich viele Elemente mehr als X_i enthalten könnte, könnte Y_{i+1} durch Hinzufügung unendlich vieler Elemente zu Y_i entstehen, wodurch die unendliche Menge $pre^*_{\mathcal{P}}(C)$ erreichbar wird. Oder beispielsweise sind alle auftretenden Mengen endlich, aber Y erfordert deutlich weniger Iterationen als X. Dass Y_i wirklich höchstens Elemente vorwegnimmt statt Nicht-Elemente von $pre^*_{\mathcal{P}}(C)$ einzuführen, folgt aus Eigenschaft (P3). Man beachte, dass $pre^*_{\mathcal{P}}(C) = \bigcup_{j \geq 0} X_j$ gilt. Wenn der Algorithmus nach $k+1$ Iterationen terminiert, dann wissen wir, dass die Folge ab dem Glied Y_k stationär wird. Für alle $j < k$ ist wegen (P2) $X_j \subseteq Y_j \subseteq Y_k$. Für alle $j \geq k$ ist $X_j \subseteq Y_j = Y_k$. Also enthält Y_k alle X_i und damit ist $pre^*_{\mathcal{P}}(C) \subseteq Y_k$ gezeigt. Insgesamt folgt $pre^*_{\mathcal{P}}(C) = Y_k$.

Die Definition von Y basiert auf einer Folge $(\mathcal{A}_i)_{i \geq 0}$ von Automaten, deren Glieder sich nur in der Transitionsmenge unterscheiden. Es seien

$$\delta_0 := \delta$$

und

$$\delta_{i+1} := \delta_i \cup \left\{ s^j \xrightarrow{\gamma}_{i+1} q \mid (p^j, \gamma) \hookrightarrow (p^k, w) \, und \, s_k \xrightarrow{w}_i q \right\}$$

für alle $i \geq 0$. Es seien nun $\mathcal{A}_i := (\Gamma, Q, \delta_i, I, F)$ und $Y_i := Conf(\mathcal{A}_i)$ für alle $i \geq 0$.

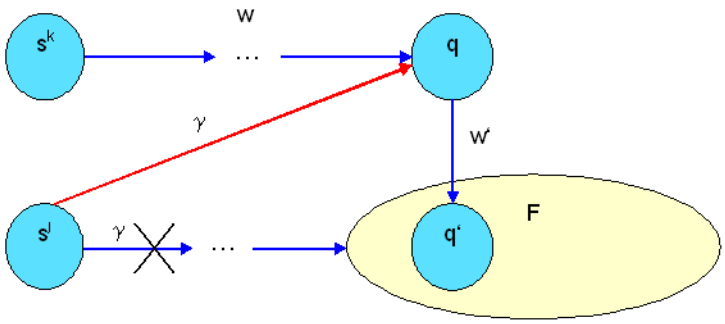


Abbildung 2.6: Funktionsweise der Y-Konstruktion

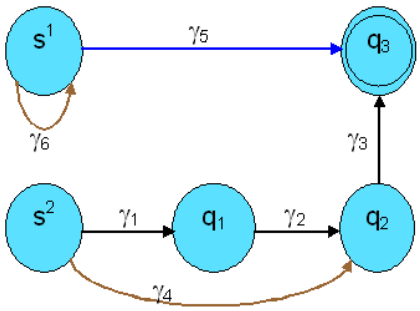


Abbildung 2.7: Beispiel für die Y-Konstruktion

Es gilt $Y_i \subseteq Y_{i+1}$ für alle $i \geq 0$. Wegen $|\delta_i| \leq |Q|^2 |\Gamma|$ ist (P1) erfüllt. Jede im Laufe des Verfahrens hinzugefügte Transition startet an einem initialen Zustand. Abbildung 2.6 soll die Funktionsweise der Konstruktion veranschaulichen.

In Abbildung 2.6 visualisieren mit Ausnahme des Pfeils von s^j nach q alle abgebildeten Komponenten Bestandteile des Automaten $\mathcal{A}_i$ für ein i. Das zugehörige PDS enthalte die Regel $(p^j, \gamma) \hookrightarrow (p^k, w)$. Es ist $\langle p^k, ww' \rangle \in Conf(\mathcal{A}_i)$, denn es gibt einen ww'-Lauf von s^k nach $q' \in F$. Für das PDS ist die Konfiguration $\langle p^j, \gamma w' \rangle$ ein direkter Vorgänger von $\langle p^k, ww' \rangle$. Jedoch ist $\langle p^j, \gamma w' \rangle \notin Conf(\mathcal{A}_i)$, weil in s^i für kein $v \in \Gamma^*$ ein γv-Lauf beginnt, der in die Menge F führt. Per Definition enthält $\mathcal{A}_{i+1}$ auch die durch den von s^j nach q verlaufenden Pfeil ausgedrückte Transition. Aus diesem Grund gibt es in $\mathcal{A}_{i+1}$ einen $\gamma w'$-Lauf von s^i nach q, und es ist $\langle p^j, \gamma w' \rangle \in Conf(\mathcal{A}_{i+1})$.

Als Beispiel betrachte man das PDS $\mathcal{P} = (P, \Gamma, \Delta)$ mit $P = \{p^1, p^2\}$, $\Gamma = \{\gamma_1, \gamma_2, \gamma_3, \gamma_4, \gamma_5, \gamma_6\}$ und $\Delta = \{(p^2, \gamma_4) \hookrightarrow (p^2, \gamma_1\gamma_2), (p^1, \gamma_5) \hookrightarrow (p^2, \gamma_4\gamma_3), (p^1, \gamma_6) \hookrightarrow (p^1, \varepsilon)\}$. Es soll $pre^*_{\mathcal{P}}(\{\langle p^2, \gamma_1\gamma_2\gamma_3 \rangle\})$ berechnet werden. Der in Abbildung 2.7 dargestellte $\mathcal{P}$-MA $\mathcal{A}$ erkennt genau $\{\langle p^2, \gamma_1\gamma_2\gamma_3 \rangle\}$, wobei die Transitionsmenge durch die schwarzen Pfeile (Beschriftungen γ_1, γ_2, γ_3) gegeben ist. Zunächst wird Y_0 mit $Conf(\mathcal{A}_0) = \{\langle p^1, \gamma_1\gamma_2\gamma_3 \rangle\}$ in-

itialisiert. Danach wird jede PDS-Regel einzeln überprüft. Die Regel $(p^2, \gamma_4) \hookrightarrow (p^2, \gamma_1\gamma_2)$ führt zu einer neuen Transition, da ein $\gamma_1\gamma_2$-Lauf von s^2 nach q_2 existiert. $\mathcal{A}_1$ erhält also die zusätzliche Transition (s^2, γ_4, q_2). Von s^2 startet in $\mathcal{A}_0$ nur dieser eine $\gamma_1\gamma_2$-Lauf, also kann auf Basis dieser PDS-Regel keine weitere Transition eingefügt werden. Die zweite Regel ist offensichtlich noch nicht anwendbar. Es gibt aber einen ε-Lauf von s_1 nach s_1, die dritte Regel führt also zur Transition (s^1, γ_6, s^1). Also ist $\mathcal{A}_1$ der Automat, der aus den schwarzen und aus den braunen Transitionen besteht (Beschriftungen γ_1, γ_2, γ_3, γ_4, γ_6). Es gilt $Y_1 = \{\langle p^2, \gamma_1\gamma_2\gamma_3\rangle, \langle p^2, \gamma_4\gamma_3\rangle\} = X_1$. Dadurch entsteht nur für die zweite Regel eine neue Situation. Da es jetzt einen $\gamma_4\gamma_3$-Lauf von s^2 nach q_3 gibt, wird die Regel (s^1, γ_5, q_3) hinzugefügt. Durch Hinzunahme des blauen Pfeils (Beschriftung γ_5) gelangt man daher zu $\mathcal{A}_2$. Es ist $Y_2 = \{\langle p^2, \gamma_1\gamma_2\gamma_3\rangle, \langle p^2, \gamma_4\gamma_3\rangle\} \cup \{p^1\} \times \gamma_6^*\gamma_5 = pre^*_{\mathcal{P}}(\{\langle p^1, \gamma_1\gamma_2\gamma_3\rangle\})$. Y wird ab jetzt stationär, X dagegen nie. Es ist $X_i = \{\langle p^2, \gamma_1\gamma_2\gamma_3\rangle, \langle p^2, \gamma_4\gamma_3\rangle\} \cup \left\{\left\langle p^1, \gamma_6^j\gamma_5\right\rangle \mid 0 \leq j \leq i-2\right\}$ für alle $i \geq 2$.

2.2.1.5 Algorithmen für das Model Checking

Die oben vorgestellte Technik erlaubt den Entwurf eines neuen Algorithmus für das Model Checking von Pushdown-Systemen mit Linear-Time-Temporallogiken. Der Algorithmus konstruiert einen Büchiautomaten und reduziert das Problem auf akzeptierende Läufe. Das Problem der akzeptierenden Läufe wird mit Hilfe der Vorgänger-Berechnung für PDS gelöst. Als Ergebnis erhält man die Menge aller Konfigurationen, die die gegebene Formel erfüllen. Um auch neue Algorithmen für das Model Checking von Pushdown-Systemen mit Branching-Time-Temporallogiken entwerfen zu können, wird eine Verallgemeinerung der vorgestellten Vorgänger-Berechnung auf Alternierende Pushdown-Systeme (APDS) und Alternierende Multi-Automaten (AMA) vorgenommen. Dem gegebenen PDS wird dann ein passender APDS zugeordnet, und sämtliche Mengen werden durch AMA dargestellt. Je nach verwendeter Logik sind völlig verschiedene Algorithmen notwendig, die aber alle von einer polynomiellen Zahl von pre^*-Berechnungen für das APDS Gebrauch machen.

2.2.1.6 Alternierende Pushdown-Systeme (APDS)

Ein Alternierendes Pushdown-System (APDS) ist wieder ein Tripel $\mathcal{P} = (P, \Gamma, \Delta)$, wobei P und Γ wie bei PDS definiert sind. Δ ist aber hier eine Funktion mit dem Definitionsbereich $P \times \Gamma$. Die Menge der negationsfreien Booleschen Formeln über $P \times \Gamma^*$ bildet den Wertebereich von Δ. Mit dieser Definition lässt sich noch nicht viel machen. Daher wird eine alternative Modellierung von Δ entwickelt. Die Formeln sind alle o. B. d. A. in disjunktiver Normalform. Statt

$$\Delta(p,\gamma) = \bigvee_{1\leq i\leq m} \bigwedge_{1\leq j\leq k_i} \langle p_{i,j}, w_{i,j}\rangle$$

wird

$$\{((p,\gamma),\{(p_{1,j},w_{1,j}) \mid 1\leq j\leq k_1\}),\cdots,((p,\gamma),\{(p_{m,j},w_{m,j}) \mid 1\leq j\leq k_m\})\} \subseteq \Delta$$

modelliert. Dann ist $\Delta \subseteq (P\times\Gamma)\times 2^{P\times\Gamma^*}$ und im Allgemeinen keine Funktion mehr. Für die letzte Zeile schreibt man auch

$$(p,\gamma) \hookrightarrow \{(p_{1,j},w_{1,j}) \mid 1\leq j\leq k_1\},\ldots,(p,\gamma) \hookrightarrow \{(p_{m,j},w_{m,j}) \mid 1\leq j\leq k_m\}$$

Eine Regel $(p,\gamma) \hookrightarrow \{(p_1,w_1),\cdots,(p_n,w_n)\}$ ermöglicht den gleichzeitigen, parallelen Übergang des APDS von $\langle p,\gamma w\rangle$ in die Konfigurationen $\langle p_1,w_1w\rangle,\ldots,\langle p_n,w_nw\rangle$ für alle $w\in\Gamma^*$. Der APDS wählt in $\langle p,\gamma w\rangle$ nichtdeterministisch eine Regel mit linker Seite (p,γ) aus und führt dann die durch die rechte Seite spezifizierte parallele Verzweigung aus. Sind alle rechten Seiten der Regeln einelementige Mengen, so ist das APDS weitgehend ein PDS.

Ein Lauf von $\mathcal{P}$ für eine initiale Konfiguration c ist ein Baum mit Konfigurationen als Knoten und c als Wurzel. Die Vater-Kind-Beziehungen ergeben sich aus den jeweils angewendeten Regeln. In der oben beschriebenen Situation hat $\langle p,\gamma w\rangle$ die Kinder $\langle p_1,w_1w\rangle,\ldots,\langle p_n,w_nw\rangle$. Wenn $\langle p,\gamma w\rangle$ noch Geschwister hat, dann können sich diese auch entsprechend verzweigen. Jeder Knoten kann sich entsprechend der Regeln weiterverzweigen.

Die Erreichbarkeitsrelation $\Rightarrow\subseteq (P\times\Gamma^*)\times 2^{P\times\Gamma^*}$ ist die kleinste Relation mit

- $c\Rightarrow\{c\}$ für alle $c\in P\times\Gamma^*$
- ist c ein direkter Vorgänger von C ist, so $c\Rightarrow C$
- wenn $c\Rightarrow\{c_1,\ldots,c_n\}$ und $c_i\Rightarrow C_i$ für $1\leq i\leq n$, dann $c\Rightarrow(C_1\cup\cdots\cup C_n)$

Für alle Mengen C von Konfigurationen von $\mathcal{P}$ definieren wir wieder die Vorgängerfunktionen

- $pre_{\mathcal{P}}(C) = \{c\in P\times\Gamma^* \mid \exists C'\subseteq C : C' \,ist\, direkter\, Nachfolger\, von\, c\}$

- $pre^*_{\mathcal{P}}$ = reflexiver und transitiver Abschluss von $pre_{\mathcal{P}}$
- $pre^+_{\mathcal{P}} = pre_{\mathcal{P}} \circ pre^*_{\mathcal{P}}$

2.2.1.7 Alternierende Multi-Automaten (AMA)

Sei $\mathcal{P} = (P, \Gamma, \Delta)$ ein Alternierendes Pushdown-System. Ein $\mathcal{P}$-AMA ist ein Tupel $\mathcal{A} = (\Gamma, Q, \delta, I, F)$, wobei Γ, Q, I und F wie bei Multi-Automaten definiert sind. δ ist eine Funktion mit dem Definitionsbereich $Q \times \Gamma$ und der Menge der negationsfreien Booleschen Formeln über Q als Wertebereich. δ lässt sich analog zu Δ als Relation $\delta \subseteq (Q \times \Gamma) \times 2^Q$ modellieren. Auch hier finden parallele Übergänge statt.

Die Transitionsrelation $\rightarrow \subseteq Q \times \Gamma^* \times 2^Q$ ist die kleinste Relation mit

- $q \xrightarrow{\varepsilon} \{q\}$ für alle $q \in Q$
- wenn $(q, \Gamma, Q') \in \delta$, dann $q \xrightarrow{\gamma} Q'$
- wenn $q \xrightarrow{w} \{q_1, \ldots, q_n\}$ und $q_i \xrightarrow{\gamma} Q_i$ für alle $1 \leq i \leq n$, so $q \xrightarrow{w\gamma} (Q_1 \cup \cdots \cup Q_n)$

$\mathcal{A}$ akzeptiert $\langle p^i, w\rangle$ gdw. $s^i \xrightarrow{w} Q'$ für ein $Q' \subseteq F$. Ein Lauf von $\mathcal{A}$ über $w \in \Gamma^*$, beginnend bei $q \in Q$ ist ein endlicher Baum mit Zuständen als Knoten, wobei q die Wurzel ist. Die Vater-Kind-Beziehungen ergeben sich aus den angewendeten Regeln. Ein Abstieg von einer Ebene zur nächsttieferen entspricht genau einem Schritt. Die Kanten sind mit den passenden Zeichen beschriftet. Alle Blätter haben dieselbe Tiefe. Sämtliche von derselben Ebene ausgehenden Kanten tragen auch dieselbe Beschriftung. Jeder Pfad von der Wurzel zu einem Blatt ist mit w beschriftet. Der Lauf ist genau dann akzeptierend, wenn alle Blätter Elemente von F sind.

2.2.1.8 Vorgänger-Berechnung für Alternierende Pushdown-Systeme

Das Verfahren läuft fast genauso ab wie das für PDS beschriebene. Die induktive Konstruktion der Folge Y basiert dabei auf der Regel

$$\delta_{i+1} := \delta_i \cup \left\{ s^j \xrightarrow{\gamma}_{i+1} (P_1 \cup \cdots \cup P_m) \mid (p^j, \gamma) \hookrightarrow \left\{(p^{k_1}, w_1), \cdots, (p^{k_m}, w_m)\right\}, s^{k_1} \xrightarrow{w_1} P_1, \ldots, s^{k_m} \xrightarrow{w_m} P_m \right\}$$

2.2.2 Statische Programmanalyse

Das Ziel der statischen Programmanalyse ist es, Wertebereiche von Variablen oder das Verhalten eines Programms zur Laufzeit approximativ zu berechnen. Die Programmanalyse findet zur Compile-Zeit statt, weshalb sie „statisch“ genannt wird. Ihr Ziel ist es, die Voraussagen so präzise wie möglich und so sicher wie nötig zu treffen. Diese approximativ berechneten Voraussagen sollten allerdings so aussagekräftig sein, dass mit ihrer Hilfe zur Compile-Zeit nützliche Entscheidungen getroffen werden können. Z. B. ist es nützlich zu wissen, dass eine Variable nur Werte zwischen 0 und 120 annimmt, um ihr den Typ Byte statt Integer zuzuordnen; oder im Fall einer getypten Sprache, sicher zu sein, dass diese Variable nicht überlaufen wird. Ein Anwendungsgebiet der Programmanalyse ist es, Redundanzen während der Compile-Zeit zu vermeiden, also z. B. Schleifen-Invarianten außerhalb der Schleife auszuführen, den Variablen möglichst kleine Wertebereiche zuzuordnen oder überflüssige Berechnungen zu erkennen, die nie benutzt werden oder schon zur Compile-Zeit feststehen. Außer im Bereich der Codeoptimierung findet die Programmanalyse auch neuerdings Anwendung in der Softwareverifikation. Hier ist ihre Aufgabe, unerwünschtes Programmverhalten zu erkennen und zu vermeiden (wie z. B. das Überlaufen einer Variable).

2.2.2.1 Reaching Definitions Analysis

Im Folgenden wird ein Programmanalyseverfahren beispielhaft vorgestellt. Der Text basiert auf dem ersten Kapitel des Buches “Principles of Program Analysis“ [**?**]. Bei der *Reaching Definitions Analysis* ist man an den vollständigen Erreichbarkeits-Informationen von Variablenzuweisungen interessiert, um Aussagen über den Programmverlauf zu machen. Erreichbarkeits-Informationen geben Auskunft darüber, an welchem Programmpunkt eine Variable zum letzten Mal bzgl. eines anderen Programmpunktes zugewiesen worden sein könnte.

Wie bei den meisten rekursiven Datenfluss-Problemen ist es nicht möglich zu entscheiden, ob eine Zuweisung einen bestimmten Programmpunkt tatsächlich erreicht. Es ist nur möglich, Aussagen darüber zu treffen, ob eine Zuweisung einen bestimmten Programmpunkt erreichen *kann*. Ob dies tatsächlich der Fall ist, hängt im Allgemeinen von der Eingabe ab.

Variablenzuweisungen haben üblicherweise die Form `x:=a`. Im Folgenden wird ihnen noch ein Label verliehen, um besser auf sie referenzieren und mit ihnen rechnen zu können: $[\texttt{x:=a}]^l$. Das Label l ist willkürlich, wird aber üblicherweise von Anfang bis zum Ende des Programms fortlaufend vergeben.

Bei der *Reaching Definitions Analysis* ist man an den so genannten *entry* und *exit* Mengen eines Programmblocks interessiert.

Die *entry* bzw. *exit* Menge soll die Label aller Zuweisungen enthalten, die diesen Programmblock erreichen *können*. *entry* und *exit* Menge eines Programmblocks unterscheiden sich darin, dass *entry* die Label der Zuweisungen enthält, welche den Programmpunkt vor dem Betreten erreichen können und *exit* die Label der Zuweisungen, welche den Programmpunkt beim Verlassen erreichen können.

Bisher wurde auf den Begriff der *Erreichbarkeit* noch nicht genau eingegangen. Formal lautet die Definition:

> Eine Zuweisung der Form $[\texttt{x:=a}]^k$ kann einen bestimmten Programmpunkt l *erreichen* (typischerweise den *entry* bzw. *exit* eines elementaren Programmblocks), wenn es einen Ablauf des Programms gibt, in dem an diesem Programmpunkt l die letzte Zuweisung von x am Programmpunkt k stattgefunden hat. Man notiert dann (x, k) in der *entry* bzw. *exit* Menge für Programmpunkt l.

Bei der *Reaching Definitions Analysis* gibt es für jeden Programmpunkt l eine *entry* und *exit* Menge, $RD_{entry}(l)$ und $RD_{exit}(l)$ (Reaching Definitions). Die *entry* und *exit* Mengen stehen in Beziehung zueinander, was entscheidend für ihre automatische Berechnung ist.

2.2.2.2 Gleichungsbasierter Ansatz

Die Beziehungen der *entry* und *exit* Mengen werden durch Gleichungen dargestellt. Die Gleichungen werden mit Hilfe folgender drei Regeln für jeden Programmpunkt l aufgestellt:

1. $RD_{entry}(l) = RD_{exit}(l_1) \cup \cdots \cup RD_{exit}(l_m)$ wenn $l_1, \ldots, l_m$ die Label der Vorgänger Knoten im Flussgraphen von Programmpunkt l sind.
2. Falls l keine Vorgänger hat (also der erste Programmpunkt ist, dann ist

$$RD_{entry}(l) = \{(x, ?) | \text{ x ist eine Variable im Programm}\}$$

3. Falls l eine Zuweisung der Form $[\texttt{x:=a}]^l$ ist, dann ist $RD_{exit}(l) = (RD_{entry}(l) \backslash \{(x, l') | l' \in Label\}) \cup \{(x, l)\}$
4. Sonst ist $RD_{exit}(l) = RD_{entry}(l)$

Auf diese Weise erhält man $2n$ Gleichungen, wenn das Programm aus n Programmpunkten besteht. Auf der linken Seite stehen jeweils n *entry* und *exit* Mengen. Diese können als $2n$-Tupel $\overrightarrow{RD}$ geschrieben werden. Der gleichungsbasierte Ansatz macht nun folgendes: es wird eine Funktion F aufgestellt, welche durch iterative Anwendung auf sich selbst die RD Mengen berechnet. F ist wie folgt definiert:

$$F : (\mathcal{P}(\mathbf{Var} \times \mathbf{Lab}))^{2n} \to (\mathcal{P}(\mathbf{Var} \times \mathbf{Lab}))^{2n}$$

Jede RD Menge ist ein Element von $\mathcal{P}(\mathbf{Var} \times \mathbf{Lab})$. Die Funktion F kann also auf das $2n$-Tupel $\overrightarrow{RD}$ angewendet werden. Bei der Definition von F berücksichtigt man das obige Gleichungssystem. Von F wird gefordert, dass $\overrightarrow{RD} = F(\overrightarrow{RD})$ gilt, d. h. $F(\overrightarrow{RD})$ soll mit den rechten Seiten der Gleichungen korrespondieren.

Auf $(P(\mathbf{Var} \times \mathbf{Lab}))^{2n}$ definiert man die Halbordnung $\sqsubseteq$:

$$\overrightarrow{RD} \sqsubseteq \overrightarrow{RD'} \iff \forall i : RD_i \subseteq RD'_i.$$

Diese Halbordung induziert auf $(P(\mathbf{Var} \times \mathbf{Lab}))^{2n}$ einen **vollständigen Verband** [**?**] mit dem kleinsten Element

$$\overrightarrow{\emptyset} = (\emptyset, \cdots, \emptyset)$$

und der paarweisen oberen Schranke

$$\overrightarrow{RD} \sqcup \overrightarrow{RD'} = (RD_1 \cup RD'_1, \cdots, RD_n \cup RD'_n).$$

Es lässt sich außerdem zeigen, dass F bzgl. $\sqsubseteq$ monoton ist d. h., dass

$$\overrightarrow{RD} \sqsubseteq \overrightarrow{RD'} \text{ impliziert } F(\overrightarrow{RD}) \sqsubseteq F(\overrightarrow{RD'})$$

gilt. Weiter erfüllt F die aufsteigende Kettenbedingung [**?**] , so dass nach endlich vielen Anwendungen von F auf sich selbst die Kette irgendwann stationär wird, d. h. $F^n(\overrightarrow{\emptyset}) = F^{n+1}(\overrightarrow{\emptyset})$ gilt.

Um die kleinste Lösung des Gleichungssystems zu berechnen, betrachte man die Folge $(F^n(\overrightarrow{\emptyset}))_n$. Es ist zu beachten, dass $\overrightarrow{\emptyset} \sqsubseteq F(\overrightarrow{\emptyset})$ gilt. Mit einer einfachen vollständigen Induktion kann man zeigen, dass für alle n, $F^n(\overrightarrow{\emptyset}) \sqsubseteq F^{n+1}(\overrightarrow{\emptyset})$ gilt. Alle Elemente der Folge sind aus $(\mathcal{P}(\mathbf{Var} \times \mathbf{Lab}))^n$ und da sowohl **Var** als auch **Lab** in der Praxis endliche Mengen sind, können nicht alle Elemente der Folge disjunkt sein; die Folge erfüllt also die *aufsteigende Kettenbedingung*. D. h es existiert ein n, so dass:

$$F^{n+1}(\overrightarrow{\emptyset}) = F^n(\overrightarrow{\emptyset}).$$

Es ist $F^{n+1}(\overrightarrow{\emptyset}) = F(F^n(\overrightarrow{\emptyset}))$ und mit der obigen Gleichung können wir sagen, dass $F^n(\overrightarrow{\emptyset})$ ein *Fixpunkt* von F ist und eine Lösung des Gleichungssystems darstellt.

Tatsächlich erhält man sogar die kleinste Lösung, denn angenommen $\overrightarrow{RD}$ ist eine andere Lösung, d. h. es gilt $\overrightarrow{RD} = F(\overrightarrow{RD})$, dann lässt sich mit einer vollständigen Induktion zeigen, dass $F^n(\overrightarrow{\emptyset}) \sqsubseteq \overrightarrow{RD}$. Dies gilt auch dann, wenn $\overrightarrow{RD}$ die kleinste Lösung ist. Da die Lösung $F^n(\overrightarrow{\emptyset})$ beinhaltet, also die kleinste Menge von Reaching Definitions, welche konsistent mit unserem Programm sind, erhält man eine Lösung, die den Anforderungen genügt. Man könnte zwar weitere Paare hinzunehmen, ohne dass die Lösung inkonsistent zu dem Programm wird, aber dies steht im Widerspruch zu dem Grundsatz, dass die Lösung so präzise wie möglich und so sicher wie nötig sein soll.

Um die Berechnung nun zu automatisieren, kann man den folgenden einfachen nichtdeterministischen Algorithmus verwenden:

Initialisierung:
$\overrightarrow{RD} := \overrightarrow{\emptyset}$ d. h. $RD_1 := \emptyset; \ldots; RD_{2n} := \emptyset;$

Iteration:

`while` (es gibt ein j, so dass $RD_j \neq F_j(RD_1, \cdots, RD_{2n})$) `do`
$RD_j := F_j(RD_1, \cdots, RD_{2n})$

Der Algorithmus startet mit dem Nullvektor $(\overrightarrow{\emptyset})$ und wendet dann solange die rechte Seite einer Gleichung auf die linke an, bis der kleinste Fixpunkt von F erreicht ist.

2.2.3 Erreichbarkeitsanalyse paralleler Prozesse mit Baumautomaten

Ein Eingabealphabet ist eine endliche Menge $\mathcal{F}$ von Symbolen mit einer Stelligkeitsfunktion $\eta : \mathcal{F} \rightarrow \mathbb{N}$. Die Unterteilung von $\mathcal{F}$ ist nach der Stelligkeit dargestellt ($\mathcal{F} = \mathcal{F}_0 \cup \mathcal{F}_1 \cup \mathcal{F}_2 \cup ....$). $\mathcal{T}(\mathcal{F})$ bezeichnet die Menge von Termen über $\mathcal{F}$. Elemente von $\mathcal{T}(\mathcal{F})$ werden endliche Bäume genannt. Ein endlicher bottom-up Baumautomat $\mathcal{A}$ ist ein 4-Tupel $< \mathcal{F}, Q, F, R >$, wobei $\mathcal{F}$ das Eingabealphabet mit Stelligkeit, Q eine endliche Menge von Zuständen, F$\subseteq$Q die Menge der Endzustände und R eine Menge von Übergangsregeln von der Form $f(q_1, q_2, ..., q_n) \rightarrow q$ mit $n = \eta(f)$ für $f \in \mathcal{F}$ sind. Baumautomaten mit ε-Regeln erlauben Übergänge von der Form $q \rightarrow q´$.
Bottom-up bedeutet, dass für jeden Knoten ein Zustand gemäß der Übergangsregel R gewählt wird. Sobald die Zustände für die Kinder des Knotens berechnet sind. An den Blättern gibt es keine Kinder, folglich können wir hier direkt einen Zustand gemäß R wählen.
Darüber hinaus bedeutet $t \stackrel{A}{\rightarrow} q$, dass durch endlich viele Regelanwendungen von $\mathcal{A}$ der

endliche Baum $t \in \mathcal{T}(\mathcal{F})$ durch $q \in$ Q ersetzt wird. $\mathcal{A}$ akzeptiert t genau dann, wenn für einen Endzustand q $t \xrightarrow{A} q$ gilt. L($\mathcal{A}$) bezeichnet die Menge aller akzeptierenden Terme von $\mathcal{A}$. Baumsprachen, die mit L($\mathcal{A}$) übereinstimmen, sind ebenfalls reguläre Baumsprachen für einen Baumautomaten $\mathcal{A}$. Reguläre Baumsprachen sind abgeschlossen gegen Komplement, Vereinigung und Schnitt.
Ein Baumautomat ist *completely specified* (vollständig), falls es für ein $f \in \mathcal{F}$ und $q_1, ..., q_n \in$Q eine Regel von der Form $f(q_1, q_2, ..., q_n) \rightarrow q$ gibt.
Die Größe eines Baumautomatens $\mathcal{A}$ wird mit $\mid \mathcal{A} \mid$ bezeichnet, dies ist dabei die Anzahl von Zuständen plus der Anzahl von den Übergangsregeln von $\mathcal{A}$.
Ein Baumautomat ist deterministisch, falls alle Übergangsregeln verschiedene linke Seiten haben (und wenn es keine ε-Regeln gibt). Für einen Baumautomaten $\mathcal{A}$ kann in Zeit O($\mid \mathcal{A} \mid$) ermittelt werden, ob die Sprache L($\mathcal{A}$) leer ist. Ausserdem kann für einen gegebenen Baum t in polynomieller
Zeit $\mid \mathcal{A} \mid + \mid t \mid$ ermittelt werden, ob t von einem Baumautomaten $\mathcal{A}$ akzeptiert wird.
Um die Erreichbarkeitsanalyse von parallelen Prozessen mit Baumautomaten zu erläutern, müssen wir uns erst einmal mit der Prozess-Algebra vertraut machen. Die Prozess-Algebra besteht zu einem Teil aus der Syntax und zum anderen aus der Semantik. Die Syntax besteht aus der Menge der Handlungsnamen ($Act = \{a, b, c, ...\}$), aus der Menge der Prozessvariablen ($Var = \{X, Y, Z, ...\}$) und aus der Menge der Prozess-Algebra (PA) Terme ($E_{PA} = \{t, u, ...\}$), die durch die folgende BNF festgelegt sind:

$$t, u ::= 0 \mid X \mid t.u \mid t \| u$$

Für ein $t \in E_{PA}$ ist $Var(t)$ die Menge der Prozessvariablen, die in t vorkommen und $Subterm(t)$ die Menge aller Teiltermen von t. Eine PA-Deklaration ist eine endliche Menge $\triangle = \{X_i \xrightarrow{a_i} t_i — i = 1, ..., n\}$ von Prozessübergangsregeln. Dabei müssen die X_i´s nicht verschieden sein. Die Menge der Prozessvariablen, die in $\triangle$vorkommen, werden mit $Var(\triangle)$ bezeichnet, und $Subterm(\triangle)$ steht für die Vereinigung aller $Subterm(t)$, so dass t eine rechte oder eine linke Seite einer Regel in $\triangle$ ist.
Mit der Semantik werden die Übergänge von einem PA-Term zu einen anderen PA-Term bei Ausführung einer Aktion beschrieben.
Wir können Baumautomaten benutzen, um eine Menge von Termen aus E_{PA} zu erkennen. Dieses ist möglich, weil $E_{PA} = \mathcal{T}(\mathcal{F})$ ist für das $\mathcal{F}$ mit F_0={0,X,Y,.....} und F_2={. , ||}. Für ein $t \in E_{PA}$ ist die Baumsprache $\{t\}$ regulär und ein Automat für $\{t\}$ braucht nur $|t|$ Zustände. Die Menge der terminierenden Prozess(term)e wird mit $L^{\emptyset}$ bezeichnet. $L^{\emptyset}$ ist eine reguläre Baumsprache. Ein Automat für $L^{\emptyset}$ braucht nur einen Zustand haben.
Wir wollen uns jetzt mit der Regularität von erreichbaren Mengen beschäftigen. Für ein Element t aus E_{PA} bezeichnet Pre*(t) (Post*(t)) die Menge von Vorgängern (die Menge

von Nachfolgern) von t. Dabei sind Pre$^*(t)$, Pre (t) und Pre$^+$(t) (Post*(t), Post (t) und Post$^+$(t)) reguläre Baumsprachen. Ein Baumautomat braucht für die reguläre Baumsprache Pre*(t) nur $O(|\triangle| + |t|)$ Zustände.
Für uns ist noch wichtiger , dass auch Pre*(L) für eine reguläre Sprache L regulär ist. Falls L eine reguläre Teilmenge von E_{PA} ist, dann ist auch Pre*(L) regulär. Man kann aus einem Automaten A_L, welcher L erkennt, einen Automaten für A_{Pre^*} konstruieren. A_{Pre^*} ist eine Kombination aus drei Bestandteilen. Der erste ist ein vollständiger Automat $\mathcal{A}_\emptyset$, der terminierende Prozesse erkennt. Der zweite Bestandteil ist ein Automat A_L, der die Sprache L akzeptiert. Die letzte Komponente benutzt einen Wahrheitswert um darzustellen, dass Schritte ausgeführt worden sind. Dabei benötigt der Automat für A_{Pre^*} nur 4k Zustände, falls der Automat A_L k Zustände haben sollte.

2.3 Analyse mit Automaten

2.3.1 Erfüllbarkeitsprüfung von Formeln der Presburger-Arithmetik mit Hilfe endlicher deterministischer Automaten

Die Presburger-Arithmetik erlaubt es, Gleichungen und Ungleichungen über ganzzahligen Variablen durch Boolesche Operatoren und Quantoren zu verknüpfen. Jede der beiden Seiten einer Gleichung oder Ungleichung ist eine aus Variablen und konstanten Werten gebildete Summe, wobei die Variablen beliebige konstante Koeffizienten besitzen dürfen. Wichtig ist, dass die Presburger-Arithmetik aus Gründen der Entscheidbarkeit mit Ausnahme der Variablenkoeffizienten keine Multiplikationen ermöglicht. Terme wie z. B. $3 * x_1 * x_2$ sind also syntaktisch nicht korrekt. Als Boolesche Operatoren stehen $\wedge$, $\vee$, und $\neg$ zur Verfügung, außerdem gibt es die Quantoren $\exists$ und $\forall$. Die Quantoren beziehen sich auf einzelne Variablen. Das hier dargestellte Verfahren zur Erfüllbarkeitsprüfung wurde [**?**] entnommen.

2.3.1.1 Presburger-Arithmetik

Die folgende Grammatik gibt die exakte Syntax an.

- $F ::= ATOM \mid \neg(F) \mid (F \vee F) \mid (F \wedge F) \mid \forall V : (F) \mid \exists V : (F) \mid (F)$
- $ATOM ::= T\,R\,T$
- $T ::= L \mid V \mid T + T \mid T - T \mid L * V$
- $R ::= < \mid > \mid \leq \mid \geq \mid =$
- $V ::= smallAlpha \mid V.smallAlpha \mid V.digit$
- $smallAlpha ::= a \mid \cdots \mid z$
- $digit ::= 0 \mid \cdots \mid 9$
- $L ::= ganze\,Zahlen\,(positive, negative, 0)$

Gleichungen und Ungleichungen werden als Atome bezeichnet, obwohl sie im Syntaxbaum natürlich selbst noch durch mehrknotige Bäume repräsentiert sind. Ersetzt man diese Bäume jeweils durch einen Metaknoten, dann ist die Menge der entstandenen Metaknoten

die Blättermenge des modifizierten Syntaxbaums.

Die Semantik der Presburger-Arithmetik wird so definiert, wie man es bei Betrachtung der Syntax auch erwartet. Da eine Formalisierung an dieser Stelle keinen Mehrwert bringt, soll darauf verzichtet werden.

Wir sind daran interessiert, für beliebige Formeln der Presburger-Arithmetik zu entscheiden, ob sie erfüllbar sind oder nicht. Die Entscheidbarkeit dieses Problems wird durch das nachfolgend beschriebene Verfahren bewiesen. Allerdings kann nicht erwartet werden, dass es für beliebige Eingaben effizient ist, denn für das Problem wurde ohne Zugrundelegung von Hypothesen eine doppelt exponentielle untere Schranke für den Worst Case nachgewiesen. Es ist also deutlich schwieriger als alle NP-vollständigen Probleme. Überdies kann jedes Problem aus NP auf die Presburger-Arithmetik reduziert werden. Dies lässt sich beispielsweise durch eine einfache Reduktion des NP-vollständigen Problems 3-SAT auf die Erfüllbarkeitsprüfung nachweisen. Für eine 3-SAT-Formel $\bigwedge_{1\leq i\leq m}(L_{i,1} \vee L_{i,2} \vee L_{i,3})$ mit Literalen $L_{i,j} \in \{x_{i,j}, \neg x_{i,j}\}$ und Variablen $x_{i,j} \in \{x_1, \ldots, x_n\}$ wird in polynomieller Zeit die erfüllbarkeitsäquivalente Presburger-Formel $(\bigwedge_{1\leq i\leq m}(x_{i,1} = c_{i,1} \vee x_{i,2} = c_{i,2} \vee x_{i,3} = c_{i,1})) \wedge (\bigwedge_{1\leq i\leq n}(x_i = 1 \vee x_i = 0))$ berechnet. Dabei ist $c_{i,j} = 1$, falls $L_{i,j} = x_{i,j}$ und $c_{i,j} = 0$ falls $L_{i,j} = \neg x_{i,j}$.

Anwendungen der Presburger-Arithmetik bestehen beispielsweise im Lösen von Systemen linearer diophantischer Gleichungen und in der Integer Programmierung. Lineare diophantische Gleichungen sind genau solche Gleichungen, die auch in der Presburger-Arithmetik formuliert werden können. Die beiden erwähnten Probleme lassen sich auf das Lösen von Formeln der Presburger-Arithmetik reduzieren, also auf das Auffinden einer erfüllenden Belegung (falls vorhanden) der in der Formel auftretenden freien Variablen. Glücklicherweise eignet sich das hier vorgestellte Verfahren auch zur Lösung dieses Problems, denn der für die Presburger-Formel konstruierte Ergebnisautomat akzeptiert genau die erfüllenden Belegungen der freien Variablen, repräsentiert also die Lösungsmenge der Formel. Als eine weitere Anwendung der Presburger-Arithmetik ist die Systemverifikation zu nennen. Beispielsweise könnten Formeln der Presburger-Arithmetik als Zusicherungen in einem Quelltext verwendet werden. Die Automatenkonstruktion bietet sich besonders dort an, weil sich Zusicherungen von einem Schritt zum nächsten oft nur minimal ändern. Für die nächste Zusicherung muss das Verfahren dann nicht von vorne gestartet werden, sondern es können einige Zwischenergebnisse wiederverwendet werden.

2.3.1.2 Repräsentation von Variablenbelegungen

O. B. d. A. nehmen wir an, dass Gleichungen [Ungleichungen] immer in der Form $a_1x_1 + a_2x_2 + \ldots + a_nx_n = c$ $[a_1x_1 + a_2x_2 + \ldots + a_nx_n \leq c]$ mit $n > 0$ vorliegen, wobei $x_1, \ldots, x_n$ paarweise verschiedene Variablen und $a_1, \ldots, a_n$ ganzzahlige Koeffizienten mit $a_i \neq 0$ für ein i sind und c eine beliebige ganzzahlige Konstante ist. Zusätzlich definieren wir noch $a = (a_1, a_2, \ldots, a_n)$ und $x = (x_1, x_2, \ldots, x_n)$. Das Verfahren konstruiert zunächst für jedes Atom einen Automaten, der genau die Lösungen des Atoms akzeptiert. Die Atom-Automaten können beliebige Belegungen der im zugehörigen Atom vorkommenden Variablen lesen und treffen die korrekte Entscheidung. Für ein Atom in der oben beschriebenen Normalform ergibt sich ein Automat, der auf dem Bitvektoralphabet $\{0,1\}^n$ arbeitet. Die i-te Position der Vektoren bezieht sich dabei auf x_i. Wenn nun ein Wort $w = (b_1, \ldots, b_m)$ von Bitvektoren b_j gelesen wird, dann wird für jede Variable x_i ein Bitmuster der Länge m gelesen, nämlich $b_{1,i}b_{2,i}\ldots b_{m,i}$. Dieses Bitmuster wird als Zweierkomplementdarstellung einer ganzen Zahl - der Belegung von x_i - interpretiert. Das erste Bit lässt das Vorzeichen erkennen, früher eingelesene Vektoren haben eine höhere Bitwertigkeit, es wird also ausgehend vom Most Significant Bit in Richtung Least Significant Bit gelesen. Jedes Wort kann auf diese Weise als Variablenbelegung interpretiert werden. Das leere Wort werde als Belegung interpretiert, die allen Variablen den Wert 0 zuordnet.

2.3.1.3 Konstruktion von Automaten für Gleichungen

Für ein Atom $a_1x_1 + a_2x_2 + \ldots + a_nx_n = c$ soll ein deterministischer endlicher Automat erzeugt werden, der genau die Lösungsmenge der Gleichung akzeptiert, wobei die Lösungen wie oben beschrieben dargestellt werden. Die Idee ist, dass jeder Zustand einen bestimmten Wert repräsentiert, den die linke Seite annehmen kann, und dass sich der Automat beim Lesen einer Folge von Bitvektoren stets in dem Zustand befindet, der den momentanen Wert der linken Seite repräsentiert. Der momentane Wert ergibt sich aus dem bereits gelesenen Teil des Wortes, denn dieser ist ja auch eine Variablenbelegung.

Wenn der Automat von einem Zustand q in einen Zustand q' übergeht, während er den Bitvektor b liest, steht dies für eine Wertänderung des Ausdrucks $a_1x_1 + a_2x_2 + \ldots + a_nx_n$ von q auf q', der durch die Veränderung der Variablenbelegung zustande kommt. Diese Veränderung wiederum resultiert auf Bitvektorebene daraus, dass der Vektor b an die bereits gelesene Belegung hinten angehängt wird. Für jede Variable x_i gilt: Bedeutete der bisher gelesene Bitstring den Wert d, so bedeutet der neue, um b_i nach hinten verlängerte Bitstring den Wert $2d + b_i$ (Eigenschaft der Zweierkomplementdarstellung). Da die Gleichung linear ist, verdoppelt sich die linke Seite zunächst. Es muss dann noch

für alle i mit $b_i = 1$ der Wert a_i hinzuaddiert werden. In Vektorschreibweise bedeutet dies gerade $q' = 2q + a.b$, wobei . die Matrixmultiplikation ist. Befindet sich der Automat im Zustand q, während er b liest, so geht er in q' über.

Der Automat benötigt einen Initialzustand *init*, in dem der Vorzeichen-Bitvektor gelesen wird. Der Automat befindet sich genau dann in *init*, wenn das bisher gelesene Wort das leere Wort ist, also jede Variable mit 0 belegt ist. Die linke Seite ist dann auch 0. Bekommt die Variable x_i das Vorzeichenbit $b_i = 0$, so bleibt ihre Belegung gleich 0. Falls $b_i = 1$, so ist ihre neue Belegung gleich -1. Der Automat geht also beim Lesen des Vorzeichen-Bitvektors b in den Zustand $q' = -a.b$ über. Der Initialzustand ist akzeptierend genau dann, wenn $c = 0$ ist. Er hat keine eingehenden Transitionen. In jedem Fall ist der Zustand c akzeptierend.

Nun ist klar, wie der Automat ausgehend von *init* Zustand für Zustand erzeugt werden kann. Für jeden Vektor $b \in \{0,1\}^n$ wird der Nachfolgezustand berechnet, welcher hinzugefügt wird, falls er noch nicht existiert. Falls der Nachfolgezustand neu hinzugefügt wurde, wird er zusätzlich in eine Queue aktiver Zustände eingefügt, damit für ihn später ebenfalls Transitionen berechnet werden. Um die Terminierung zu gewährleisten, kann die Tatsache ausgenutzt werden, dass Zustände q mit $|q| > \sum_{1 \leq i \leq n} |a_i|$ nicht konstruiert werden müssen, weil von ihnen aus der akzeptierende Zustand c nicht mehr erreicht werden kann. Ergibt sich nach der Rechenregel ein solcher Nachfolgezustand, so sollte die Transition auf einen endgültig ablehnenden Zustand *reject* umgebogen werden. Weil die Schranke jedoch nicht scharf ist, werden weiterhin überflüssige Zustände konstruiert. Dagegen hat das Verfahren die positive Eigenschaft, dass die Ergebnisautomaten deterministisch sind.

Wir lösen das Problem der überflüssigen Zustände durch eine Rückwärtskonstruktion, die im Zustand c gestartet wird. Auf diese Weise bekommt der Automat nur Zustände, von denen aus c erreichbar ist. Der Fall, dass der Algorithmus einen unendlich langen Pfad zurückverfolgt, kann nicht auftreten, denn die Rückwärtskonstruktion kann höchstens die endlich vielen Zustände q mit $|q| \leq \sum_{1 \leq i \leq n} |a_i|$ konstruieren. Überflüssige Zustände werden genau dann konstruiert, wenn die Gleichung unerfüllbar ist (z. B. $7x_1 + 21x_2 = 6$). In diesem Fall werden *init* keine ausgehenden Transitionen zugewiesen, was leicht geprüft werden kann. Nach den bisherigen Ausführungen sollte klar sein, dass Algorithmus 2 einen minimalen deterministischen endlichen Automaten konstruiert, der genau die Lösungsmenge der Gleichung akzeptiert. Wird der Zustand *reject* hinzugefügt, so ist *reject* nicht überflüssig, denn er ist der einzige Zustand, von dem aus c nicht erreichbar ist. Er ist also mit keinem anderen Zustand äquivalent. Wegen $a_i \neq 0$ für ein i gibt es stets Eingaben, die nicht mehr zu Lösungen verlängert werden können. Es werden

$O((\log c) \sum_{1 \leq i \leq n} |a_i|)$ viele Zustände erzeugt.

procedure *automatonForEquation*
  Zustandsmenge $Q := \{c, init\}$;
  Transitionsmenge $T := \emptyset$;
  Queue $queue := (c)$;
  while $queue \neq ()$ **do**
    $q := dequeue(queue)$;
    for all $b \in \{0,1\}^n$ **do**
      $q_0 := (q - a.b)/2$;
      if q_0 ganzzahlig **then**
        if $q_0 \notin Q$ **then** $Q := Q \cup \{q_0\}$ und $enqueue(queue, q_0)$;
        $T := T \cup \{(q_0, b, q)\}$;
        if $q = -a.b$ **then** $T := T \cup \{(init, b, q)\}$;
  if T enthält keine ausgehenden Transitionen für *init* **then**
    $Q := \{init\}$;
    $T := \{init\} \times \{0,1\}^n \times \{init\}$;
    init ist nicht akzeptierend;
  else
    $Q := Q \cup \{reject\}$;
    $T := T \cup \{(q, b, reject) \mid \neg\exists q' \in Q : (q, b, q') \in T\}$;
    mache c akzeptierend;
    if $c = 0$ **then** mache *init* akzeptierend;
end procedure

Algorithmus 2 : Konstruktion eines Automaten für eine Gleichung

2.3.1.4 Konstruktion von Automaten für Ungleichungen

Das vorgestellte Konstruktionsprinzip kann auf Ungleichungen der Form $a_1x_1 + a_2x_2 + \ldots + a_nx_n \leq c$ übertragen werden. Der Zustand q bedeutet nun, dass die linke Seite momentan kleiner oder gleich q ist. Der Initialzustand *init* ist genau dann akzeptierend, wenn $c \geq 0$ ist. Für alle anderen Zustände q ist q genau dann akzeptierend, falls $q \leq c$. Bei der Rückwärtskonstruktion wird $q_0 := \lfloor (q - a.b)/2 \rfloor$ anstelle von $q_0 := (q - a.b)/2$ benutzt. Ungleichungen sind stets erfüllbar, weshalb Schritt 3 entfallen kann.

Allerdings sind die entstehenden Automaten nicht notwendigerweise deterministisch. Für die Ungleichung $x_1 - x_2 \leq 2$ wird der Zustand 2 konstruiert. Bei der Bearbei-

tung von 2 wird für den Bitvektor $(0,1)$ die Transition $(1,(0,1),2)$ hinzugefügt, weil $\lfloor(2-a.(0,1))/2\rfloor = 1$ ist. Bei der Bearbeitung von 1 wird aber die Transition $(1,(0,1),1)$ erzeugt, da $\lfloor 1-a.(0,1)/2\rfloor = 1$ ist. Die Determinisierung eines nichtdeterministischen Automaten kann im Allgemeinen zu einem exponentiellen blow-up des Automaten führen. Hier haben wir es jedoch mit einem deutlich günstigeren Spezialfall zu tun, der eine Determinisierung in linearer Zeit zulässt. Genauer gesagt können wir den Automaten derart modifizieren, dass der günstige Spezialfall vorliegt. Was wir benötigen, nennt sich geordneter endlicher Automat.

Definition: Ein endlicher Automat A heißt geordnet, wenn es eine totale Ordnung $\prec$ auf seiner Zustandsmenge Q gibt, so dass für alle $q,r \in Q$ mit $q \prec r$ die von A_q erkannte Sprache eine Teilmenge der von A_r erkannten Sprache ist. Dabei bezeichnet A_s für ein $s \in Q$ den Automaten, der sich aus A durch die Erklärung von s zum einzigen Initialzustand ergibt.

Lemma: Jeder geordnete endliche Automat lässt sich in Zeit $O(|A|)$ in einen äquivalenten deterministischen Automaten überführen. Dabei muss nur die Transitionsmenge verändert werden.

Um unseren Automaten zu einem geordneten Automaten zu machen, müssen wir für jeden Zustand $q \in Q$ und jeden Bitvektor $b \in \{0,1\}^n$ für den kleinsten (bzgl. der üblichen Ordnungsrelation $<$ auf ganzen Zahlen) Zustand q' mit $q' \geq 2q + a.b$ die Transition (q,b,q') hinzufügen, falls es überhaupt Zustände $q' \geq 2q + a.b$ gibt. Mit $\prec \;:=\; >$ ist der so vervollständigte Automat ein geordneter Automat.

Intuitiv ist auch klar, was dort passiert: Von einem Zustand q_1 aus, der eine kleinere Zahl repräsentiert, sollten alle Resteingaben akzeptiert werden, die auch von einem Zustand q_2 aus akzeptiert werden, der eine größere Zahl repräsentiert, denn wenn die Resteingabe von q_2 aus in einen akzeptierenden Zustand r führt, so ist $r \leq c$. Startet man von q_1 aus mit der gleichen Resteingabe, so kann der erreichte Zustand nicht größer sein als r und ist damit ebenfalls akzeptierend. Die geordnete Determinisierung macht nichts anderes, als bei Mehrdeutigkeiten bzgl. eines Zustands q und Zeichens b nur die existierende Transition (q,b,q') mit dem kleinsten Integer-Wert q' im Automaten zu belassen und alle anderen zu entfernen, weil der Automat während der Ausführung so stets die stärkste Information verwaltet. Der Zustand *init* wird von $\prec$ nicht berücksichtigt, denn er besitzt keine eingehenden Transitionen und wird deshalb nie mit anderen Zuständen verglichen. Algorithmus 3 konstruiert einen minimalen deterministischen endlichen Automaten für

eine Ungleichung.

procedure *automatonForInequation*
Zustandsmenge $Q := \{c, init\}$;
Transitionsmenge $T := \emptyset$;
Queue $queue := (c)$;
if $0 \leq c$ **then** mache $init$ akzeptierend;
while $queue \neq ()$ **do**
$q := dequeue(queue)$;
if $q \leq c$ **then** mache q akzeptierend;
for all $b \in \{0,1\}^n$ **do**
$q_0 := \lfloor (q - a.b)/2 \rfloor$;
if $q_0 \notin Q$ **then** $Q := Q \cup \{q_0\}$ und $enqueue(queue, q_0)$;
$T := T \cup \{(q_0, b, q)\}$;
if $q \geq -a.b$ **then** $T := T \cup \{(init, b, q)\}$;
vervollständige den Automaten;
determinisiere den Automaten in linearer Zeit;
$Q := Q \cup \{reject\}$;
$T := T \cup \{(q, b, reject) \mid \neg\exists q' \in Q : (q, b, q') \in T\}$;

Algorithmus 3 : Konstruktion eines Automaten für eine Ungleichung

2.3.1.5 Konstruktion von Automaten für beliebige Formeln

Um einen Automaten für eine beliebige Formel der Presburger-Arithmetik konstruieren zu können, wird die Formel zunächst in Pränex-Normalform gebracht. Diese hat die Form $Q_1 x_1 : (Q_2 x_2 (\cdots (Q_m x_m : (\Phi(y_1, \cdots, y_k)) \cdots)$, wobei $Q_1, \cdots, Q_m$ Quantoren sind und $x_1, \cdots, x_m$ die durch diese Quantoren gebundenden Variablen sind. $y_1, \cdots, y_k$ sind die freien Variablen der Formel. Φ ist eine Komposition aus Und-, Oder- und Nicht- Formeln und von Atomen als innersten Elementen. Zusätzlich fordern wir, dass sich Negationen in der Normalform nur noch auf Gleichungen beziehen dürfen.

Eine Möglichkeit ist, für jedes Atom einen Automaten zu konstruieren, wobei nur Variablen mit Koeffizienten $\neq 0$ berücksichtigt werden, und anschliessend bottom-up entlang des Syntaxbaums Automaten für komplexere Teilformeln zu berechnen. Für Boolesche Verknüpfungen können Produktautomaten konstruiert werden, die die beiden zugrunde liegenden Automaten simulieren. Verwendet der den linken Operanden darstellende Automat Bitvektoren der Länge n und der den rechten Operanden darstellende Automat Bitvektoren der Länge m, so arbeitet der Produktautomat mit Bitvektoren der Länge

$n + m$. Bei Oder-Verknüpfungen akzeptiert der Produktautomat, wenn mindestens einer der simulierten Automaten akzeptiert; bei Und-Verknüpfungen, wenn beide akzeptieren. Es ist zu beachten dass die Bitvektoren-Teile für die beiden simulierten Automaten nicht immer unabhängig voneinander gewählt werden können. Gehören zwei Bitpositionen zu ein und derselben Variablen, so dürfen sie nicht verschieden belegt werden. Verbotene Bitvektoren sollten sofort in den endgültig ablehnenden Zustand führen, dann bleibt die Lösungsmenge korrekt. Für $\exists$-Quantoren wird auf dem den Operator darstellenden Automaten eine Projektion durchgeführt. Die Projektion entfernt aus allen Kantenlabels das Bit, das zu der durch den Quantor gebundenen Variablen gehört. Der Automat kann dadurch nichtdeterministisch werden und muss mit einem allgemeinen Determinisierungsverfahren in eine deterministischen Automaten überführt werden. $\forall$-Quantoren werden behandelt, indem der Automat komplementiert, projiziert und nochmals komplementiert wird.

In unserem Projekt haben wir allerdings einen anderen Weg gewählt. Wir berechnen ebenfalls die oben angegebene Normalform und gehen im Syntaxbaum nach dem Bottom-Up-Ansatz vor. Die Atom-Automaten arbeiten jedoch alle auf dem gleichen Alphabet. Die Bitvektoren dieses Alphabets sehen für jede in der Formel auftretende (relevante) Variable eine Bitposition vor, die Zuordnung ist für alle Atom-Automaten identisch. Dadurch können die Und-Operationen einfach durch den Schnitt der beiden Operanden-Automaten und die Oder-Operationen durch die Vereinigung der Automaten berechnet werden. Der Negation einer Gleichung entspricht die Komplementierung des zugehörigen Automaten. Die Quantoren werden genauso behandelt wie im ersten Verfahren. Für Details wird auf die Dokumentation der Implementierung verwiesen.

2.3.2 Model Checking mit Automaten

In der Programmentwicklung steht man unweigerlich vor dem Problem (solange es sich nicht um ein triviales Programm handelt), sicherstellen zu müssen (wenn nicht “muss”, dann ist es doch zumindest wünschenswert), dass ein Programm bestimmte Eigenschaften erfüllt oder nicht erfüllt. Lösungsansätze reichen von einfachen Programmtestdurchläufen bis hin zu Theorembeweisern. Ein Lösungsansatz unter diesen stellt das automatentheoretische Model Checking dar.

2.3.2.1 Kripkestrukturen

Model Checking gehört zu den automatischen Verifikationsmethoden. Der Vorteil, vollautomatisch (minus der immer notwendigen Vorarbeit) Systeme verifizieren zu können, hat u. A. zur Konsequenz, größere Systeme und mehr Szenarien in der selben Zeit untersuchen zu können, als mit halb-automatischen oder manuellen Verfahren möglich wäre. Dabei wird das Programm (oder auch ein Protokoll, oder...) abstrahiert und diese Abstraktion hinsichtlich der gewünschten Eigenschaften untersucht. Unter der Voraussetzung, die Abstraktionstiefe korrekt gewählt zu haben, lassen sich dann gewünschte Eigenschaften des Programms als gegeben oder verletzt herausstellen.

Im Falle des Model Checkings mit Hilfe von Automaten und linearer Temporallogik (siehe unten) wird das Programm in einen Zustandsgraphen, eine Kripke-Struktur, überführt.

Der automatentheoretische Hintergrund (vor allem in Hinsicht auf Büchiautomaten) kann im Abschnitt 2.1.1.2 in Erfahrung gebracht werden, soll aber kurz umrissen werden :

- sei K Kripke-Struktur, sei ϕ Logik-Formel
- wandle K in Büchiautomat A_K (mit Sprache $L(A_K)$) um
- konstruiere Büchiautomat $A_{\neg\phi}$ (mit Sprache $L(A_{\neg\phi})$)
- konstruiere Büchiautomat $A_\cap = A_K \cap A_{\neg\phi}$
- ist $L(A_\cap) = \emptyset$, dann erfüllt K alle Anforderungen, die mit ϕ einhergehen. Ansonsten enthält $L(A_\cap)$ mindestens ein Gegenbeispiel.

Die Konstruktion des Büchiautomaten aus einer Kripke-Struktur ist in [**?**] beschrieben.

2.3.2.2 LTL

Bei LTL handelt es sich um eine Logik zur Beschreibung von Ereignissen entlang eines Berechnungspfades, ohne Verzweigungen. Genauer gesagt und auf den Kontext des Model Checkings bezogen, soll ein Programm eine Eigenschaft, ausgedrückt in LTL, erfüllen, so müssen **ALLE** möglichen Programmabläufe die LTL-Formel wahr machen.

Kernstück der LTL sind atomare Aussagen (AP).

Die Syntax von LTL wird durch die folgenden zwei Regeln beschrieben :

- Sei AP die Menge der Propositionen.

1. $\forall p \in AP : p$ ist eine Formel.
2. $\forall f_1, f_2$ mit f_1, f_2 Formel : dann sind $\neg f_1, (f_1 \wedge f_2), (f_1 \vee f_2), X f_1, (f_1 U f_2)$ ebenfalls Formeln.

$true$ und $false$ können als $\mathbf{T} = p \vee \neg p$ und $\mathbf{F} = \neg \mathbf{T}$ ausgedrückt werden.

Die Definition der Semantik kann in [**?**] in Erfahrung gebracht werden.

2.3.2.3 Konstruktion eines Büchi-Automatens

Dieser Algorithmus [**?**] ist relativ einfach zu realisieren, hat allerdings den Nachteil das seine Laufzeit suboptimal ist, da eine u. U. exponentielle (in der Menge der Teilformeln von ϕ) Anzahl von Knoten erzeugt werden muss.

Der Büchiautomat kann allerdings fast direkt angegeben werden :

Sei ϕ LTL-Formel, sei $cl(\phi)$ die Menge aller Teilformeln von ϕ, sei AP die Menge der atomaren Aussagen. Dann ist der generalisierte Büchiautomat definiert über:

- Alphabet : $\Sigma = 2^{AP}$
- Zustände : $Q \subseteq 2^{cl(\phi)}$ und $q \in Q$ gdw.
  1. $\mathbf{F} \notin q$
  2. $(\ \phi_1 \wedge \phi_2) \in q \Rightarrow \phi_1 \in q$ und $\phi_2 \in q$
  3. $(\ \phi_1 \vee \phi_2) \in q \Rightarrow \phi_1 \in q$ oder $\phi_2 \in q$
- Anfangszustände : $I = \{q | \phi \in q, q \in Q\}$
- Übergänge : $(q, a, p) \in \Delta$ gdw.
  1. $p \in q \Rightarrow p \in a$, $\neg p$ analog
  2. $X\phi_1 \in q \Rightarrow \phi_1 \in p$
  3. $\phi_1 U \phi_2 \in q \Rightarrow \phi_2 \in q \ \ oder \ \ \phi_1 \in q \wedge \phi_1 U \phi_2 \in p$
  4. $\phi_1 R \phi_2 \in q \Rightarrow \phi_1 \in q \ \ und \ \ \phi_1 \in q \vee \phi_1 U \phi_2 \in p$
- Akzeptanzmenge : $F = \{F_{\phi_1 U \phi_2} | F_{\phi_1 U \phi_2} \subseteq Q, \phi_1 U \phi_2 \in cl(\phi)\}$, mit $F_{\phi_1 U \phi_2} = \{q | q \in Q, (\phi_2 \in q \wedge \phi_1 U \phi_2 \in q) \vee (\phi_1 U \phi_2 \notin q)\}$

Wichtig : Ist die Akzeptanzmenge leer, so akzeptiert der Automat alle (möglichen) unendlichen Wörter!

Bei der Übergangsrelation ist zu beachten, daß Δ den Regeln 1.-4. genügen muss, d. h. enthält p keine atomaren Aussagen ist $a = \emptyset$, was äquivalent dazu ist, daß $a = \{\mathbf{T}\}$ ist!

Der resultierende Automat ist ein generalisierter Büchiautomat. Zur Umwandlung dieses Automaten in einen "normalen" Büchiautomaten siehe Abschnitt 2.1.1.3.5.

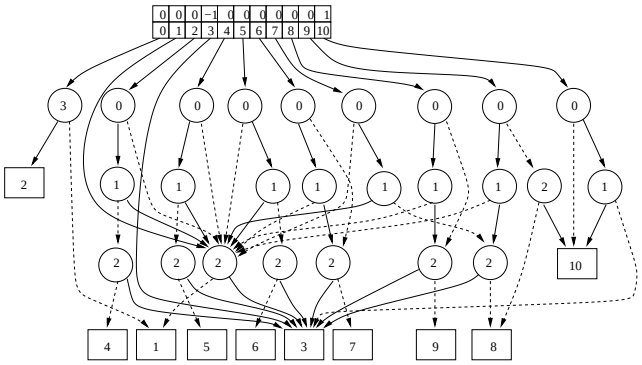


Abbildung 2.8: BDD-basierende Repräsentation eines Automaten

2.4 Existierende Lösungen

2.4.1 Mona/Mosel

Dieser Vortrag diente dem Zweck, zwei bereits existierende Werkzeuge zur Automatenkonstruktion und Analyse vorzustellen. Außerdem sollte aufgezeigt werden, wie diese realisiert wurden. Bei den beiden Projekten handelt es sich um MONA und MOSEL.

2.4.1.1 MONA

MONA [**?**] ist ein Programm, dass auf der Kommandozeile arbeitet. Es nimmt als Eingabe Programme in der MONA-eigenen Syntax, die der WS1S(*Weak monadic Second-order theory of 1 Successor*)-Logik entspricht, entgegen. Nach dem Einlesen eines MONA-Programms wird es in einen DFA transformiert und es werden einige Analysen an dem MONA-Programm durchgeführt. MONA gibt als Analyseergebnis eine erfüllende Belegung der Variablen des MONA-Programms und, als Gegenbeispiel, eine nicht erfüllende Belegung kürzester Länge aus.

MONA ist in der Lage, den DFA als Zustandsübergangstabelle auf der Kommandozeile auszugeben. Es ist aber auch möglich MONA anzuweisen, den Automaten als *.dot*-Datei auszugeben. Diese *.dot*-Datei kann dann, beispielsweise, mit dem Programm *graphviz* angezeigt werden.

Intern werden die DFAs mit einer Datenstruktur verwaltet, die *multi-terminal, shared BDD* heißt. Dabei handelt es sich um einen azyklischen Graphen wie er in Abbildung 2.8 zu sehen ist. In dieser Abbildung sehen wir ein Array mit zwei Zeilen und elf Spalten. Die Einträge in der unteren Zeile entsprechen den Startzuständen des DFAs, die obere Zeile gibt die Art des Zustands an. Der Wert 1 steht hier für akzeptierend, -1 für nicht

akzeptierend und 0 für "don't care". Die Variablen des MONA-Programms wurden durchgehend indiziert, so dass jeder Variablen eine Zahl zugeordnet ist. Die Variablen werden in dem Bild als innere Konten (rund) dargestellt. Die Blätter des Graphen enthalten die Endzustände (eckig) nach einem Zustandsübergang. Eingaben für den Automaten werden als Bitvektor dargestellt.

Beispielhaft wird nun ein Zustandsübergang vollzogen: Ausgehend von Abbildung 2.8, nehmen wir an, wir befinden uns im Zustand 2 ("don't care"). Es sei der Bitvektor (1,0,0,1) eine Eingabe für den Automaten. Jede Stelle im Eingabevektor korrespondiert mit einer Variablen. So gehört die Stelle 0 im Eingabevektor zur Variablen 0, die Stelle 1 zur Variablen 1 u. s. w.. Da wir uns im Zustand 2 befinden, folgen wir dem Pfeil (ausgehenden Kante), der uns zur ersten Variablen führt. Die erste Variable ist die Variable mit Index 0. Wir schauen uns die Eingabe für diese Variable an, Sie befindet sich an Stelle 0 im Eingabevektor. Falls die Eingabe 1 ist, folgen wir dem durchgezogenen Pfeil, bei einer 0 dem gestrichelten. Ist die Eingabe 1, so folgen dem Pfeil zur Variablen 1. An Stelle 1 befindet sich im Eingabevektor eine 0. Wir folgen deshalb dem gestrichelten Pfeil zur Variablen 2. Auch hier steht an 2. Stelle 2 Eingabevektor eine 0 und führt uns zum Blatt 4. Dies bedeutet, dass wir für den gegebenen Eingabevektor einen Zustandswechsel vom Zustand 2 zum Zustand 4 machen müssen.

Neben der Möglichkeit, DFAs zu konstruieren, ist es in MONA auch möglich, Baumautomaten zu erstellen. In MONA werden diese GTAs (*Guided Tree Automata*) genannt. Als Eingabelogik für solche GTAs wird eine Verallgemeinerung der WS1S verwendet, die WS2S (*Weak monadic Second-order theory of 2 Succesors*). Die 1 in WS1S deutet an, dass Elemente mit nur einem Nachfolger behandelt werden, in MONA werden diese Elemente als String interpretiert. WS2S hingegen behandelt Elemente mit zwei Nachfolgern (links, rechts). Diese werden als endliche Bäume interpretiert. So sind die Variablen in WS1S natürliche Zahlen und in WS2S Positionen in einem unendlichen Binärbaum.

Die in MONA genutzte Logik ist zudem auch mächtig genug, um Ausdrücke der Presburger-Arithmetik, sowie die M2L-Str(*Monadic Second-order Logic on Strings*) zu emulieren.

Ein weiteres Feature von MONA ist die Möglichkeit, DFAs und GTAs zu exportieren und zu importieren.

2.4.1.2 MOSEL

MOSEL [**?**] ist in mehreren aufeinander aufbauenden Schichten angeordnet (siehe Abbildung 2.9). Auf der Ebene *Tool Layer* befindet sich die *Minimal Logic*. Die *Minimal*

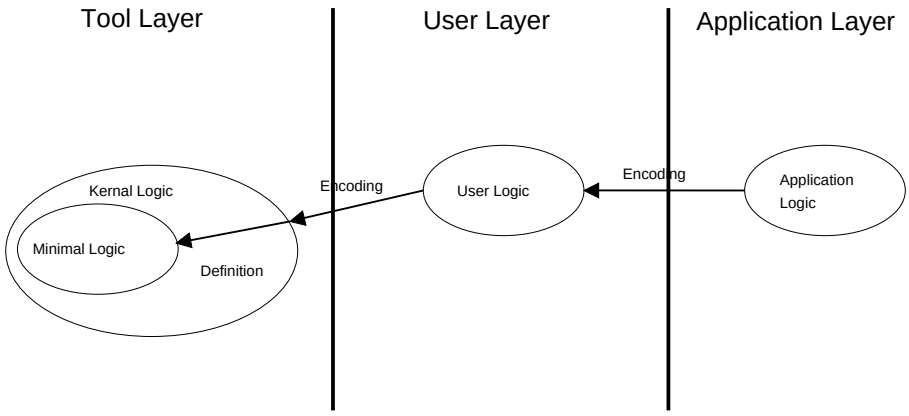


Abbildung 2.9: Logik-schichten in MOSEL

Logic ist eine Sprache, die aus einer minimalen Mengen von Primitiven besteht, deren Semantik mit Automaten korrespondiert. Die hier verwendete Sprache ist eine Variante der M2L-Str. Aus ihr lassen sich Automaten induktiv konstruieren. Die Vorteile der *Minimal Logic* sind, dass ihre Semantik einfach ist und sich Beweise zur Abgeschlossenheit und Korrektheit von reicheren Sprachen häufig auf diese *Minimal Logic* zurückführen lassen.

Die *Minimal Logic* wird um die *Kernel Logic* erweitert, die komplexere Ausdrücke erlaubt. Denn ein Nachteil der *Minimal Logic* ist, dass sie keine komplexeren Konstrukte zur Verfügung stellt, wie dies höhere Sprachen tun. Damit ist *Minimal Logic* als Frontend für die Erstellung von Automaten ungeeignet. Die *Kernel Logic* wird bei der Übersetzung in Automaten in Primitive zergliedert und auf die *Minimal Logic* zurückgeführt. Aus Gründen der Effizienz ist es aber nicht immer sinnvoll, diesen Weg zu gehen. Einige komplexe Ausdrücke werden, vorbei an der *Minimal Logic*, direkt in die Automatenkonstruktion mit einbezogen. Auf diese Art wird eine Effizienzsteigerung erreicht.

Auf der nächste Ebene (*User Layer*) werden andere, von MOSEL unabhängige, Sprachen(*User Logic*) an MOSEL angepasst. Hier könnte man beispielsweise die Presburger-Arithmetik an MOSEL anpassen. Dazu müsste man die anzupassende Sprache solange transformieren, bis sie auf die *Kernel Logic* abgebildet wird. Praktisch wurde dies schon mit der WS1S von MONA vollzogen. Dabei wurde in einem ersten Schritt ein "flattening" durchgeführt. Ziel hierbei war es, einige Funktionen und Konstanten durch Relationen zu ersetzen. Dann musste man die Typen von MONA in MOSEL einbetten ("*type embedding*"). Der letzte Schritt bestand aus einer Normalisierung. Hierbei wurden Allquantoren durch negierte Existenzquantoren ersetzt.

Hier muss man anmerken, dass es kein allgemeines Vorgehen gibt, andere Sprachen nach MOSEL zu integrieren. Die Integration einer anderen Sprache kann sich von der Einbin-

dung der WS1S von MONA drastisch unterscheiden.

Die letzte Ebene heißt *Application Layer*, auf dieser Ebene sollen Anwendungen eingebunden werden, die Mosel nutzen. Es soll ermöglicht werden, dass aus einer Anwendung heraus auf Mosel zugegriffen wird. Als eine Anwendung ist die Modellierung und Verifikation von Hardware angedacht.

2.4.2 Überblick über Automaten-Bibliotheken

Die Modularisierung ist eines der zentralen Konzepte in der Softwareentwicklung. Sie hat den Vorteil, dass Projekte einfach erweiterbar sind, und dass sich Kosten- und Zeitaufwand für die Realisierung eines Projektes erheblich reduzieren lassen. Der Quelltext eines Moduls wird dann in Bibliotheken abgelegt, die als eigenständige und unabhängige Objekte in ein Softwareprojekt eingefügt werden können.

2.4.2.1 Lizenzen

Bei Verwendung von Bibliotheken gibt es einige rechtliche Grundlagen, die man beachten muss. Die meisten frei verwendbaren Bibliotheken stehen unter der GPL oder einer leicht modifizierten GPL.

GPL - General Public License: Die GPL [**?**] bildet eine rechtliche Grundlage, um frei Software zu schaffen und zu veröffentlichen, die von fremden Personen genutzt werden darf (kopieren, verteilen, weiterentwickeln oder anpassen). Nach der geltenden Rechtslage ist ein verändertes Werk aber nicht mehr automatisch, wie das ursprüngliche Werk, frei verwendbar. Um diesen Mangel zu beseitigen, erklärt sich ein Nutzer durch den Gebrauch einer Software unter der GPL damit einverstanden, dass seine eigene Software den gleichen rechtlichen Grundlagen unterliegt wie die ursprüngliche Software. Diesen Vorgang nennt man auch Copyleft.

LGPL - Lesser General Public License: Die LGPL [**?**], ursprünglich Library General Public License genannt, ist speziell für Softwarebibliotheken entwickelt worden. Sie entspringt einer Kritik an der Forderung der GPL, dass eine Software, die Bibliotheken unter der GPL nutzt, ebenfalls unter der GPL veröffentlicht werden muss. Steht eine Bibliothek unter der LGPL, so kann sie frei verwendet werden und es müssen nur Veränderungen an der Bibliothek öffentlich gemacht werden.

BSD: Die BSD-Lizenz [**?**] ähnelt nur noch in groben Zügen der GPL. Sofware unter der BSD Lizenz darf von fremden Personen genutzt werden, und Änderungen

müssen nicht veröffentlicht werden. Das heißt, die Software steht nicht unter der Einschränkung des Copylefts. Die BSD-Lizenz schränkt einen User nur bezüglich der Copyright-Vermerke ein. Diese dürfen unter keinen Umständen entfernt werden.

2.4.2.2 dk.brics.automaton

dk.brics.automaton [**?**] ist von Anders Møller am BRICS research center der Universität von Aarhus entwickelt worden.
Es handelt sich hierbei um eine Bibliothek, die DFA/NFA-Automaten und reguläre Ausdrücke implementiert. Diese Bibliothek wird ständig weiterentwickelt. Die letzte verfügbare Version wurde am 8. August 2005 veröffentlicht und ist bereits an Java 1.5 angepaßt. Ältere Versionen für Java 1.4 sind ebenfalls erhältlich.
Der Sourcecode ist nutzbar unter den Bedingungen der Open Source BSD Lizenz.

Funktionsumfang: Mit Hilfe von dk.brics.automaton lassen sich Automaten auf zwei Arten modellieren. Zum einen kann man einen Automaten durch explizite Angabe der Zustände und Zustandsübergangsfunktion modellieren. Auf diese Art lassen sich auch NFAs implementieren. Die zweite Möglichkeit besteht darin, einen Automaten durch die Angabe eines regulären Ausdrucks erzeugen zu lassen.

Automat: Auf Automaten sind eine Vielzahl weiterer Operationen erlaubt: Es stehen drei Algorithmen zur Minimierung von Automaten zur Verfügung: Brzozowski's $O(2^n)$; Hopcroft's O(n log n); Huffman's $O(n^2)$, Komplement, Produktsprache, Quotientensprache, Entfernen von überflüssigen Zuständen, Vereinigung von Automaten, Kleene'scher Abschluss von Automaten, Konvertieren eines NFA-Automaten in einen DFA-Automaten.

Fazit: dk.brics.automaton ist eine sehr kleine und übersichtliche Automatenbibliothek. Sie besteht aus sechs Klassen, was sie sehr übersichtlich und verständlich macht. Sie erfüllt so gut wie alle Ansprüche an eine Automatenbibliothek und arbeitet darüber hinaus sehr effizient.

Sehr positiv fällt außerdem auf, dass ein Automat aus Zuständen und Zustandsübergangsfunktion identisch zum Automaten seines regulären Ausdrucks ist. Dies klingt selbstverständlich, ist aber bei allen anderen Bibliotheken leider nicht der Fall.

2.4.2.3 JRexx

Die Automatenbiliothek JRexx [**?**] ist von der karneim.com - Gesellschaft für Softwarearchitektur mbH implementiert worden, um ein Pattern Matching unter Zuhilfenahme von Automaten durchzuführen.
Dabei wurden DFA und NFA implementiert, sowie die Erzeugung eines Automaten durch einen gegebenen regulären Ausdruck. Es handelt sich hierbei um ein Open Source Produkt unter der LGPL.

Funktionsumfang: Wie bei dk.brics.automaton lassen sich Automaten auf zwei Arten modellieren. Zum einen kann man einen Automaten durch explizite Angabe der Zustände und Zustandsübergangsfunktion modellieren. Die zweite Möglichkeit ist es, einen Automaten durch die Angabe eines regulären Ausdrucks erzeugen zu lassen.

Automat: Auf Automaten sind eine Vielzahl weiterer Operationen erlaubt: Automatenminimierung (ohne Angabe des verwendeten Algorithmus), Komplement, Produktsprache, Quotientensprache, Vereinigung von Automaten, Zustandsüberwachung.

Fazit: Diese Bibliothek ist primär entwickelt worden, um ein Pattern Matching optimal zu unterstützen. Aus diesem Grund ist die Bibliothek zu stark mit speziellen Funktionalitäten überladen. Dieses macht die gesamte Skriptbibliothek sehr unübersichtlich. Weitere schwerwiegende Nachteile bilden die Ausgaben(System.out.println(autm)), die trotz intensiver Bemühungen nicht klar verständlich werden.
Es gibt keine Funktion, die eine Überführung eines NFA-Automaten in ein DFA-Automaten ermöglicht.
Als einzige positive Neuerung bleibt die Implementierung eines ChangeListeners zu nennen, die es möglich macht, alle Aktionen in einem Automaten zu überwachen.

2.4.2.4 JAT

Das Java Automata Toolkit [**?**] bildet die Basis für das Automaten-Tool Java Computability Toolkit und wurde 1997 von Matthew B. Robinson am Department of Computer Science SUNY Institute of Technology für Java 1.2 entwickelt.
Die Bibliothek ermöglicht das Arbeiten mit DFA/NFA Automaten, regulären Ausdrücken und Turingmaschinen, auf die ich in der weiteren Ausarbeitung aber nicht näher eingehen werden.

Funktionsumfang: Wie in den bisher besprochenen Bibliotheken lassen sich auch hier Automaten auf zwei Arten modellieren. Zum einen kann man einen Automaten durch explizite Angabe der Zustände und Zustandsübergangsfunktion modellieren. Die zweite Möglichkeit stellt eine Modellierung eines Automaten durch die Angabe eines regulären Ausdrucks dar.

Automat: Automaten im System können unter Zuhilfenahme folgender Funktionalitäten verändert und bearbeitet werden: Komplement, Produktsprache, Quotientensprache, Minimierung von Automaten, Vereinigung von Automaten, Vergleich von Automaten, Konvertieren eines NFA-Automaten in einen DFA-Automaten

Fazit: Der größte Kritikpunkt an JAT ist die Implementierung, die nicht wirklich objektorientiert ist. Die Übergangsfunktion wird durch Angabe von Knotennummern realisiert. Der Grund dafür liegt darin, dass ursprünglich eine C Version implementiert wurde, die dann auch nach Java konvertiert wurde.
Um diese Bibliothek unter Java 1.4.2 lauffähig zu machen, mussten einige Änderungen durchgeführt werden.
Außerdem fällt negativ auf, dass der Automat auf Basis eines regulären Ausdrucks vom Automaten aus Zuständen und Zustandsübergangsfunktion abweicht.

2.4.2.5 JFlap

JFlap [**?**] ist von allen angegebenen Bibliotheken die funktional umfassenste Codesammlung. Mit JFlap lassen sich DFA, NFA, reguläre Ausdrücke, Grammatiken, PDA und TM implementieren. Sie wird aktuell von Susan H. Rodger Associate Professor of the Practice Department of Computer Science Duke University, Durham, betreut.
Hierbei handelt es sich um eine laufende Entwicklung, die 1990 begonnen wurde.

Funktionsumfang: Auch hier beschränken wir uns wieder auf Automaten und reguläre Ausdrücke. Zum einen kann man einen Automaten mittels Zuständen und Zustandsübergangsfunktion modellieren. Leider ist es trotz intensiver Bemühungen nicht gelungen, einen Automaten mittels eines gegebenen regulären Ausdrucks zu konstruieren.

Automat: Auf Automaten sind eine Vielzahl weiterer Operationen erlaubt: Automatenminimierung, Komplement, Produktsprache, Quotientensprache, Entfernen von überflüssigen Zuständen, Vereinigung von Automaten, Vergleich von Automaten, Konvertieren eines NFA-Automaten in einen DFA-Automaten

Fazit: Auf der einen Seite hat JFlap eine Menge Vorteile. Zum einen ist die Bibliothek sehr umfangreich und implementiert eine Menge weiterer Konstrukte wie PDAs und Grammatiken. So ist es zum Beispiel möglich, sich zu einem gegeben Automaten eine Grammatik erzeugen zu lassen.
Außerdem wird immer noch an JFlap weiterentwickelt und es ist bereits eine Java 1.5 Version verfügbar. JFlap bietet einen guten Editor, für den die Bibliothek primär entwickelt wurde. Dieser Editor ist aber auch bzgl. der flexiblen Nutzung, da es keine Trennung zwischen Modell und View gibt, sein größter Nachteil. Dieses kann man gut an den Zuständen sehen. Im Konstruktor eines Zustandes muss eine Position angegeben werden. Auf dieser kann er dann im Editor angezeigt werden.
Ein kleiner Nachteil ist die Unübersichtlichkeit der Programmierung.

2.4.2.6 Fazit

Diese Recherchen zeigen, dass sich das Engagement großteils um DFA/NFA-Automatenbibliotheken dreht. Andere Automatenimplementationen, wie Büchi- oder Baumautomaten sind nicht zu finden. Von allen Bibliotheken ist dk.brics.automaton die am besten umgesetzte und könnte als Basis einer Eigenentwicklung oder Weiterentwicklung genutzt werden. JFlap und JRexx könnten für eine GUI interessant sein, um Ideen für eine grafische Umsetzung zu sammeln.

2.5 Werkzeuge

2.5.1 jABC and Friends

In einer Welt, in der immer mehr Aufgaben mit Rechnerunterstützung bewältigt werden und in der es daher einen großen Bedarf an Programmen gibt, wäre es wünschenswert, dass man neben den eigentlichen Programmierexperten – von denen es nicht genug gibt – noch andere Gruppen an der Entwicklung von Programmen beteiligen könnte.

Diese Beteiligung muss auf einer hohen Ebene stattfinden, auf der Implementierungsdetails ausgeblendet sind, und auf der es nur um die Eigenschaften sowie das nach außen sichtbare Verhalten der zu entwickelnden Applikation geht.

Im Folgenden werden einige Ansätze beschrieben, die eine solche „programmierungsfreie Programmierung“ ermöglichen. Dabei geht es darum, fertige und wiederverwendbare Programmeinheiten zu Applikationen zu kombinieren, wobei Modelle benutzt werden, die eine „High-Level“-Sicht bieten. Das sind zudem Modelle, die abstrakt genug sind, um eine Verifikation des entwickelten Programms zu ermöglichen.

2.5.1.1 Metaframe

2.5.1.1.1 Programmierungsfreies Programmieren Das Metaframe-System (vgl. [**?**]) konkretisiert die in der Einleitung motivierte „programmierungsfreie Programmierung“. Die wiederverwendbaren Programmeinheiten heißen *building blocks* (BB) oder *service independent building blocks* (SIB). Sie bestehen aus klassischen Programmfragmenten (Prozeduren, Module, Funktionen, Klassen) und werden im Metaframe-System als Knoten dargestellt. Benutzer kombinieren diese Knoten zu Graphen, wobei die Kanten gemäß des Verhaltensaspekts konstruiert werden, das heißt sie sind gerichtet und geben die Reihenfolge wieder, in der die den BB zugrunde liegenden Programmfragmente ausgeführt werden sollen.

2.5.1.1.2 BB-Graphen Der innerhalb eines solchen Fragments erfolgende Programmfluss führt – abhängig von Kontext und Eingabe – zu einem bestimmten Ergebnis. Dieses Ergebnis legt fest, an welcher Stelle im Programmablauf fortzufahren ist. An einem BB teilt man (konzeptionell) die Menge möglicher Ergebnisse in Äquivalenzklassen ein, wobei zwei Ergebnisse genau dann in derselben Äquivalenzklasse liegen, wenn sie zur Fortsetzung des Programmablaufs an gleicher Stelle führen sollen. Für jede dieser Äquivalenzklassen erhält ein BB eine benannte ausgehende Kante, Die Kante führt zu dem BB, der das Programmfragment repräsentiert, an dem fortgesetzt werden soll.

Der auf diese Weise konstruierte (Fluss-)Graph kann einerseits dazu verwendet werden, einen Prototypen oder ein fertiges Programm zu kompilieren, andererseits bietet er eine Sicht auf das entstehende Programm als Transitionssystem. Programmeinheiten sind auf ihr Ein-/Ausgabeverhalten reduziert, ihr Einwirken auf den Kontext ist ausgeblendet. Daher stellt der (Fluss-)Graph im Allgemeinen nur Äquivalenzklassen der Zustandsmenge des entstehenden Programms dar.

Dabei bleibt offen, in welcher konkreten Programmiersprache die zu den BB korrespondierenden Programmfragmente implementiert werden. BBs können auch bereits als Spezifikation existieren, ohne dass es eine Implementierung gibt.

2.5.1.1.3 Verifikation durch Model Checking Der BB-Graph kann als Kripkemodell[1] interpretiert werden, in dem die BBs Zustände sind, in denen Mengen von atomaren Aussagen gelten oder nicht gelten. Die benannten Kanten bzw. *branches* zwischen den BBs entsprechen den *actions*. Dieses Modell erlaubt es, temporallogische (unendliche Folgen von BBs betreffende) Aussagen über den BB-Graphen zu formulieren. Diese Aussagen entsprechen Anforderungen an die Art und Weise, wie BBs ausgeführt werden. Es ist unter anderem möglich, zu zu beschreiben,

- in welcher Reihenfolge BBs ausgeführt werden,
- dass irgendein BB (unter einer Bedingung) irgendwann einmal ausgeführt werden muss,
- dass zwei bestimmte BBs niemals gemeinsam während einer Ausführung des Systems auftauchen.

Zu einer Anwendung können Hunderte solcher Aussagen bzw. *constraints* formuliert werden, die während des Entwicklungsprozesses ständig effizient überprüfbar sind (*Model Checking*). Wird ein *constraint* verletzt, gilt eine Aussage also nicht, ist das ein sicherer Hinweis darauf, dass ein Entwicklungsschritt unternommen wurde, der nicht mit der Spezifikation verträglich ist.

2.5.1.1.4 Entwicklergruppen Weiterhin erlaubt das Metaframe-System, den Softwareentwicklungsprozess auf vier Gruppen zu verteilen.

- **Programmierer** sind zuständig für die Implementierung der BBs und für die Laufzeitumgebung, in der die kompilierte Applikation ausgeführt wird.

[1] Ein Kripkemodell über einer Menge AP atomarer Propositionen ist ein Quadrupel $T = (S, Act, \rightarrow, I)$, wobei S eine Zustandsmenge ist, Act eine Menge von *actions*, $\rightarrow \subset S \times Act \times S$ eine Zustandsübergangsrelation und $I : S \rightarrow \mathfrak{P}(AP)$ eine Interpretation.

- **Constraints-Modellierer** haben die Aufgabe, Anforderungen an die Applikation als temporallogische *constraints* zu formulieren.
- **Anwendungsexperten** bauen SIBs zu vollständigen Applikationen zusammen.
- **Endbenutzer** können, ohne dass Expertenwissen notwendig ist, die Applikation modifizieren, wobei selbstverständlich nur mit den vorhandenen *constraints* zu vereinbarende Änderungen erlaubt sind.

2.5.1.2 Agent Building Center

Das *Agent Building Center*, kurz *ABC*, ist ein Nachfolger des Metaframe-Systems. ABC besteht aus einer in Tcl/Tk entwickelten Benutzeroberfläche, mit der aus SIBs bestehende Graphen erstellt, bearbeitet und überprüft werden können.

2.5.1.2.1 SIBs - service independent building blocks Zu einem SIB gehören vier Stücke Code.

- SIB-Definition: Name, Klasse, Parameter, Branches (ausgehende Kanten)
- Red-line Code: Prototypischer Code, der das Verhalten des SIBs auf abstrakter Ebene implementiert.
- Local Check Code: Wird benutzt, um einzelne SIBs zu überprüfen
- Implementierung: Zum Beispiel in Java oder C++ geschriebener Code, der benutzt wird, wenn der SIB-Graph zu einem Programm kompiliert wird.

2.5.1.2.2 ABC-Komponenten SIB-Graphen werden jeweils innerhalb einer Umgebung (service definition environment) entwickelt. Die Umgebung stellt in Klassen eingeteilte SIBs zur Verfügung.

Der *local checker* ermöglicht eine einfache lokale Fehlererkennung, indem für einzelne SIBs überprüft wird, ob sie bestimmten Anforderungen genügen. Beispielsweise kann geprüft werden, ob alle Parameter gesetzt sind oder ob die Kanten richtig benannt sind. Außerdem können SIBs mit einem speziellen vom *local checker* prüfbaren *local check code* annotiert werden.

Der *tracer* dient der Fehlererkennung durch Testen. Er ermöglicht eine Ausführung des SIB-Graphen, wobei der prototypische *red-line code* ausgeführt wird.

Der *model checker* dient der globalen Fehlererkennung. Für jede Umgebung gibt es eine constraints-Datenbank, die Mengen temporallogischer Formeln erhält, die (siehe 2.5.1.1.3) spezifizieren, wie ein SIB-Graph für die Umgebung auszusehen hat. Ob die *constraints* verletzt sind, kann per Mausklick überprüft werden. Im Fehlerfall kann eine „Fehlersicht“ auf den Teilgraphen angezeigt werden, in dem die Ursache für die Verletzung des *constraints* liegt.

2.5.1.3 Java ABC

2.5.1.3.1 Java als Programmier- und Spezifikationssprache Ein neuer Ansatz, das Metaframe-System zu realisieren, ist Java ABC bzw. jABC (vgl. [**?**]), eine vollständig in Java geschriebene Modellierungsumgebung und Integrationsplattform.

Auch in Java ABC dreht sich alles um das Kombinieren von SIBs zu flussgraphähnlichen Strukturen. Anders als in ABC erfolgt die Spezifikation der SIBs nicht mehr in verschiedenen Formaten, sondern einheitlich in Java. SIBs sind Instanzen von Java-Klassen. Die Unabhängigkeit von der Sprache, in der ein aus einem SIB-Graphen kompiliertes Programm implementiert ist, geht damit aber nicht zwangsläufig verloren. SIBs sollen am besten keine eigene Programmlogik enthalten und in erster Linie als Träger von Attributen dienen. Es ist zum Beispiel weiterhin vorstellbar, dass ein SIB dazu da ist, um Parameter für den Aufruf einer Methode einer anderen Programmiersprache zu speichern.

In Java ABC hat ein SIB

- **Attribute**, sie sind als `public` *fields* realisiert.
- **ausgehende Kanten**, realisiert als `public String []` Array.
- **eine eindeutige ID (UID)**, das ist ein `public final` Attribut vom Typ `String`, also ein Instanzattribut, das aber für alle Instanzen einer SIB-Klasse gleich ist. Die UID ermöglicht eine zuverlässigere Identifikation eines SIB als der (veränderliche) Klassenname.
- **eine *getIcon*-Methode**, diese Methode ist im Interface *SIBClass* spezifiziert.
- **eine *getTooltipText*-Methode**, die Informationen liefert, die auf der Zeichenfläche des SIB-Graphen als „Tooltip“ sichtbar werden können.

Die SIB-Klassen werden nicht unmittelbar beim Programmstart geladen, sondern erst beim Öffnen eines jABC-Projektes, und zwar mittels eines speziellen `ClassLoader`, der auf dem jetzigen Implementierungsstand noch Unterverzeichnisse des Projektverzeichnisses

nach SIBs durchsucht. Später sollen SIB-Klassen aber auch über das Netzwerk aus einem zentralen Repository mit Versionskontrolle und Zugriffsrechten bezogen werden können.

2.5.1.3.2 jABC-Komponenten Die Modellierungsumgebung und Integrationsplattform besteht im Wesentlichen aus den folgenden Bestandteilen.

- **Baumansicht der Projekte** und der innerhalb des aktuellen Projekts verfügbaren SIBs.
- **Zeichenfläche**, auf der SIBs zu Graphen kombiniert werden können. Dabei wird auf die Graphbibliothek `jgraph` (`http://www.jgraph.org`) aufgesetzt. Graphen werden im GXL-Format abgespeichert, einem XML-Dialekt, in dem die visuelle Struktur des Graphen repräsentiert wird und der es erlaubt, Knoten mit beliebig vielen Zusatzinformationen (Labels) zu versehen.

  Wird solch eine GXL-Datei geladen und enthält sie einen Eintrag für ein SIB, dessen Klasse nicht verfügbar ist, so wird der SIB durch einen Platzhalter dargestellt (Proxy-SIB). Es ist so weiterhin möglich, SIB-Attribute zu ändern und auch zu speichern.

  Das Einstellen der SIB-Parameter erfolgt mit dem SIB-Inspektor. Plugins können zusätzliche Inspektoren hinzufügen, mit denen Pluginfunktionalität gesteuert wird.
- Die **Plugin-Schnittstelle** ermöglicht die Erweiterung der Modellierungsumgebung um Plugins. Erst durch Plugins erhalten SIBs eine Bedeutung, in der bloßen Modellierungsumgebung sind sie nur leere Entitäten, deren Attribute man ändern kann.

  Für jedes Plugin wird eine lokal verfügbare Plugin-Klasse benötigt. Beim Start von Java ABC wird von jeder bekannten Plugin-Klasse eine Instanz erzeugt und auf dieser Instanz die `start`-Methode aufgerufen. Also wird pluginspezifischer Code ausgeführt, in dem zum Beispiel Menü-Einträge erzeugt werden können, über die ein Benutzer Pluginfunktionalität aktivieren kann.

2.5.1.4 Schlussbemerkungen

Das Paradigma der „programmierungsfreien Programmierung“ umzusetzen ist eine anspruchsvolle Aufgabe. Die vorgestellten, im Bereich der universitären Forschung entwickelten Tools sind zumindest hoffnungsvolle “proofs of concept“.

Der Einsatz von jABC ist sinnvoll, wenn gut strukturierte workflows zu modellieren sind; oder wenn man Code ausführen möchte, der in isolierte Teile mit einfacher Schnittstelle zerlegt ist – so dass die mit den Kanten modellierten Übergänge nicht zu sehr an der

Wirklichkeit vorbeigehen – und die auf viele verschiedene Weisen miteinander kombiniert werden können. Dann bietet sich die Benutzung des Tracers an.

Außerdem hat jABC eine relativ leicht zu bedienende grafische Oberfläche und API zur Erstellung und Bearbeitung von Flussgraphen, so dass, wann immer eine solche Funktionalität benötigt wird, ein Einsatz von jABC erwogen werden sollte; insbesondere dann, wenn sich abzeichnet, dass bei diesem Einsatz von anderen jABC-Benutzern wiederverwendbare Resultate entstehen.

2.5.2 Das Eclipse Project

Das Ziel des Eclipse Project ist es, eine offene Entwicklungs- und Anwendungsplattform zu entwickeln und zur Verfügung zu stellen. „Offen“ steht hier einerseits für einen vielseitigen Einsatzbereich, aber hauptsächlich auch für die Tatsache, dass es sich beim Eclipse Project um ein Open Source-Produkt handelt.

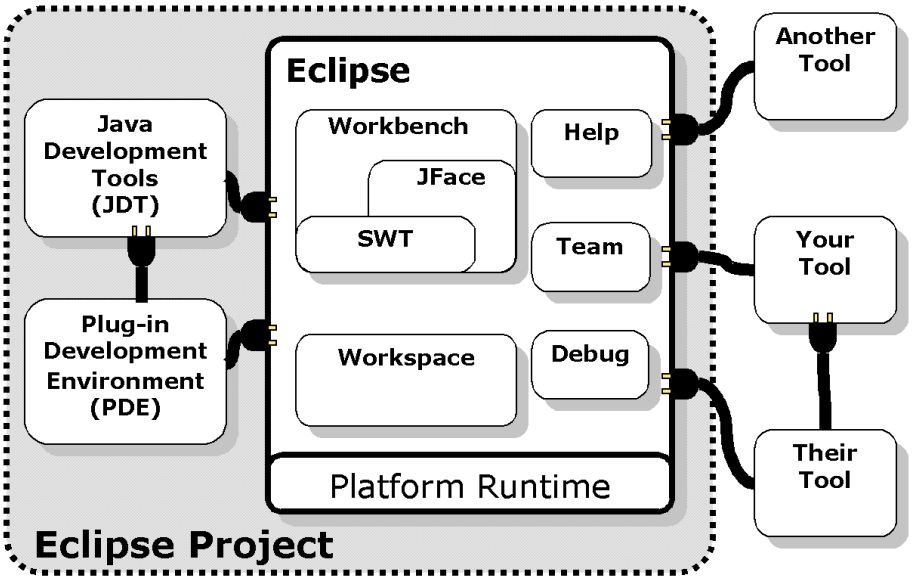


Abbildung 2.10: Überblick über das Eclipse Project (siehe auch [**?**])

2.5.2.1 Lizenzmodell

Das Eclipse Project wird vom Konsortium „Eclipse Foundation“ unter der *Common Public License* herausgegeben. Gegenwärtig wechselt das Projekt auf die *Eclipse Public License*, was jedoch inhaltlich keinen Unterschied bedeutet: Der Sourcecode ist ohne Lizenzgebühren erhältlich und kann frei verteilt und auch kommerziell genutzt werden. Lediglich der Eclipse Code selbst und auch alle Modifikationen daran müssen, auch bei kommerzieller Weiterverwendung, frei erhältlich bleiben. Eigene Module jedoch brauchen nicht

veröffentlicht zu werden.

2.5.2.2 Das Plug-in-Konzept

Die einfache Erweiterbarkeit des auf Java basierenden Projekts wird durch ein umfangreiches Plug-in-Konzept erreicht. Eclipse besteht aus einem nur relativ kleinen Kern, der jedoch *Erweiterungspunkte*, sogenannte *extension points* zur Verfügung stellt, in denen sich *Erweiterungen*, auch *extensions* genannt, einklinken können. Plug-ins sind Sammlungen von solchen Erweiterungen, stellen aber ihrerseits auch wieder Erweiterungspunkte zur Verfügung, so dass ein Geflecht von sich gegenseitig nutzenden Plug-ins zum Eclipse Projekt beiträgt.

Der Kern von Eclipse selbst besteht aus:

- der *Plattform-Laufzeitumgebung*
- dem *Workspace*, einer Art Hauptverzeichnis für von Eclipse verwaltete Projekte
- der *Workbench*, einer Umgebung mit Editoren, Ansichten, Aktionen etc.

Alle weiteren Funktionalitäten (und bereits auch viele Bestandteile der oben genannten Hauptkomponenten) sind als Plug-in realisiert. Dazu gehört auch die *Java-Entwicklungsumgebung (JDT)*, welche zusammen mit der *Plug-in-Entwicklungsumgebung (PDE)* zusammen mit Eclipse ausgeliefert wird und eine vollständige Umgebung zur Entwicklung und Erweiterung von Eclipse selbst zur Verfügung stellt.

2.5.2.2.1 Funktionsweise von Plug-ins Plug-ins bestehen aus einem deklarativen und einem implementierenden Teil. Der deklarative Teil besteht hauptsächlich aus einer Datei `plugin.xml`, welche beim Starten von Eclipse gelesen wird und den Eclipse-Kern über die vom Plug-in verwendeten bzw. zur Verfügung gestellten *Erweiterungspunkte* informiert. Das folgende Listing zeigt eine beispielhafte `plugin.xml`, welche dann näher erläutert wird.

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.0"?>
<plugin>
 <extension point="org.eclipse.ui.actionSets">
  <actionSet id="HelloWorld.actionSet"
             label="Hello World Action Set"
             visible="true">
```

```
    <menu id="HelloWorld.menu"
          label="Hello World Menü">
      <separator name="BeispielGruppe"/>
    </menu>
    <action
      class="helloWorld.actions.HelloWorldAction"
      id="HelloWorld.action"
      label="Hello World"
      menubarPath="HelloWorld.menu/BeispielGruppe"
      toolbarPath="BeispielGruppe"
      tooltip="Hallo Welt :-)"
    />
  </actionSet>
 </extension>
</plugin>
```

Die vorliegende Deklaration teilt über das Tag `<extension point>` mit, dass es den Erweiterungspunkt `org.eclipse.ui.actionSets` nutzen möchte. Dieser Erweiterungspunkt ist für das Einbinden von Aktionen in Menüs, Toolbars etc. verantwortlich. Die unter `<extension point>` geschachtelten Optionen teilen jetzt diesem Erweiterungspunkt mit, wie er verwendet werden soll. Die Beispiel-Deklaration erzeugt ein neues Menü „Hello World Menü“ und fügt diesem einen Eintrag mit dem Label „Hello World“ hinzu. Dieser ist im Tag `<action>` deklariert und verweist mit dem `class`-Attribut auf die Java-Klasse `helloWorld.actions.HelloWorldAction`, welche somit den implementatorischen Teil dieses Plug-ins enthält:

```
public class HelloWorldAction
               implements IWorkbenchWindowActionDelegate {
  private IWorkbenchWindow window;
  public void dispose() { // empty      }

  public void init(IWorkbenchWindow window) {
    this.window = window;
  }

  public void run(IAction action) {
         MessageDialog.openInformation(window.getShell(),
                      "Hello World plug-in", "Hello World!!");
  }
```

```
  public void selectionChanged(IAction action,
                               ISelection selection) {
    // empty
  }
}
```

Das `class`-Attribut einer `Action` erwartet eine Klasse, welche das Interface `org.eclipse.ui.IworkbenchWindowActionDelegate` implementiert. Dessen wichtigste zu implementierende Methode ist `public void run(IAction)`, welche beim Aufruf der Aktion, also hier durch Auswählen des neu erzeugten Menü-Eintrages, gestartet wird.

2.5.2.3 SWT / JFace

SWT, das *Standard Widget Toolkit*, ist ein mit Suns *AWT* und *Swing* vergleichbares Framework zur Entwicklung von graphischen Oberflächen in Java. Unter *Widgets* versteht man Steuerelemente wie Buttons, Listen, Textfelder usw. Im Gegensatz zu *AWT* und *Swing* benutzt *SWT* native, also betriebssystemabhängige Widgets, was die Anwendung von *SWT* für den Programmierer etwas umständlicher in Bezug auf Betriebssystemressourcen und deren Benutzung und Freigabe macht.

Eclipse basiert auf *SWT* und nicht Suns *Swing/AWT-Framework*. Daher müssen auch sämtliche graphischen Elemente, die man per Plug-in in Eclipse einbauen möchte, ebenfalls auf *SWT* basieren. Die Gründe für den Einsatz von *SWT* liegen in der Historie der Entwicklung von Java und haben ihre Wurzeln in den Unzulänglichkeiten von *AWT* und den Anfangsproblemen von Suns *Swing*.

Die Verwendung von *SWT* stellt für Swing-gewohnte Programmierer eine Umstellung dar, die jedoch durch Eclipse *JFace* abgefedert wird. *JFace* ist eine Sammlung von Klassen und Tools, die auf *SWT* aufsetzt und typische Aufgaben des Benutzer-Interface von Eclipse umsetzt. Dazu zählen Frameworks für Dialoge, Eigenschaften und Wizards etc., sowie auch Views, Editoren und Actions. Ein Beispiel für den Einsatz von JFace war bereits der MessageDialog-Aufruf im HelloWorld-Beispiel. *JFace* setzt also, jedoch ohne es dabei zu verdecken, auf *SWT* auf und bietet eine komfortablere Schnittstelle zur Programmierung von Workbench-Komponenten.

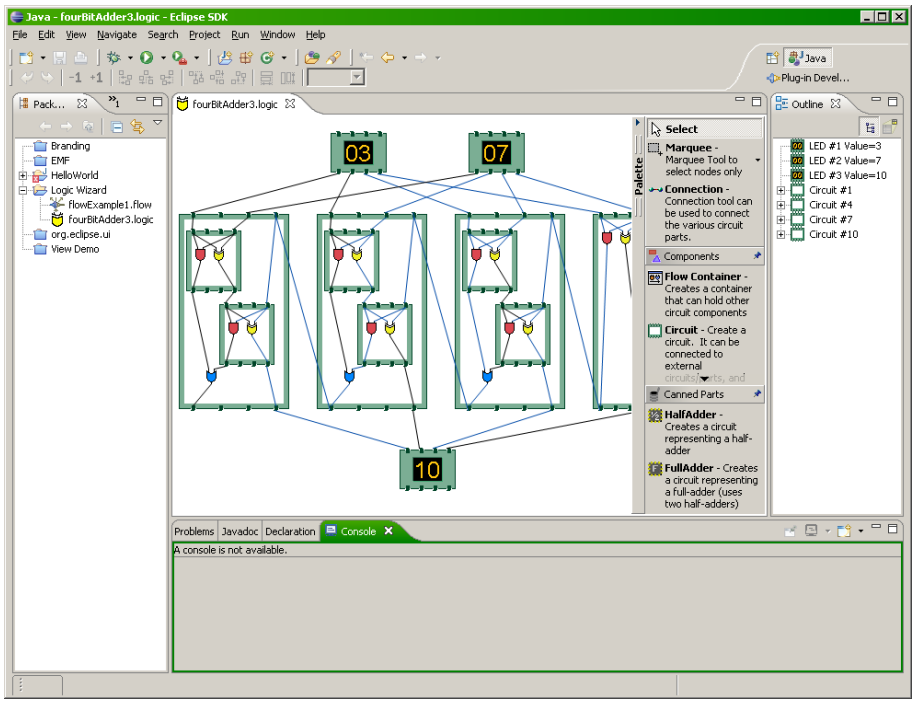


Abbildung 2.11: Beispiel eines mit GEF realisierten Editors

2.5.2.4 Das Graphical Editing Framework (GEF)

Das *Graphical Editing Framework* ist ein Eclipse-Projekt, welches die Entwicklung von graphischen Editoren für beliebige Anwendungsmodelle vereinfachen soll. Das *GEF* besteht aus zwei Teilkomponenten. Zum einem ist das *Draw2D*, eine auf *SWT* aufsetzende Bibliothek zum Zeichnen und Layouten von eigenen Komponenten und zum anderen das eigentliche *GEF* selbst, welches die Views von Eclipse um einen graphischen Editor erweitert und wesentliche Funktionen für graphisches Bearbeiten von Modellen zur Verfügung stellt.

GEF unterstützt dabei unter anderem folgende Funktionen:

- Erzeugen, Löschen, Verschieben, Verbinden, Verändern von Objekten
- Werkzeug- & Komponenten-Paletten, Drag & Drop, Undo
- Überblicks-Ansichten und Zoom
- Lineale, Gitternetz
- automatische Verbindungen
- ...

Wie bei den Views, verwendet auch das *GEF* das *MVC-Modell*, trennt also das Modell (weitgehend) von der View. Über Policies und Commands wird das Verhalten von *GEF*-Anwendungen gesteuert. Hier wird ein umfangreiches Netzwerk von Actions und Listenern eingesetzt. Weitere Informationen zu diesem umfangreichen, komplexen, aber leistungsstarken Framework gibt es unter der Eclipse-GEF-Projektseite [**?**], im IBM Redbook „Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework“ [**?**] und in Form von zahlreichen weiteren Artikeln und Tutorials im Web.

2.5.2.5 Rich-Client Platform und Branding

Die meisten Benutzer kennen Eclipse nur als Java-Entwicklungsumgebung. Für viele Anwendungen sind aber zum Beispiel die standardmäßig integrierten Funktionen des Java-Editors überflüssig und störend. Daher ist es möglich, Plug-ins in eine reduzierte Eclipse-Umgebung zu packen, die nur die notwendigsten Kern-Komponenten enthält. Dadurch kann man eine vollkommen selbstständige und individuelle Anwendung mit eigenem Titel, Splash-Screen etc. erstellen.

2.5.2.6 Fazit

Die Eclipse-Plattform stellt eine umfangreiche Umgebung für Anwendungen zur Verfügung, welche vorrangig auf die Bearbeitung von textuell erfassbaren Modellen und deren Anzeige in verschiedensten Ansichten ausgelegt ist. Durch Plug-ins kann man eigene Erweiterungen unter Zuhilfenahme von bereits existierenden Editoren und Ansichten entwickeln.

Gute Java-Kenntnisse sind für die Entwicklung von Eclipse Voraussetzung. Objektorientierte Prinzipien und bekannte Design-Patterns spielen eine wichtige Rolle. Zu berücksichtigen ist, dass Eclipse auf der SWT-Grafik-Bibliothek aufbaut und nicht kompatibel mit dem Java-Standard Swing ist. Die Dokumentation ist gut und umfangreich, für Anfänger aber sicherlich zunächst „schwer verdaulich“. Tutorials und Literatur zum Thema verfolgen meistens die Strategie „Lernen durch Beispiele“, das Zitat „Always start by copying the structure of a similar plug-in.“ aus [**?**] sei hier beispielhaft genannt.

Schlussendlich bleibt also festzuhalten, dass Eclipse ein mächtiges Werkzeug ist und für fast jede Anwendung, in der beliebige Daten manipuliert, visualisiert und in sonstiger Weise verarbeitet werden, geeignet ist. Die Erstellung dieser Anwendungen bedarf eines

gewissen Einsatzes von Kenntnissen und damit einer hohen Einarbeitungszeit, was aber am Ende mit einer erweiterbaren, auf verbreiteten Standards basierenden Applikation belohnt wird, an der, sofern man sich selbst an Dokumentations- und Implementationsstandards hält, auch andere weiterentwickeln können.

Kapitel 3

Gruppenergebnisse

Die Erarbeitung des theoretischen Hintergrunds war der erste Schritt bei der Durchführung des Projektes. Danach mussten vor dem Hintergrund des neu erworbenen Wissens klare Ziele formuliert werden. Die Verwirklichung dieser Ziele bedeutet die Aufteilung der Problemfelder, um ein klar voneinander getrenntes und strukturiertes Arbeiten zu ermöglichen.

In seiner gröbsten Form gibt es drei ganz klar definierte Teile: die Analyseverfahren, die dafür benötigte Automatenbibliothek und eine grafische Unterstützung dieser beiden Komponenten. Diese Teile lassen sich ebenfalls aufteilen.

Transformationen:

- Presburger-Arithmetik
- LTL Model Checking
- While-Sprache

Automatenbibliothek, modular aufgebaut durch Vererbung und ausgelagerte Operationen:

- der abstrakte Kern
- endliche Automaten
- Büchi-Automaten
- OBDDs
- Kripkestrukturen

Grafische Unterstützung, modular aufgebaut durch Plugins:

- Viewer für Automaten
- Editor für Automaten
- Verschiedene Ein- und Ausgaben für Analysen
- Workspace als Zusammenfassung der Komponenten

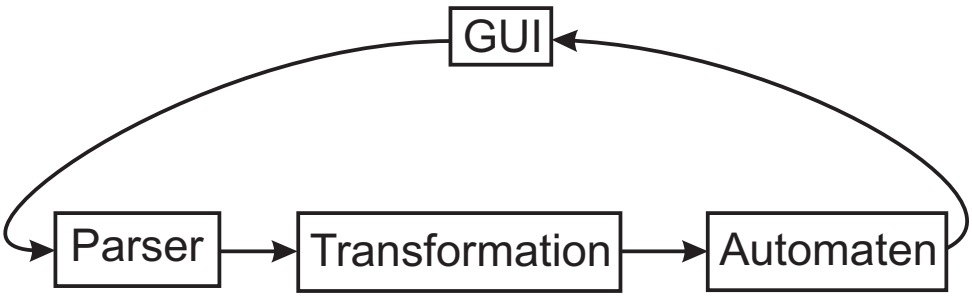
Ferner muss man noch eine vierte Komponente berücksichtigen: Eine Hilfskomponente in Form eines Parsers. Dieser ist notwendig, um für Analyseverfahren geeignete Eingaben zu gewinnen.

Es war nötig, Folgendes mit einem Parser einlesen zu können:

- Presburger Formeln
- temporallogische Formeln
- (omega-) reguläre Ausdrücke

Bei den Komponenten besteht eine gewisse Abhängigkeit. Analysen nutzen Parser und Automaten, die grafische Oberfläche stellt Analysen mit und auf Automaten bereit und ist somit auch zum Testen nützlich.

Insgesamt lassen sich diese Abhängigkeiten der Komponenten strenger als Pipeline auffassen.



Die Zusammenarbeit dieser Komponenten ergibt das endgültige Produkt.

3.1 Transformation

In diesem Kapitel wird der Code für alle im Verlauf der PG realisierten automatenbasierten Analysen von Formeln und Programmen beschrieben. Die Implementationen sind im Package `aaa.translation` zu finden. Begonnen wird mit einer Erläuterung des die Arbeit der Analysen unterstützenden Hilfs-Package `aaa.translation.util`. Hier geht es insbesondere um die Klasse `BPNode`. Im Anschluss daran folgt eine Beschreibung der Transformation von Presburger-Formeln in endliche Automaten, welche die Lösungsmengen der Formeln repräsentieren. Danach wird erklärt, wie die Klasse `KripkeLTLModelChecker` das Model Checking von Kripkemodellen mit LTL-Formeln basierend auf Büchiautomaten realisiert. Dann wird die Übersetzung von LTL-Formeln in Büchiautomaten, welche die Menge der die Formeln erfüllenden Pfade darstellen, thematisiert. Das Kapitel schließt mit einem Abschnitt über die Überführung von While-Programmen in Kripkemodelle, die die möglichen Kontrollflüsse der Programme modellieren.

3.1.1 `aaa.translation.util`

Das Package `aaa.translation.util` hält Hilfsklassen bereit, die die Arbeit der Analysen unterstützen.

3.1.1.1 `BPNode`

3.1.1.1.1 Zweck von `BPNode` Aus Instanzen der Klasse `BPNode` können binäre Ausdrucksbäume für Formeln aufgebaut werden. Sowohl die Presburger-Transformation als auch die LTL-Transformation übersetzen den vom Parser gelieferten aus `PNode`-Instanzen bestehenden Ausdrucksbaum in einen äquivalenten Baum aus Instanzen einer Unterklasse von `BPNode`. Die Klasse `PNode` ist inzwischen sehr allgemein und erlaubt beliebige Knotengrade. Ihre `toString()`-Methode kann nicht für alle Sprachen sinnvolle bzw. erwünschte Ergebnisse liefern (z. B. Sprache der Presburger-Formeln). `BPNode` wurde eingeführt, um auf binären Ausdrucksbäumen operierenden Code lesbarer zu machen. Es gibt Methoden wie `getLeftChild()` und `setLeftChild(BPNode)`, und das Token ist ein einfacher `String`. Ausserdem hält die Klasse ein Label des Typs `Object` bereit.

3.1.1.1.2 Unterklassen Die einzelnen Transformationen können weitere Anforderungen an die Knotenklasse stellen, die von Unterklassen erfüllt werden können. In `PresburgerBPNode` und `LTLBPNode` existieren jeweils `is...`- und `make...`-Methoden, um den Knotentyp (das Token) zu ermitteln oder zu ändern. In `LTLBPNode` wird sogar die

`equals`-Methode überschrieben. Die `toString()`-Methode von `BPNode` rekonstruiert für die Wurzel eines (Teil-)Baums die durch diesen (Teil-)Baum repräsentierte Formel korrekt, falls die Sprache nur einfache binäre und unäre Operatoren verwendet. Die Methode ist beispielsweise auf LTL-Formeln anwendbar und wird deshalb von `LTLBPNode` nicht überschrieben.

3.1.1.1.3 Kompression von Bäumen `BPNode` definiert eine Methode zur Kompression von Bäumen. Diese Methode ersetzt im Baum alle Teilbäume, die syntaktisch gleiche Teilformeln repräsentieren, durch genau einen dieser Teilbäume. Alle Referenzen auf einen ersetzten Teilbaum werden auf den verbleibenden Teilbaum dieser Art umgesetzt. Das Ergebnis der Kompression ist im Allgemeinen nur noch ein gerichteter azyklischer Graph. Die Methode behandelt zunächst die Blätter des Originalbaumes, wobei sie sich an den Tokens orientiert. Im Anschluss werden alle Teilbäume der Tiefe 1 behandelt, dann alle der Tiefe 2 usw., wobei zwei Teilbäume der Tiefe i syntaktisch äquivalente Teilformeln repräsentieren, wenn ihre Tokens übereinstimmen, beide Wurzeln als linken Nachfolger dasselbe Objekt referenzieren und beide Wurzeln als rechten Nachfolger dasselbe Objekt referenzieren. Der Zweck der Kompression besteht weniger in der Reduktion des benötigten Speicherplatzes als in der Möglichkeit, jede Teilformel durch eine eindeutige Referenz zu identifizieren, unabhängig davon, wie oft sie in der Formel vorkommt.

3.1.1.1.4 Konversion zwischen Bäumen verschiedenen Knotentyps `PNode`-Bäume können über eine statische Methode der Klasse `BPNode` in `BPNode`-Bäume konvertiert werden. Jede direkte oder indirekte Unterklasse von `BPNode` sollte über eine statische Methode zur Konversion von `BPNode`-Bäumen in den jeweiligen Untertyp verfügen. Da auch `BPNode` über eine solche Methode besitzt, kann innerhalb der `BPNode`-Klassenhierarchie jeder Baum aus Objekten eines Typs mit genau einer Konversion in einen Baum eines beliebigen anderen Typs konvertiert werden. Dabei können natürlich Informationen verloren gehen, und nicht jede Konvertierung ist sinnvoll.

3.1.2 Transformation von Formeln der Presburger-Arithmetik in endliche Automaten

Im ersten Semester bestand die Aufgabe der Übersetzergruppe darin, das im Abschnitt 2.3.1 beschriebene Verfahren aufbauend auf unserer Automatenbibliothek und unserem Parser zu implementieren. Es musste also möglich sein, einen durch den Parser für eine Presburger-Formel generierten Ausdrucksbaum entgegenzunehmen und einen `FiniteAutomaton` zu konstruieren, der die Lösungsmenge (erfüllende Belegungen der frei-

en Variablen) der Formel repräsentiert. Im zweiten Semester sind nur noch Anpassungen an die weiterentwickelte Automatenbibliothek, Korrekturen und kleinere Verbesserungen durchgeführt worden.

Während sich die im besagten Abschnitt und in der dort benutzten Quelle ausgebreitete Theorie vorwiegend auf die Konstruktion von Automaten für Atome konzentriert, mussten bei der Implementierung weitere Details ausgearbeitet werden. Beispiele hierfür sind die Normalisierung von Formeln und Atomen und der Umgang mit unerwünschten Formeln. Die Konstruktion von Automaten für Boolesche Verknüpfungen weicht aus Gründen einer einfacheren Implementierung deutlich von der theoretischen Vorlage ab.

Insgesamt ist eine einfach einzubindende Implementierung entstanden, die über eine kleine Schnittstelle verfügt. Da Berechnungen einige Zeit in Anspruch nehmen können, bietet die Schnittstelle ein Listener-Konzept, das es Objekten erlaubt, sich während des Automaten-Konstruktionsprozesses über den Fortschritt der Berechnungen informieren zu lassen. Überdies wurden zur angemessenen Reaktion auf Fehlerzustände spezielle Exceptions entworfen.

3.1.2.1 Statische Struktur

Der Code befindet sich im Paket `aaa.translation.presburger`. Bevor der Transformationsprozess beschrieben wird, sollen zunächst die Bestandteile dieses Pakets erläutert werden.

3.1.2.1.1 `PresburgerToDFATranslator` Dies ist die die Transformation steuernde zentrale Klasse. Eine Instanz kann eine beliebige Zahl von Formeln der Presburger-Arithmetik nacheinander in endliche Automaten übersetzen. Ein im Konstruktor übergebenes Flag bestimmt, ob bei jedem Konstruktionsprozess auch alle Automaten gespeichert werden, die nur als Zwischenergebnisse entstehen. Durch einen Aufruf der Methode `computeResult(PNode)` wird der Konstruktionsprozess angestoßen, wobei als Parameter die Wurzel des Ausdrucksbaumes der Presburger-Formel erwartet wird. Der Rückgabewert der Methode ist der Ergebnisautomat. Die Eingabe, die normalisierte Eingabe und der Ergebnisautomaten des letzten Aufrufs sind über Methoden zugänglich. Gleiches gilt für das Array aller relevanten Variablen, die Liste aller normalisierten Atome der Normalform und die vom Ergebnisautomaten benutzte Variablenordnung. Eine Variablenordnung ist ein Array, das für jede Bitposition i des `AssignmentAlphabet`s der Bitvektorlänge n in der $(n-i-1)$-ten Komponente den Namen der Variablen bereithält, auf deren Wert sich die Position i bezieht. Die Umkehrung der Indizes hängt damit zusammen, dass die Bitvektor-Darstellung als 0-1-String wie üblich absteigend von links

nach rechts durchnummeriert ist $(n-1,\ldots,0)$. Desweiteren ist die Registrierung und Deregistrierung von Listenern möglich.

3.1.2.1.2 `PresburgerAtom` `PresburgerAtom` berechnet für einen im Konstruktor übergebenen Ausdrucksbaum eines Atoms eine Normalform für das Atom. Variablenhaltige Terme werden auf die linke Seite gebracht, und rechts steht nur ein konstanter Term. Die linke Seite wird durch ein `int`-Array der Koeffizienten und ein `String`-Array der zugehörigen Variablen, die rechte Seite durch einen `int` repräsentiert. Diese Daten können nach der Erzeugung des `PresburgerAtom`s mit den passenden Methoden abgerufen werden. Die Methode `isEquation()` liefert genau dann `true` zurück, wenn das Atom eine Gleichung ist. Bei Ungleichungen ist das zwischen der linken und rechten Seite stehende Relationszeichen implizit $\leq$. Die `toString()`-Methode gibt die Normalform als `String` zurück, wobei die Terme immer durch + verbunden sind. Im Unterschied dazu wird im von `toBeautifulString()` gelieferten `String` vor negativen Koeffizienten ein binäres − eingesetzt. Die Atom-Normalisierung wurde in die Klasse `PresburgerAtom` ausgelagert, weil sie naturgemäß ein recht eigenständiger Schritt ist, der bis auf den Teilbaum keine weiteren Informationen benötigt.

3.1.2.1.3 `PresburgerBPNode` Intern arbeitet `PresburgerToDFATranslator` nur mit aus `PresburgerBPNode`-Instanzen zusammengesetzten Ausdrucksbäumen. `PresburgerBPNode` ist eine Unterklasse von `BPNode` und erbt die in Abschnitt 3.1.1 aufgezählten grundsätzlichen Vorteile der Oberklasse. Zusätzlich gibt es zwei Attribute zur Referenzierung von endlichen Automaten und `PresburgerAtom`en. Die spezialisierte `toString()`-Methode stellt Quantoren korrekt dar und verwendet für Knoten, die die Wurzel eines Atoms sind und ein `PresburgerAtom` referenzieren, dessen Methode `toBeautifulString()`. `PresburgerBPNode` enthält Methoden wie `isEx()` und `isNot()`, um den Typ ohne Angabe von Literalen ermitteln zu können. Außerdem gibt es Methoden wie `makeEx()` und `makeNot()`, die die Festlegung des Typs ohne Angabe von Literalen ermöglichen. Bei Syntaxänderungen der Presburger-Formeln muss also nur `PresburgerBPNode` angepasst werden. Werden auch Zwischenergebnisse gespeichert, so wird jeder Zwischenautomat in dem Knoten der Normalform gespeichert, für den er berechnet worden ist. Die Normalform enthält in den Prädikatsknoten stets `PresburgerAtom`e. Die Variablenordnung der Automaten ist direkt aus deren `AssignmentAlphabet`en ersichtlich.

3.1.2.1.4 PresburgerToDFATranslatorListener Wenn sich ein Objekt bei `PresburgerToDFATranslator` als Listener registrieren möchte, muss seine Klasse das Interface `PresburgerToDFATranslatorListener` implementieren. Der Translator in-

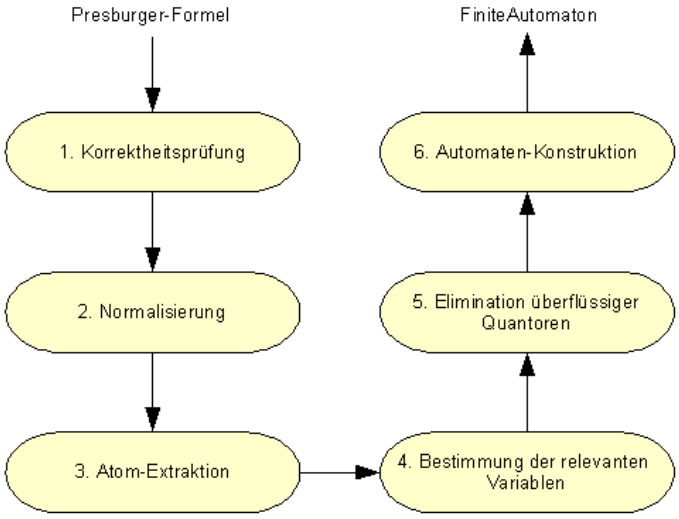


Abbildung 3.1: Transformation einer Presburger-Formel in einen Automaten

formiert alle registrierten Listener nach der erfolgreichen Korrektheitsprüfung einer Eingabe, nach dem erfolgreichen Durchführen weiterer Vorarbeiten (Schritte zwei bis vier im Prozess-Diagramm) und jeweils zu Beginn und nach Beendigung der Berechnung von Zwischenergebnissen. Während der Berechnung kann also stets verfolgt werden, an welcher Stelle sich die Bottom-Up-Konstruktion gerade befindet. Auf Statusmeldungen während der Konstruktion von Automaten für Atome wurde verzichtet.

3.1.2.2 Transformationsprozess

Der Prozess der Transformation eines Presburger-Ausdrucksbaums in einen endlichen deterministischen Automaten ist in die sechs in Abbildung 3.1 dargestellten Aktivitäten einteilbar.

3.1.2.2.1 Korrektheitsprüfung `PresburgerToDFATranslator` setzt voraus, dass die Eingabe eine bezüglich der im Abschnitt „Erfüllbarkeitsprüfung von Formeln der Presburger-Arithmetik“ angegebenen Grammatik syntaktisch korrekte Formel repräsentiert. Wurde der Syntaxbaum vom Parser erstellt, so ist die Voraussetzung erfüllt. Hier werden aber noch zwei weitere, vom Parser nicht berücksichtigte Korrektheitsaspekte geprüft. Erstens werden Formeln wie $\exists x_1 : (((3 * x1 = x3) \wedge \forall x1 : ((x1 + 2 * x2 = x3))))$, in denen Quantoren mit identischer Variable ineinander geschachtelt sind, als inkorrekt angesehen. Zweitens werden auch Formeln wie $((x1 - x2 \leq 7) \vee \exists x2 : (x1 + x2 = x3))$, in denen ein und dieselbe Variable sowohl frei als auch gebunden vorkommt, als nicht korrekt erachtet. Beide Fälle könnten (analog zur Variablenüberdeckung in einigen Pro-

grammiersprachen) sinnvoll interpretiert und durch Variablenumbenennungen aufgelöst werden. Dies tut das Programm aber nicht, weil der Programmierer beide Fälle als sehr ungewöhnlich ansieht und darin eher ein Versehen des Anwenders vermutet. Algorithmisch wird die Korrektheitsprüfung mit einem rekursiven Durchlauf des Syntaxbaums gelöst. Dabei werden als Parameter der aktuelle Knoten, eine `HashMap` der bisher gefundenen frei auftretenden Variablen, eine `HashMap` der bisher gefundenen Quantor-gebundenen Variablen und eine `HashMap` der auf dem Pfad von der Wurzel zum aktuellen Knoten liegenden Quantor-gebundenen Variablen verwendet. `HashSet`s von Variablen genügen dem Algorithmus, doch die `HashMap`s bilden Variablen auf Knoten ab, so dass in den Exceptions auch Positionsangaben untergebracht werden können. Die Parameter erlauben die Erkennung sämtlicher Fehlerfälle. Erreicht man beispielsweise einen Quantorknoten, dessen Variable bereits auf dem aktuellen Pfad gebunden ist, so sind zwei Quantoren mit derselben Variable ineinander geschachtelt, und der erste Fehlerfall liegt vor.

3.1.2.2.2 Normalisierung In diesem Schritt wird die Formel in Pränexnormalform überführt. Diese hat den Vorteil, dass für die Booleschen Verknüpfungen auf sehr einfache Weise Automaten berechnet werden können. Alle Atom-Automaten arbeiten auf demselben Alphabet, und im inneren Teil der Formel werden diese Atome lediglich durch $\wedge$ und $\vee$ verknüpft, was den Automaten-Operationen $\cap$ und $\cup$ entspricht. Jeder Quantor erfordert eine Projektion des Automaten, wobei sich das Alphabet verändert. Eine Boolesche Verknüpfung oberhalb eines Quantors könnte im Allgemeinen nicht mehr durch eine Standard-Automaten-Operation behandelt werden, weil die beiden Kind-Automaten unterschiedliche Alphabete verwenden. Ein Standard-Produktautomat wäre in diesem Fall aber naiv, weil die Kinder gemeinsame Variablen enthalten können, die Automaten also nicht unabhängig voneinander simuliert werden können. Gibt es gemeinsame Variablen, so gibt es inkonsistente Bitvektoren, die Variablen mehrfach belegen. Das Lesen solcher Vektoren muss sofort in den endgültig ablehnenden Zustand führen. Eine kompaktere Produkt-Automaten-Operation müsste eigens für diese Anwendung unter größerem Aufwand implementiert werden, daher haben wir uns für den hier dargestellten Weg entschieden. Man könnte auch die Alphabete und Automaten nach Bedarf angleichen, so wie es bei `BuchiAutomaton` möglich ist. Die Quantor-Variablen werden zudem so angeordnet, dass ein besonders einfach zu implementierender und effizient zu berechnender Spezialfall der Projektion ausreicht. Der erste Schritt der Normalisierung ist die Bereinigung der Formel. Dabei werden Variablen, die an mehrere nicht ineinander geschachtelte Qantoren gebunden sind (dies ist erlaubt), umbenannt. Beispielsweise wird $((\exists x : F_1) \vee (\forall x : F_2))$ in $((\exists x\_subst1 : F_1) \vee (\forall x\_subst2 : F_2))$ transformiert. Im zweiten Schritt werden alle Negationen unter Berücksichtigung der De Morgan'schen Gesetze und des Involutionsgesetzes (doppelte Negationen heben sich auf) nach innen delegiert. Ungleichungen können auf ein-

fache Weise negiert werden, so dass der Negations-Operator verschwindet. Über Gleichungen darf ein Negations-Operator stehen bleiben, denn die Gleichung müsste durch die Disjunktion zweier Ungleichungen ersetzt werden, von denen jede einzelne ähnlich aufwändig zu berechnen ist wie die Gleichung. Andererseits ist der für die Gleichung berechnete Automat deterministisch und das Komplement in Linearzeit zu berechnen. Im dritten Schritt werden die Quantoren nach oben gezogen, aus $((\exists x\_subst1 : F_1) \vee (\forall x\_subst2 : F_2))$ wird $\exists x\_subst1 : (\forall x\_subst2 : (F_1 \vee F_2))$.

3.1.2.2.3 Atom-Extraktion Nach der Normalisierung steht die Menge der Atome fest, die nun bestimmt wird. Dazu wird der Syntaxbaum rekursiv durchlaufen und für alle gefundenen Knoten, die mit einem Prädikat markiert sind, ein `PresburgerAtom` mit dem Knoten als Parameter erzeugt. Das `PresburgerAtom` wird im Knoten gespeichert. Alle gefundenen Atome werden in eine Liste aufgenommen. Die für das Atom berechnete Normalform entspricht der im Abschnitt „Erfüllbarkeitsprüfung von Formeln der Presburger-Arithmetik“ definierten. Hinzu kommt lediglich, dass die Variablen lexikographisch sortiert sind und alle Koeffizienten von Null verschieden sind. Das `PresburgerAtom` durchläuft jeden der beiden Term-Bäume und aktualisiert die Variablenkoeffizienten in einer HashMap, wann immer er auf die Variable stößt (sie kann ja mehrfach in der Summe vorkommen). Dabei wird über einen Vorzeichenparameter auch das binäre $-$ korrekt verarbeitet. Anschließend werden die beiden Terme zur linken Seite verrechnet, aber die konstanten Summanden zur rechten Seite verrechnet.

3.1.2.2.4 Bestimmung der relevanten Variablen Der normalisierte Syntaxbaum kann irrelevante Variablen enthalten. Dabei handelt es sich zum einen um Quantor-Variablen, die im Gültigkeitsbereich des Quantors gar nicht auftreten. Zum anderen sind solche Variablen irrelevant, die zwar in unnormalisierten Atomen vorkommen, jedoch in keiner der durch die `PresburgerAtom`e dargestellten Normalformen. So ist in $(((7{*}x{-}8{*}y{+}z = 7{*}x{+}2{*}y{-}16){\vee}(4{*}x{-}8{*}y{-}3{*}x = x{+}2{*}z)){\wedge}(30{*}y{+}60{*}z \leq 0))$ die Variable x irrelevant. Die Menge der relevanten Variablen ergibt sich als die Vereinigung der Variablenmengen der einzelnen Atom-Normalformen. Die so erhaltenen Variablen werden im `String`-Array `allVariables` gespeichert. Das Array ist gemäß der folgenden Ordnung sortiert:

- Freie Variablen sind untereinander lexikographisch geordnet.
- Quantor-Variablen sind nach zunehmender Tiefe im Syntaxbaum geordnet.
- Jede freie Variable ist kleiner als jede Quantor-Variable.

Am Anfang des Arrays stehen also lexikographisch sortiert alle freien Variablen. Mit diesem Präfix des Arrays wird das die Variablenordnung darstellende Array gefüllt. Den letzten Teil bilden die Quantor-Variablen; je größer der Index, desto größer ist die Tiefe, in der sich der zugehörige Quantor befindet. Geschlossene Presburger-Formeln enthalten keine freien Variablen, und die Variablenordnung hat die Länge 0. Damit die Alphabetskonstruktion nicht scheitert, wird in diesem Fall für die geschlossene Formel Φ die Normalform von $\Phi \wedge (dummy \leq 0 \vee dummy \geq 0)$ verwendet. Sowohl die Zahl quantifizierter als auch die Zahl freier Variablen ist nach oben unbeschränkt.

3.1.2.2.5 Elimination überflüssiger Quantoren In diesem Schritt werden alle Quantoren Qx : entfernt, für die x eine irrelevante Variable ist. Die Semantik der Formel wird dadurch nicht verändert.

3.1.2.2.6 Automaten-Konstruktion Hierbei findet ein rekursiver Post-Order-Durchlauf des normalisierten Syntaxbaums statt, das Verfahren geht also wie in der Theorie beschrieben nach dem Bottom-Up-Ansatz vor. Auf der untersten Ebene werden für die `PresburgerAtom`e Automaten berechnet. Diese Atom-Automaten arbeiten auf einem Alphabet mit Bitvektoren der Länge von `allVariables`. Jede Bitposition entspricht genau einer Arrayposition. Genauer gesagt ist `allVariables` die Variablenordnung jedes einzelnen Atom-Automaten. Das zugrundeliegende `PresburgerAtom` muss man sich also auf sämtliche relevante Variablen expandiert vorstellen, wobei nicht vorkommende Variablen den Koeffizienten 0 erhalten. Alle Atom-Automaten benutzen dasselbe Alphabet.

Nachdem alle Atome und Booleschen Verknüpfungen behandelt worden sind, werden die Quantor-Automaten berechnet. Der innerste Quantor wird zuerst bearbeitet. Aufgrund der gewählten Variablenordnung steht die zugehörige Variable in der letzten Komponente von `allVariables`. Der zugehörige Bitindex ist 0. Diese Bitposition muss wegprojiziert werden, was besonders einfach ist. Es müssen lediglich sämtliche Symbolintervall-Grenzen des Automaten durch 2 dividiert werden. Aufeinanderfolgende Intervalle verschmelzen genau dann zu einem Intervall, wenn zwischen ihnen nur ein Symbol lag. Beim nächsten Quantor bezieht sich wieder die letzte Bitposition auf die durch diesen Quantor gebundene Variable, dies ist bei allen Quantoren so.

Direkt im Anschluss an die Projektion muss noch sichergestellt werden, dass alle Repräsentationen einer Lösung akzeptiert werden. Der Automat für $(x = 1 \wedge y = 4)$ akzeptiert alle Wörter der Sprache $(0,0)(0,0)^*(0,1)(0,0)(1,0)$, was auch korrekt ist. Durch Wegprojizieren der Variablen y ergibt sich ein Automat für $\exists y : ((x = 1 \wedge y = 4))$, der allerdings nur 00^*001 statt 00^*1 akzeptiert. Nach der Projektion ist lediglich sichergestellt, dass Repräsentationen ab einer bestimmten Länge akzeptiert werden. Die gewünschte Ei-

genschaft wird wiederhergestellt, indem eine Transition vom Startzustand zum Zustand q mit Beschriftung b eingefügt wird, wenn q über ein Wort in b^+ erreichbar ist. Algorithmisch bildet der Startzustand Schicht 0, anschließend werden der Reihe nach die Schichten $1, 2, \ldots$ berechnet. Schicht i, $i > 0$, enthält genau die Zustände, die von Schicht $i-1$ aus in genau einem Schritt erreichbar sind und in keiner der Schichten j mit $j < i$ bereits enthalten waren. Für jeden Zustand q der Schicht i wird die Menge der Vektoren b berechnet, so dass q vom Startzustand aus durch Lesen des Wortes b^i erreichbar ist. Für $i = 1$ ist dies die Vektormenge der eingehenden Transition. Für $i > 1$ ist für jeden Vorgänger q' von q in Schicht $i-1$ die Vektormenge von q' mit der Vektormenge der Transition von q' nach q zu schneiden, und anschließend sind die so erhaltenen Schnitte zu vereinigen. Der Algorithmus terminiert, sobald eine Schicht leer ist. Enthält in einer Schicht ein Zustand in seiner Vektormenge den Nullvektor, so wird der Startzustand akzeptierend, da das leere Wort als Nullvektor interpretiert wird. Vor Ausführung dieses Algorithmus werden aus dem Automaten alle ε-Transitionen entfernt.

Problematisch bei Quantoren ist die Determinisierung des projizierten Automaten, die mit einem allgemeinen Verfahren durchgeführt wird und daher einen exponentiellen Blow-Up zur Folge haben kann.

3.1.2.3 Verbesserungsmöglichkeiten

3.1.2.3.1 Quantoren-Reihenfolge in der Pränex-Normalform Beim Nach-Oben-Ziehen von Quantoren während der Normalisierung kann die Reihenfolge der Quantoren beliebig verändert werden, weil die Operationen kommutativ sind. Es wird vermutet, dass vor allem Quantorenwechsel einen Blow-Up verursachen, daher sollte die Zahl der Quantorenwechsel minimiert werden. Um flexibel experimentieren zu können, sollte die Anordnungsstrategie für den Aufrufer wählbar gemacht werden.

3.1.2.3.2 Auswahl anderer Normalformen Auch völlig andere Normalformen sind denkbar. Man könnte zum Beispiel alle Quantoren nach innen delegieren. Dann benötigt man für die Behandlung von $\wedge$ und $\vee$ allerdings entweder die schon erwähnte kompakte Produkt-Automaten-Konstruktion oder die Alphabetsangleichung. Eine Alphabetsangleichung fällt deutlich leichter, wenn die Transitionen mit OBDDs beschriftet werden. Die OBDDs könnten im einfachsten Fall die Bitvektoren des Alphabets seriell einlesen, was die Automatenbibliothek bereits erlaubt. Die Projektion beliebiger Positionen ist für OBDDs auch recht einfach zu realisieren. Es könnten weitere Normalformen implementiert werden, die einstellbar gemacht werden. Es ist zu vermuten, dass es von der konkreten Formel abhängig ist, welche der implementierten Normalformen am günstigsten ist. Vielleicht kann man dies zu einem gewissen Grad sogar automatisch erkennen.

3.1.2.3.3 Inkrementelles Verändern der Formel Wenn sich eine Formel nur so ändert, dass sich im Syntaxbaum der Normalform lediglich vereinzelte Knoten ändern, so genügt es, bei Speicherung von Zwischenergebnissen von diesen Knoten ausgehend für alle auf dem jeweiligen Pfad zur Wurzel liegenden Knoten neue Automaten zu berechnen, der Rest ist noch immer aktuell. Dies kann sich vor allem bei Verifikations-Anwendungen bezahlt machen, wo sich aufeinander folgende Zusicherungen gemäss eines Kalküls nicht beliebig unterscheiden. Allerdings wird vom Parser ein komplett neuer Syntaxbaum geliefert, für den erst wieder die Normalform berechnet werden muss. Hier muss das System Matchings erkennen, indem es die beiden normalisierten Ausdrucksbäume miteinander vergleicht. Oder aber man führt die Änderungen direkt im normalisierten Ausdrucksbaum durch.

3.1.2.3.4 Veränderung der Automaten-Struktur Die gewählte Automaten-Konstruktion arbeitet unter Umständen mit sehr großen Alphabeten. Man kommt aber stets mit dem Alphabet $\{0, 1\}$ aus, wenn man auch die Bitvektoren des bisherigen Alphabets sequentiell einliest. Jeder Zustand wird dann durch eine Art Entscheidungsdiagramm (OBDD) ersetzt. Dadurch erhöht sich zwar die Zahl der Zustände, aber nicht so stark wie die Zahl der Transitionen sinkt. Die Projektion beliebiger Bits sollte einfacher werden, genauso wie die kompakte Produkt-Automaten-Konstruktion und die Alphabetsangleichung.

3.1.2.3.5 Parallelisierung von Berechnungen Das Verfahren ist sehr rechenintensiv, dürfte aber auch gut parallelisierbar sein. Unabhängige Zweige des Baums können unabhängig voneinander berechnet werden. Eine andere Möglichkeit, bei der allerdings mehrere Threads gleichzeitig auf demselben Automaten arbeiten, ist, das Alphabet (bei Anwendung des herkömmlichen Verfahrens) auf die Threads aufzuteilen. Der erste Thread ist zum Beispiel nur für den Teil $\{000000, \ldots, 011111\}$ zuständig, während sich ein zweiter Thread um $\{100000, \ldots, 111111\}$ kümmert. Es ist zu überlegen, ob das Verfahren nicht gleich als verteiltes System auszulegen ist. Aber auch beim Einsatz auf einzelnen PCs wird Parallelisierung angesichts der auf dem Vormarsch befindlichen Multi-Core-CPUs immer wichtiger.

3.1.3 Model Checking von Kripkemodellen mit LTL-Formeln

Eines der Teilziele der PG im zweiten Semester bestand darin, dem Triple-A-Framework Features hinzuzufügen, die das Model Checking von Kripkemodellen mit Formeln der Linear Temporal Logic (LTL) auf möglichst effiziente und benutzerfreundliche Weise

ermöglichen. Der von uns verfolgte automatenbasierte Lösungsansatz sieht vor, dass sowohl für das Kripkemodell als auch für die Formel jeweils ein Büchiautomat berechnet wird. Als Alphabete verwenden diese Automaten Potenzmengen von Mengen atomarer Propositionen. Wörter sind hier also keine Pfade in Kripkemodellen, sondern Folgen von Zustandslabels, die auf den Pfaden der Reihe nach auftreten. Ein in einem Kripkemodell existierender Pfad $q_1, q_2, \ldots$ (die q_i sind Zustände) entspricht also in einem Büchiautomaten dem Wort $L(q_1), L(q_2), \ldots$ (Label $L(q)$ ist Menge der in q gültigen atomaren Propositionen).

Der für das Kripkemodell konstruierte Büchiautomat repräsentiert auf diese Weise die Menge der im Kripkemodell möglichen Pfade. Die Graphstruktur des Büchiautomaten und des Modells unterscheiden sich nicht. Einfach ausgedrückt ergibt sich der Büchiautomat aus dem Kripkemodell, indem für jeden Zustand q das Label $L(q)$ an jede ausgehende Kante von q kopiert wird und dann aus q gelöscht wird. Desweiteren werden alle Zustände akzeptierend. Offensichtlich enthält der Ergebnisautomat exakt dieselben Informationen wie das Kripkemodell. Für Details sei auf Abschnitt 3.2.6 verwiesen.

Die Konstruktion eines Büchiautomaten für eine LTL-Formel gestaltet sich algorithmisch und komplexitätstheoretisch schwieriger. Die Zahl der Zustände des Büchiautomaten ist im worst-case exponentiell in der Länge der Formel. Der Automat repräsentiert genau diejenigen Pfade, die die Formel erfüllen, wobei er sich auf kein spezielles Kripkemodell bezieht. Ein und derselbe Automat kann also für das Checking vieler verschiedener Kripkemodelle verwendet werden. Die Implementierung ist eine Realisierung des Algorithmus von Gerth, Peled, Vardi und Wolper, der in [**?**] dargestellt ist. Hier liegt der Schwerpunkt auf den Eigenheiten der Implementierung.

3.1.3.1 Das Package `aaa.translation.ltl`

Der Code ist im Package `aaa.translation.ltl` zu finden. Die Klasse `LTLtoBuchiTranslator` kann verwendet werden, um LTL-Formeln in Büchiautomaten zu übersetzen. `IAutomaton` und `IAutomatonState` sind von `LTLtoBuchiTranslator` benutzte, nach außen nicht sichtbare Hilfsklassen. `LTLBPNode` stellt Formelbäume (Ausdrucksbäume) von LTL-Formeln dar. Ein weiterer Bestandteil des Package ist `KripkeLTLModelChecker`, ein Model Checker für das Prüfen von Kripkemodellen mit Formeln der LTL, der von `LTLtoBuchiTranslator` Gebrauch macht. Im Zusammenhang mit `KripkeLTLModelChecker` spielt das Interface `LTLPreprocessor` eine wichtige Rolle. Zur Zeit wird dieses Interface nur von `WhileLTLPreprocessor` implementiert. Die Präprozessoren haben die Aufgabe, sinnvolle Formeln, die aber aus technischen Gründen in ihrer Originalform Fehler verursachen würden, unter Erhaltung der Semantik so zu

modifizieren, dass dies nicht mehr geschieht.

3.1.3.2 Die Klasse `KripkeLTLModelChecker`

Eine Instanz dieser Klasse kann genutzt werden, um beliebig viele Modellprüfungen nacheinander durchzuführen. Das geschieht jeweils durch einen Aufruf der Methode `check(KripkeModel, LTLPreprocessor, LTLBPNode)`.

`check(KripkeModel, LTLPreprocessor, LTLBPNode)` lässt zunächst das Kripkemodell in einen Büchiautomaten transformieren. Dann wird die LTL-Formel vom Präprozessor bearbeitet, falls ein solcher übergeben wurde. Der Präprozessor kann zu dem Ergebnis kommen, dass die Formel unsinnig ist und dies mit einer `IllegalArgumentException` signalisieren, was einen Abbruch der Modellprüfung zur Folge hat. Andernfalls sollte der Präprozessor die Formel – falls nötig – in geeigneter Weise modifizieren, so dass die „neue“ Formel nun in einen Büchiautomaten übersetzt werden kann. Dem `LTLtoBuchiTranslator` wird dabei das Alphabet des für das Kripkemodell berechneten Büchiautomaten übergeben, das damit auch das Alphabet des für die Formel berechneten Büchiautomaten wird. Auf diese Weise kann später ein Gegenbeispiel und evtl. der Schnittautomat berechnet werden, ohne einen Alphabetsabgleich durchzuführen.

Der `KripkeLTLModelChecker` verfügt über ein modifizierbares Flag, dass die Interpretation von Formeln startet. Ist das Flag gesetzt, so wird davon ausgegangen, dass die gegebene Formel eine zu erfüllende Eigenschaft des gegebenen Modells spezifiziert. In diesem Fall wird die Formel zusätzlich zum Preprocessing negiert. Das Modell besteht die Prüfung genau dann, wenn der Schnitt der beiden Büchiautomaten leer ist. Dieser Interpretations-Modus ist voreingestellt. Ist das Flag dagegen nicht gesetzt, so geht der Model Checker davon aus, dass das Kripkemodell die Formel nicht erfüllen soll und unterlässt die Negation der LTL-Formel. Wieder besteht das Modell die Prüfung genau dann, wenn der Schnitt leer ist.

Ob der Schnittautomat tatsächlich konstruiert wird, hängt von einem weiteren Flag ab. Standardmäßig wird er nicht konstruiert, denn in der Regel genügt dem Anwender ein einziges Gegenbeispiel. `BuchiAutomaton` bietet eine Methode `findCommonWord(BuchiAutomaton)` an, die die beiden Büchiautomaten parallel durchläuft und ein gemeinsames Wort zurückgibt, sobald sie eines findet. Eine ähnliche Methode liefert den passenden Pfad im Automaten, für den die Methode aufgerufen worden ist. Diese Methode wird vom Model Checker auf dem Automaten aufgerufen, der für das Kripkemodell berechnet wurde, um ein Gegenbeispiel zu erhalten. Wegen der oben angesprochenen strukturellen Gleichheit von Modell und Automat existiert der so erhaltene Pfad auch im Modell und ist damit aussagekräftig.

Es ist als realistisch einzuschätzen, dass der Benutzer seine Anforderungen an ein Modell als eine Menge von wenigen bis vielen Formeln formuliert und jede Anforderung einzeln überprüfen lässt, anstatt alle Formeln durch Boolesche Operatoren miteinander zu verbinden. So erhält er detailliertere Informationen. Hier kann die `check(KripkeModel, LTLPreprocessor, LTLBPNode)`-Methode ohne Bedenken aufgerufen werden, denn wenn sich die Referenz auf das Kripkemodell nicht von derjenigen aus dem letzten Aufruf unterscheidet, wird das Kripkemodell nicht erneut in einen Büchiautomaten transformiert. Analog gilt für identische Referenzen auf Ausdrucksbäume, dass bei unveränderter Präprozessor-Referenz die Transformation der Formel in einen Automaten nicht erneut angestoßen wird. Bei unterschiedlichen Präprozessor-Referenzen ist eine neue Transformation erforderlich, wenn die aktuelle Referenz nicht `null` ist, andernfalls kann auf die im Model Checker gespeicherte Originalformel zurückgegriffen werden. Mit `KripkeLTLModelChecker` kann man also auch verschiedene Modelle mit einer Formel effizient prüfen. Dieses Verhalten des Model Checkers führt allerdings zu falschen Ergebnissen, wenn die referenzierten Objekte nachträglich verändert werden. In diesem Fall sollte vor jedem Checking der Zustand des Model Checkers durch einen `clear()`-Aufruf zurückgesetzt werden.

3.1.3.3 Präprozessoren für LTL-Formeln

Die Präprozessoren wurden ursprünglich durch Eigenheiten des Model Checkings von Kripkemodellen motiviert, die aus Whileprogrammen erzeugt wurden. Es ist aber denkbar, dass ein Preprocessing auch noch bei anderen Modellen erforderlich wird, weshalb die Präprozessoren als allgemeines Konzept eingeführt wurden.

Unsere While-Sprache arbeitet mit ganzzahligen Variablen, wobei für jede Variable `x` der individuelle Wertebereich in Form des kleinsten möglichen Wertes und des größten möglichen Wertes angegeben werden muss. Jede Variable kann immer auch einen der Werte `underflow`, `overflow` und `uninit` annehmen. Das für ein Whileprogramm generierte Kripkemodell macht Aussagen über mögliche Zustandsübergänge während der Programmausführung. Um die Zustandskomponenten, d. h. die Variablen, einzeln beschreiben zu können, enthält das Kripkemodell atomare Propositionen des Typs `variable=wert`. Dies ist ein sehr verständliches Format, und der Benutzer kann solche Propositionen in seinen LTL-Formeln direkt verwenden.

Das Kripkemodell enthält nur die Propositionen in seinem Alphabet, die es wirklich für seine Modellierungszwecke benötigt. Die Proposition `x=3` existiert also nur dann, wenn es einen Zustand gibt, in dem `x=3` gilt. Für jede Variable `var` mit Wertebereich $\{L_{var}, \ldots, U_{var}\}$ sämtliche Propositionen aus $\{var = wert \mid L_{var} \leq wert \leq U_{var}\}$ dem Al-

phabet hinzuzufügen, kann beträchtlichen Overhead verursachen, da beispielsweise Variablen vorsichtshalber wie 32-Bit Integer deklariert sein könnten, obwohl sie tatsächlich nur 152 verschiedene Werte annehmen. Andererseits kann der modelcheckende Benutzer nicht unbedingt überschauen, welche Werte angenommen werden und welche nicht. Auch so etwas soll ja gerade durch das Model Checking erkundbar sein. Wenn eine LTL-Formel aber beispielsweise die Proposition `x=3` enthält, `x` den Wertebereich $\{0, \ldots, 22\}$ hat, aber nie den Wert 3 annimmt, so bricht die Transformation der Formel mit einer Exception ab, weil ihr Büchiautomat das Alphabet des Kripkemodells (identisch mit dem Alphabet des für das Kripkemodell berechneten Büchiautomaten) verwendet. Die Transitionen lassen sich nicht mit Mengen beschriften, die `x=3` oder `!x=3` enthalten.

Dieses Problem löst `WhileLTLPreprocessor`. Er bekommt das `AssignmentAlphabet` des Kripkemodells oder Büchiautomaten und ein `Environment` für das Programm im Konstruktor übergeben. Das `Environment` enthält alle Variablen, deren Wertebereiche und deren aktuellen Wert. Die aktuellen Werte sind nur während der Transformation des Whileprogramms in ein Kripkemodell von Interesse und haben hier keine Bedeutung. Übergibt der `KripkeLTLModelChecker` eine Formel an die `preprocess`-Methode des `WhileLTLPreprocessor`s, so werden alle in der Formel enthaltenen atomaren Propositionen auf ihre Sinnhaftigkeit überprüft. Eine atomare Proposition ist genau dann sinnvoll, wenn sie das Format `var=value` hat, wobei `var` ein dem `Environment` bekannter Variablenname ist und `value` entweder die dezimale Repräsentation einer Zahl aus dem Wertebereich von `var` ist oder mit einem der Werte `underflow`, `overflow` oder `uninit` identisch ist. Enthält die Formel eine nicht sinnvolle Proposition, so wird das Preprocessing mit einer `IllegalArgumentException` abgebrochen. Sinnvolle Propositionen, die im Alphabet enthalten sind, erfordern keine weiteren Aktivitäten. Dem Alphabet unbekannte sinnvolle Propositionen sind dem Alphabet nur deshalb unbekannt, weil sie nie erfüllt sind. Darum werden solche Propositionen in der Formel korrekterweise durch `false` ersetzt.

Es wäre auch denkbar, im Kripkemodell nicht Propositionen des oben beschriebenen Typs zu verwenden, sondern jedes einzelne Bit des Offsets relativ zur unteren Schranke zu beschreiben. Für die Variable `x` mit Wertebereich $\{-4, \ldots, 7\}$ beispielsweise werden dann die Propositionen `x0`, `x1`, `x2` und `x3` erzeugt. Die Entsprechung für `x=3` wäre dann `!x3 && x2 && x1 && x0`. Bei dieser Lösung ist die Zahl der für eine Variable erzeugten Propositionen logarithmisch in der Länge des Wertebereiches. Dagegen beträgt sie bei der ersten Lösung im best-case 1, im worst-case ist sie linear in der Länge des Wertebereiches. Bei der zweiten Lösung gibt es das Problem nicht konstruierter Propositionen genau genommen nicht, jedoch sollte der Benutzer in seinen Formeln weiterhin `x=3` schreiben dürfen. Dafür könnte aber ein anderer Präprozessor geschrieben werden, der jede menschenlesbare Proposition durch die äquivalente Konjunktion ersetzt.

3.1.4 Transformation von LTL-Formeln in Büchiautomaten

Eine Instanz von `LTLtoBuchiTranslator` ist in der Lage, beliebig viele LTL-Formeln nacheinander in Büchiautomaten zu transformieren. Bezüglich des Algorithmus sei auf [**?**] verwiesen.

3.1.4.1 Alphabete

Gemeinsam mit der Formel kann ein `AssignmentAlphabet` übergeben werden, das der Büchiautomat verwenden soll. Auf diese Weise ist das Alphabet des Automaten nicht auf die in der Formel vorhandenen atomaren Propositionen beschränkt. Wird kein Alphabet übergeben, so wird aus den Propositionen der Formel ein Alphabet konstruiert. Falls die Formel keine Propositionen enthält, bekommt das Alphabet dennoch die Proposition `p`, da propositionsfreie `AssignmentAlphabet`e nicht erlaubt sind.

3.1.4.2 Normalisierung

Die Formel wird in die in Abschnitt [**?**] definierte Normalform überführt. Es kann optional eine anschließende Vereinfachungsstufe eingeschaltet werden, die z. B. `teilformel && true` zu `teilformel` oder `teilformel U true` zu `true` vereinfacht. Konstanten pflanzen sich also rekursiv in Richtung Wurzel so weit wie möglich fort. Standardmäßig ist die Vereinfachung abgeschaltet, `KripkeLTLModelChecker` schaltet sie aber ein. Insbesondere bei Verwendung von `WhileLTLPreprocessor` können Formeln entstehen, die sich noch vereinfachen lassen.

3.1.4.3 Zwischenautomaten

Es wird nicht direkt ein Büchiautomat konstruiert, sondern zunächst ein `IAutomaton` (Intermediate Automaton). `IAutomaton` und die zugehörige Zustandsklasse `IAutomatonState` sind speziell auf die Anforderungen der Transformation zugeschnitten. Jeder Zustand verwaltet jeweils eine `HashSet<IAutomatonState>` der Zustände, von denen Transitionen eingehen, eine `LinkedList<LTLBPNode>` von „neuen“ Formeln und je eine `HashSet<LTLBPNode>` zur Verwaltung der „alten”Formeln und der Formeln, die im nächsten Zustand gelten müssen. Es wurden noch zwei zusätzliche Instanzen von `HashSet<LTLBPNode>` eingeführt. Die eine enthält genau die atomaren Propositionen unter den „alten“ Formeln, die andere verwaltet genau diejenigen atomaren Propositionen, deren Negation zu den „alten“ Formeln gehört. Bei der Berechnung der Trigger sind die relevanten Elemente dann unmittelbar verfügbar. `IAutomaton` verwaltet die Menge aller

Zustände als `HashMap<IAutomatonState, IAutomatonState>`, wobei jeder Zustand auf sich selbst abgebildet wird. Die `contains`-Methode von `HashSet<IAutomatonState>` liefert das enthaltene, dem Argument gleichende Element nicht zurück. Das enthaltene Element ist aber von Interesse, wenn nach einem vorhandenen Zustand gesucht wird, der jeweils dieselbe Menge „alter“ Formeln und „nächster“ Formeln aufweist wie ein neuer, noch nicht eingefügter Zustand. Die `equals`- und die `hashCode`-Methoden von `IAutomatonState` sind passenderweise so überschrieben, dass der Hashcode nur von den „alten“ und „nächsten“ Formeln abhängt, und zwei Zustände genau dann gleich sind, wenn die besagten Formelmengen übereinstimmen. Zum Abschluss lässt der Zwischenautomat den Büchiautomaten degeneralisieren.

3.1.4.4 Repräsentation von Teilformelmengen

Wie die Parametrisierung der Mengen und Listen schon vermuten lässt, repräsentiert der Algorithmus eine Teilformel direkt durch die Wurzel ihres `LTLBPNode`-Baums. So kann leicht auf ihre Unterformeln zugegriffen werden. Da `HashSet`s miteinander verglichen werden, sollte der Test auf syntaktische Gleichheit zweier Teilformeln effizient sein. Stellen mehrere Bäume syntaktisch äquivalente Teilformeln dar, so kann der Gleichheitstest nur dadurch erfolgen, dass beide Bäume komplett durchlaufen werden, oder eine Knotenmarkierung (z. B. `String`-Repräsentation des Teilbaumes oder eindeutiger `int` für jede vorkommende Teilformel) verwendet wird. Stattdessen sieht der hier gewählte Weg vor, dass der Baum der Normalform komprimiert wird, bevor ein Zwischenautomat konstruiert wird. Die Kompression ergibt sich dadurch, dass im Normalform-Baum alle Teilbäume, die syntaktisch gleiche Teilformeln darstellen, durch einen dieser Teilbäume ersetzt werden. Alle Referenzen auf einen ersetzten Teilbaum werden auf den verbleibenden Teilbaum dieser Art umgesetzt. Das Ergebnis der Kompression ist im Allgemeinen nur noch ein gerichteter azyklischer Graph. Dieser stellt jedoch sicher, dass zwei unterschiedliche Knoten auch zwei syntaktisch verschiedene Teilformeln repräsentieren. Ein Gleichheitstest reduziert sich nun auf einen sehr effizienten Referenzenvergleich, den die `equals`-Methode von `LTLBPNode` durchführt.

3.1.5 Transformation von Whileprogrammen in Kripkemodelle

3.1.5.1 Motivation

Das While-Transformationsprojekt verwirklicht den Schritt, neben kleinen, händisch erstellten Kripke-Strukturen, auch größere und problemspezifische Strukturen automatisch erzeugen zu können.

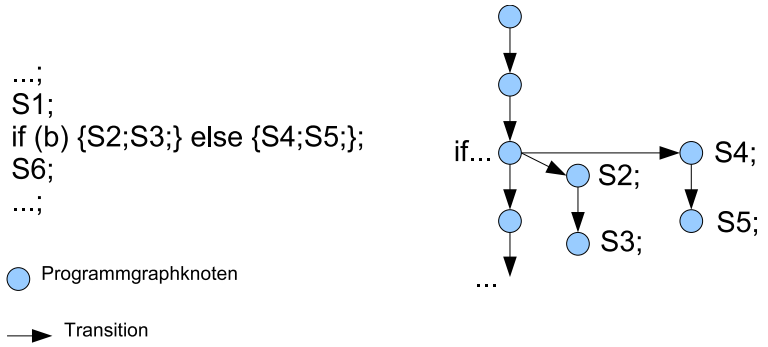


Abbildung 3.2: If zu Programmgraph

Als Ausgangspunkt dient dazu die “While“-Sprache, eine Untersprache von Java (siehe [**?**]).

3.1.5.1.1 Anforderungen Aufgrund des zeitlich eng abgesteckten Rahmens, der zur Umsetzung zur Verfügung stand, war die Mindestanforderung an das Projekt, die “While“-Sprache so zu verarbeiten, dass ein verwertbares Modell erzeugt wird.

Es wurden ebenfalls Versuche unternommen, die While-Sprache hinsichtlich der Modellbildung zu erweitern.

Der Autor möchte jedoch, dass diese Realisierung als ein Prototyp oder Experiment verstanden wird, das gute verwertbare Ansätze besitzt und u. U. verstecktes Potential, jedoch die Sprache an sich Schwächen hinsichtlich der Modellierung aufweist. Siehe dazu 3.1.5.1.8.

3.1.5.1.2 Programmgraphen Programmgraphen sind allgemeine Graphen, die den Fluss eines Programms modellieren.

Programmgraphen in Kontext dieser Analyse seien wie folgt (informell) definiert:

Programmgraph

Jeder Knoten eines Programmgraphen hat beliebig viele Nachfolger und ist mit einer Anweisung x aus der Sprache “While“ beschriftet. Einer der Nachfolger ist ein Programmgraphknoten, der mit der Anweisung y beschriftet ist, die x im While-Programm folgt, falls diese existiert. Weitere Nachfolger existieren nur, wenn x weitere Unterprogrammfragmente einbettet (wie z. B. “while“, “if“, “atomic“, siehe 3.1.5.1.4.1). Die weiteren Nachfolger repräsentieren diese Programmfragmente und sind Unterprogrammgraphen.
Ein Programmgraph ist ein Baum[1].

Für ein Beispiel siehe Abbildung 3.2.

[1] Enthält also keine Schleifen und führt auch keine Unterbäume zusammen

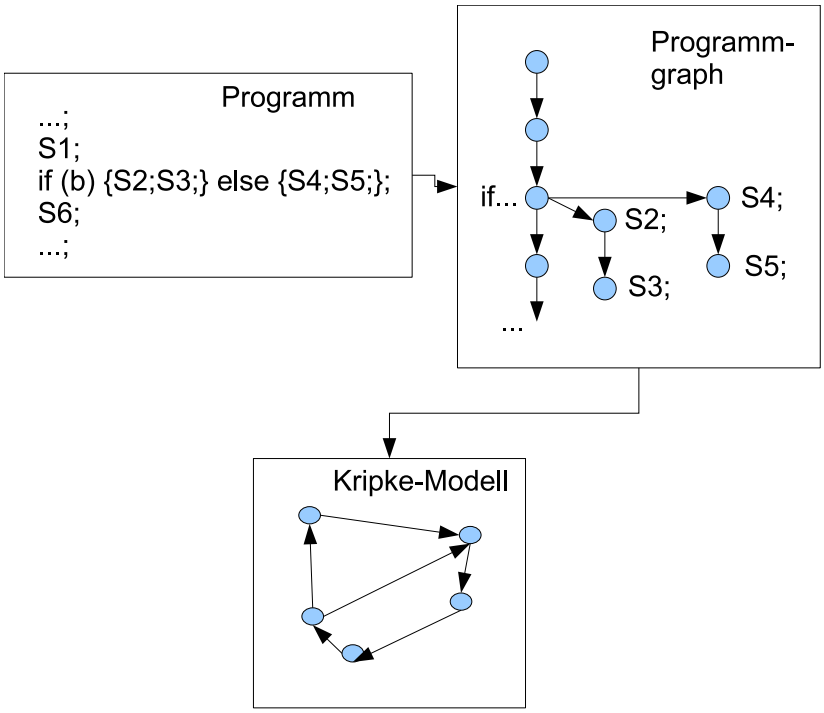


Abbildung 3.3: Pipeline

3.1.5.1.3 Prinzip Das Prinzip sei hier anhand eines einfachen Beispiels erklärt. Dabei wird angenommen, dass hier nur ein Programmfragment von einem einzigen Prozess bearbeitet wird und das deterministisch, anhand der Ausführungsumgebung, genau eine mögliche “Zukunft“ gewählt wird.

Nach dem Parsen des “While“-Programmes wird das Fragment in einen Programmgraphen (siehe 3.1.5.1.2) umgewandelt. Eine leere Kripkestruktur wird im Speicher erzeugt und der Prozess beginnt nun, den Programmgraphen von seiner Wurzel an abzuarbeiten.

An jedem Knoten des Programmgraphen, in dem der Prozess seine Arbeit beginnt, wird nun ein Knoten in der Kripkestruktur erzeugt. Dieser Knoten enthält zur eindeutigen Beschreibung des Programmablaufs Informationen über seine Umgebung, in der er erzeugt wurde. Dazu gehört z. B. der Knoten des Programmgraphen, als auch Variablenbelegungen des While-Programms. Bis zum Ende des Programmfragments wird dieser Schritt wiederholt, außer, es wird versucht einen Knoten anzulegen, der schon in der Kripkestruktur existent ist. An dem Punkt kann die Prozedur vorzeitig abgebrochen werden. Das Resultat ist eine Kripkestruktur, die genau einen Programmdurchlauf beschreibt. Graphisch dargestellt ist der Prozess in Abbildung 3.3.

3.1.5.1.3.1 Erweiterung Die vorher beschriebene Methode ist ausreichend, um genau einen Programmdurchlauf eines Prozesses zu beschreiben. Um einige Grade komplexer wird es, wenn mehrere Prozesse parallel laufen und diese sogar noch auf ein und die sel-

ben Variablen zugreifen. Dies führt dazu, dass an Punkten, an dem alle oder mehrere Prozesse jeweils eine Anweisung ausführen könnten, alle Möglichkeiten vom Kripkemodell modelliert werden müssen.

Es ist nicht Aufgabe einer Modellierungssprache eine bestimmte Prozessreihenfolge vorzugeben.

3.1.5.1.4 While-Programme

While-Programme gliedern sich in drei Teile:

- Deklaration. Hier werden die verfügbaren Variablen deklariert und deren Wertebereiche angegeben. Es sei darauf hingewiesen, dass ein Überschreiten bzw. Unterschreiten von *min* oder *max* zum Abbruch der Modellkonstruktion führt.

```
Decl
  varName [min,max];
  ...
endDecl;
```

- Methoden-/Prozessteil. Hier werden Methoden bzw. Prozesse als While-Programmfragment beschrieben. Jede Methode ist gleichzeitig auch ein Prozess und umgekehrt. Es hängt nur davon ab, wie sie aufgerufen werden, ob sie als Methode oder Prozess ausgeführt werden.

```
Proc procName ()
{
  ...
} endProc;
...
```

- Initialisierung. Dieses Programmfragment wird vom initialen Prozess ausgeführt.

```
Init
{
  ...
} endInit;
```

Der Gesamtaufbau der Syntax kann in 3.19 in Erfahrung gebracht werden. Die Syntax der einzelnen Kommandos folgt in 3.1.5.1.4.1.

3.1.5.1.4.1 While Kommando Syntax/Semantik Die Syntax der einzelnen Kommandos ist jeweils als PEG (siehe [**?**]) angegeben.

Assignments

VAR "=" ARITH
Weist VAR den Wert, zu dem ARITH auswertet, zu, wobei ARTIH ein beliebiger arithmetischer Ausdruck sein kann.

If

"if" "(" BOOL ")" "{" MBODY1 "}" {"else" "{" MBODY2 "}" }?
Falls BOOL zu wahr ausgewertet wird, wird als nächstes MBODY1 ausgeführt. Wertet BOOL zu falsch aus, wird, falls der "else"-Teil vorhanden ist, MBODY2 ausgeführt, andernfalls kommt die nächste Anweisung zur Ausführung.

while

"while" "(" BOOL ")" "{" MBODY "}"
Solange BOOL zu wahr ausgewertet wird, wird immer wieder MBODY ausgeführt.

Skip

"skip"
Leere Anweisung.

Spawn

"spawn" VAR "(" ")"
Startet einen neuen Prozess, repräsentiert durch die Methode mit dem Namen VAR.

Atomic

"atomic" "{" MBODY "}"
Führt MBODY ohne Unterbrechung durch andere Prozesse aus. Die einzige Ausnahme ist eine Wait- oder Block-Anweisung innerhalb von MBODY.

Call

"call" VAR "(" ")"
Springt zur ersten Anweisung der Methode VAR. Terminiert die Methode, wird die Ausführung bei der Anweisung nach der Call-Anweisung, die den Sprung veranlasst hat, fortgesetzt.

Block/Wait

"block" "until" "(" BOOL ") oder "wait" "until" "(" BOOL ")

Blockiert die Ausführung des Prozesses, bis BOOL zu wahr auswertet. Dies gilt auch für die Ausführung innerhalb einer "atomic" Umgebung.

3.1.5.1.4.2 Programmbeispiel Für ein einfaches Programmbeispiel siehe Abbildung 3.4. Das Beispiel modelliert den von Dekker erdachten Algorithmus für wechselseitigen Ausschluss zweier Prozesse. Die Idee hinter dem Algorithmus ist, dass jeder Prozess seinen kritischen Bereich (`mutex = mutex + 1; mutex = mutex -1;`) durch eine Vorbedingung absichert. Dies geschieht durch entsprechendes Setzen der Variablen `turn`, welche garantiert, dass jeder Prozess einmal an die Reihe kommt, und `a` bzw. `b`, welche signalisieren, dass der jeweils andere Prozess seinen kritischen Bereich verlassen hat, bzw. auf einen Neueintritt wartet.

3.1.5.1.5 Modellbildung Die Modellbildung gliedert sich in in drei Klassen, die jeweils eine dedizierte Funktion übernehmen und jeweils die ihr direkt untergeordnete Klasse steuern (siehe 3.5):

- Sandbox: Erzeugung der initialen Umgebung, der Programmgraphen und Verwalter über Einsprungspunkte[2] in die Programmgraphen, Steuerung aller Kontexte
- Context: Bildung der Kripkestruktur parallel zu allen anderen Kontexten, Verwaltung der Prozesse
- Prozess: führt Programm aus

3.1.5.1.5.1 Sandbox Die Sandbox stellt den Rahmen der Modellbildung dar. Sie ist hauptsächlich für die Erzeugung der Programmgraphen und für die Steuerung der einzel-

[2] Wurzelknoten

```
Decl
  a [0,1];
  b [0,1];
  turn [0,1];
  mutex [0,2];
endDecl;
Proc p0 ( )
{
  while (1 < 2)
  {
    a = 1;
    turn = 0;
    wait until ( ( b == 0 ) || ( turn == 1 ) );
    mutex = mutex + 1; mutex = mutex -1;
    a = 0;
  };
} endProc;
Proc p1 ( )
{
  while (1 < 2)
  {
    b = 1;
    turn = 1;
    wait until ( ( a == 0 ) || ( turn == 0 ) );
    mutex = mutex + 1; mutex = mutex -1;
    b = 0;
  };
} endProc;
Init
{
  atomic
  {
    a = 0; b = 0; turn = 0; mutex = 0;
    spawn p0 ( );
    spawn p1 ( );
  };
} endInit;
```

Abbildung 3.4: Aufbau eines einfachen While-Programms

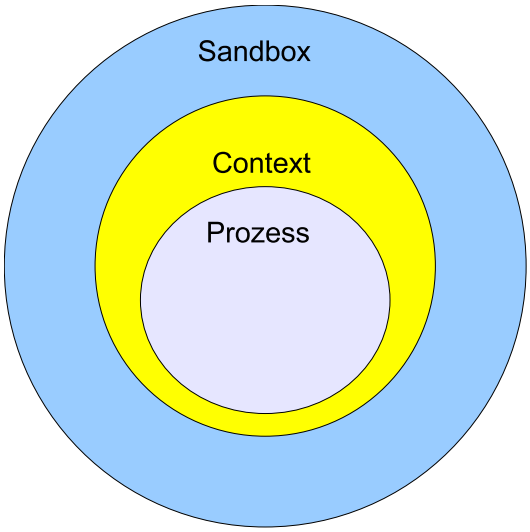


Abbildung 3.5: Schalenmodell

nen Kontexte zuständig.

1. Erzeuge für jede Methode einen eigenen Programmgraphen.
2. Erzeuge eine Variablenumgebung env_{init}.
3. Erzeuge einen Kontext, der die Methode “INIT“ unter einer Kopie von env_{init} ausführt
4. Für jeden Kontext c:
4.1 Wenn c n (mit $n > 1$) Prozesse verwaltet, klone c $n - 1$ mal. Jeder der n Kontexte führt einen anderen Prozess als nächstes aus.
5. Für jeden Kontext c:
5.1 Lasse jeden Kontext c Algorithmus 5 ausführen.
6. Für jeden Kontext c:
6.1 Entferne Kontext c, falls c terminiert hat.
7. Ist die Anzahl laufender Kontexte ungleich null, gehe zu 4.

Algorithmus 4 : Sandbox

3.1.5.1.5.2 Kontext Ein Kontext verwaltet:

- Eine eigene Variablenumgebung
- laufende Prozesse innerhalb des Kontexts
- Indikator, welcher Prozess als nächstes ausgeführt wird
- Knoten k_{alt} im Kripke-Modell, der zuletzt erzeugt wurde

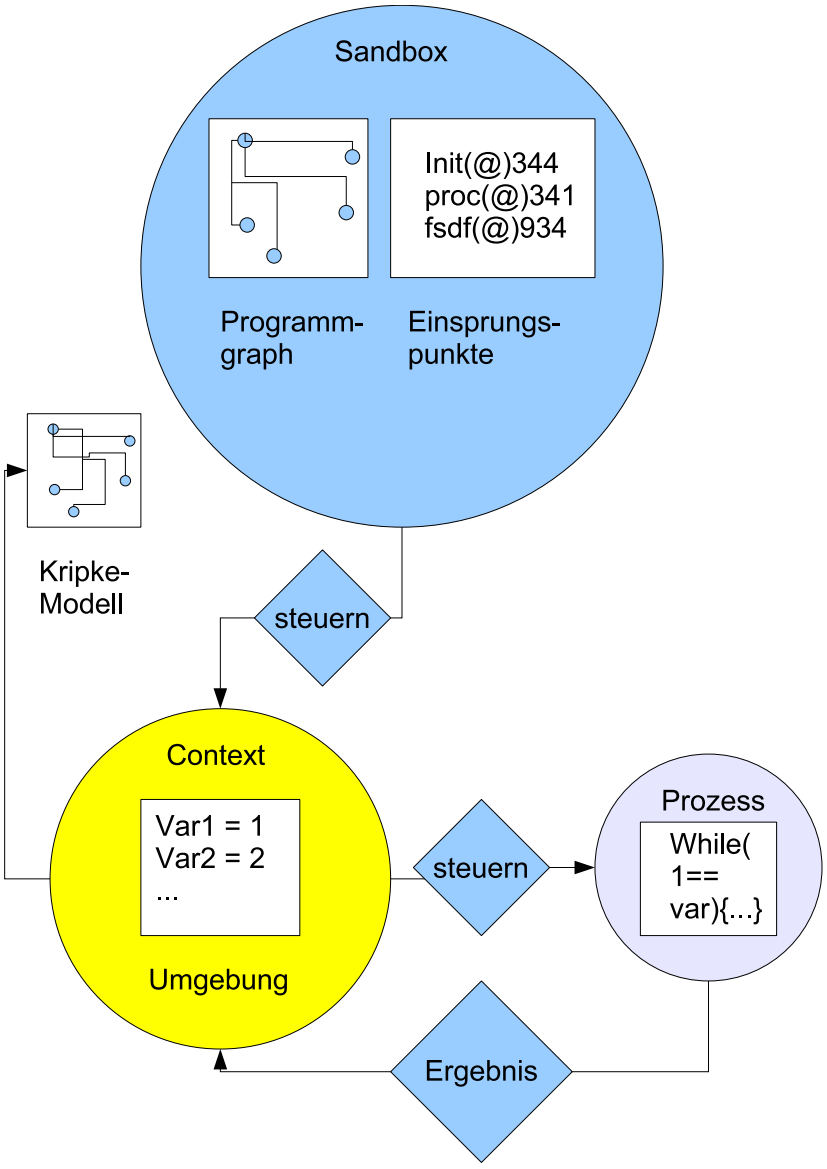


Abbildung 3.6: Ablauf

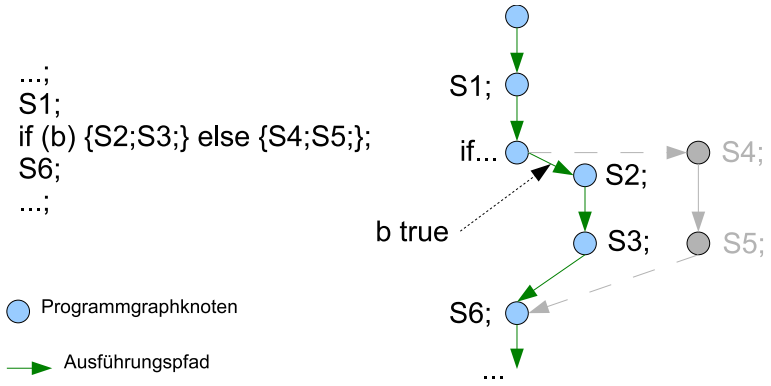


Abbildung 3.7: If Ausführung

Der Algorithmus, der abgearbeitet wird, entspricht diesen Punkten:

1. Lasse den zugewiesenen Prozess laufen. Speichere alle Programmgraphknoten, die der Prozess durchläuft, falls er nicht sofort blockt oder terminiert. Erzeuge ggf. neue Prozesse wenn der laufende Prozess es anfordert.
   1.1. Konnte der Prozess keinen Fortschritt machen (ein Knoten wurde zweimal besucht) forciere einen Block.
2. Entferne den Prozess falls er terminiert hat.
3. Erzeuge einen temporären Kripke-Knoten k anhand der aktuellen Umgebung.
4. Existiert ein zu k äquivalenter Knoten in der Kripkestruktur bereits, terminiere.
5. Füge k der Kripke-Struktur hinzu. Erzeuge eine Transition im Kripke-Modell von k_{alt} zu k.
6. $k_{alt} = k$.
7. Signalisiere Termination (alle Prozesse haben terminiert) oder einen Block.

Algorithmus 5 : Kontext

3.1.5.1.5.3 Prozess Ein Prozess führt eine Anweisung an einem Programmgraphknoten gemäß 3.1.5.1.4.1 aus. Nach Abarbeitung der Anweisung, signalisiert der Prozess dem verwaltenden Kontext, ob er anhalten[3] (blocken) oder weiterlaufen möchte[4]. Wenn ein Prozess keine gültige Anweisung mehr ausführen kann, signalisiert er seine Termination.

Falls die Variablenumgebung geändert werden muss, wird direkt auf die Variablenumgebung des verwaltenden Kontextes zurückgegriffen. Bedingte Anweisungen (z. B. “if“) werden über Auswertungsmethoden des Kontexts ausgewertet.

[3]Dies geschieht immer dann, wenn die Umgebung des Kontexts sich verändert hat. Also wenn ein neuer Prozess erzeugt werden soll, der Wert einer Variablen sich geändert hat oder mit “wait“/“block“ dies explizit angefordert wird.

[4]Diese Situation tritt immer dann ein, wenn eine Anweisung bearbeitet wurde, die Umgebung sich aber nicht geändert hat.

Beispiel

Sei das Programmfragment aus Abbildung 3.7 gegeben. Es wird an dieser Stelle vereinfacht angenommen, dass S_i nichtblockende Anweisungen sind, b zu wahr auswertet und erst bei einer Umgebungsänderung geblockt wird. Der Prozess wird zuerst S_1 am aktuellen Programmgraphknoten bearbeiten und dann den darauf folgenden Knoten auswerten. Dieser ist mit einer If-Anweisung markiert. Der Prozess wird nun bei dem verwaltenden Kontext anfragen, zu welchem Wert b auswertet. Wie angenommen, wird der Prozess darauf die Anwort "wahr" erhalten. Da If Unterprogrammfragmente einbettet, speichert der Prozess den Nachfolgeknoten S_6, also den Knoten, welcher der If-Anweisung folgt, separat, um nach Abarbeitung des Unterprogrammfragments, dort die Ausführung fortzusetzen. Nach der Speicherung, verzweigt der Prozess in den Programmgraphenteil, der ausgeführt werden soll, wenn If zu wahr auswertet. Nach Abarbeitung von S_2 und S_3 wird die Programmausführung beim vorher gesicherten Knoten S_6 fortgesetzt.

3.1.5.1.6 Anwendung Im Allgemeinen bietet dieses Package kein von einer externen Anwendung verwertbares Interface. Der Vollständigkeit halber seien hier jedoch die Schritte genannt, um eine Kripkestruktur zu erzeugen :

```
SandBox s = new SandBox();

WhileParser p = new WhileParser();

PNode pt = p.parse(WHILEPROG);

s.exec(pt);

KripkeModel k = s.ka.getKripke(true);
```

3.1.5.1.7 Änderung des Verhaltens der Modellbildung Es ist möglich, die Art und Weise, wie ein Modell gebildet wird, zu beeinflussen. Jedoch sei darauf hingewiesen, dass diese Möglichkeiten experimenteller Natur sind und deren Ergebnisse (vorallem in Kombination mit anderen Einstellungen) von Endlosschleifen bis zu extrem ungenauen Modellen reichen können.

3.1.5.1.7.1 Context.java *`static ContextMangle contextMangle = ContextMangle.NONE;`*

Beeinflusst das Verhalten, wie die Umgebungsinformationen eines Knotens der zu konstru-

ierenden Kripke-Struktur zusammengefasst werden. In diesem Falle sind dies die Informationen über alle laufenden Kontexte. Die Standardeinstellung (`ContextMangle.NONE`) führt dazu, dass keine Informationen über laufende Kontexte gespeichert werden. `ContextMangle.NumberOf` speichert nur die Anzahl der laufenden Kontexte, `ContextMangle.FULL` speichert die internen Kennungen der Kontexte. Gerade letztere Einstellung produziert einen Baum und lässt die Modellbildung nur terminieren, wenn auch **alle** Prozesse terminieren[5] können. Die Kripke-Struktur wird sehr gross und es wird eine exorbitante Menge an Arbeitsspeicher benötigt.

`static ProcessMangle processMangle = ProcessMangle.FULL;`

Analog zu `ContextMangle`, aber bezogen auf Prozessinformationen. Weitere Einstellungsmöglichkeiten sind `NumberOf` und `NONE`. Letztere Einstellungsmöglichkeiten verringern die Genauigkeit der erzeugten Kripke-Struktur.

`static StateMangle stateMangle = StateMangle.FULL;`

Hat nur eine Auswirkung, falls `processMangle = ProcessMangle.FULL` ist. Entscheidet, ob die Knotennummer des Programmgraphen im Knoten der Kripkestruktur notiert wird (im Falle `FULL` wird dies getan). Weitere Einstellungsmöglichkeit: `NONE`: es werden keinerlei Informationen gespeichert. Diese Einstellung kann die Genauigkeit der Kripke-Struktur u. U. drastisch reduzieren.

`private static int maxProcesses = 256;`

Maximale Anzahl der Prozesse, die gleichzeitig von einem Kontext verwaltet werden.

3.1.5.1.7.2 Process.java *`static final boolean allBlock = true;`*

Lässt den Prozess nur eine Anweisung ausführen, außer es wird eine "`atomic`"-Anweisung ausgeführt. Setzen der Variablen auf `false` lässt einen Prozess solange unterbrechungsfrei laufen, bis eine Zuweisung ausgeführt wurde, explizit mit "wait" geblockt wird oder ein neuer Prozess erzeugt werden soll. Abhängig vom While-Programm kann dies u. U. die Genauigkeit verringern.

`private static int maxRecursion = 256;`

Maximale Rekursionstiefe für Methodenaufrufe.

3.1.5.1.8 Problemanalyse Mit Hilfe dieser leicht erweiterten While-Sprache ist es möglich, schneller und einfacher größere Modelle zu erstellen, als dies mit rein manueller

[5] Im Sinne von: jeder Prozess gelangt immer an eine Anweisung ohne Nachfolger

Eingabe zu bewerkstelligen wäre.

Die Sprache und die Implementation an sich weisen jedoch einige Einschränkungen auf, die hauptsächlich der knappen Zeit anzulasten sind, auf die hier kurz eingegangen werden soll :

- "While" ist zu sehr eine traditionell Programmiersprache. Man mag auf den ersten Blick meinen, dass dies von Vorteil wäre, jedoch, vergleicht man diese Sprache mit problemspezifischen Sprachen wie z. B. PROMELA aus dem Spin-Projekt (siehe [**?**]), fällt auf, dass die dort modellierten Modelle sich wesentlich kürzer beschreiben lassen. While trägt zu viel syntaktischen Ballast mit sich herum. Vor allem Schlüsselwörter vergrößern den Schreibaufwand erheblich.

- Keine Typisierung. Dies ist eine Einschränkung dieser Implementation. Typen und Enumerationen erhöhen die Lesbarkeit und vereinfachen oft die Modellierung, erfordern jedoch eine aufwändigere Variablenbehandlung.

- Prozess-/Methodenaufrufe sind nicht parametrisiert. Dies ist eine Einschränkung dieser Implementation. Es ist möglich, Methoden Parameter zu übergeben, jedoch ist dies nur über call-by-reference möglich. Da damit auch eine Übergabe von Konstanten unmöglich ist, wurde die Möglichkeit zur Parameterübergabe nicht weiter verfolgt. Die Implementation ist unvollständig. Zur vollständigen Unterstützung von parametrisierten Aufrufen ist eine andere Konzeption der Methodenbehandlung notwendig, als dies hier ansatzweise verwirklicht wurde.

- Keine lokalen Variablen. Dies ist eine Einschränkung dieser Implementation und steht (intern) eng im Zusammenhang mit den parametrisierten Methodenaufrufen.

- Keine Queues. Dies ist eine Einschränkung dieser Implementation. Es ist zwar möglich Queues anzulegen, und über vier Anweisungen diese zu steuern, jedoch fehlen Anweisungen auf bestimmte Konfigurationen der Queue zu warten, was diese quasi unbrauchbar macht.

Das Grundkonzept an sich hat jedoch den Vorteil, dass damit eine verteilte Modellerzeugung möglich ist. Kontexte oder Prozesse können als Threads realisiert werden und auf separaten Prozessoren oder Netzknoten eines Clusters laufen. Unter Umständen ist auch eine asynchrone Ausführung der Sandbox möglich.

3.2 Automaten

Um die Vision eines universellen und automatischen Analysewerkzeugs auf der Basis von Automaten zu verwirklichen, wurden auf der Suche nach einem Kern dieses Werkzeugs verschiedene schon vorhandene Automatenbibliotheken untersucht. Es stellte sich heraus, dass für vorgesehene zukünftige Erweiterungen keine Bibliothek allen Anforderungen gerecht werden konnte. Insbesondere ließen die untersuchten Bibliotheken ein flexibles Alphabetkonzept vermissen, das heißt sie bieten nur ein Alphabet fester Größe mit unveränderlichem Elementtyp. Bei der Darstellung von Mengen transitionsbewirkender Zeichen sind die bestehenden Bibliotheken außerdem zu sehr auf den Anwendungsfall „Mustererkennung“ ausgelegt, was für automatische Analysen, wie sie von der Projektgruppe geplant waren, von Nachteil ist. Daher wurde der Entschluss gefasst, eine komplett neue Bibliothek für Automaten zu verfassen, die sich einerseits an Konzepten schon bestehender Bibliotheken orientiert, andererseits aber – insbesondere in den o. g. Punkten – eine größere Flexibilität ermöglicht.

Im Folgenden wird zuerst das Fundament der Bibliothek beschrieben. Es wird auf die elementaren Bausteine eines jeden Automatens eingegangen und beschrieben, wie diese in der Bibliothek verwirklicht wurden. Der Kern der Bibliothek ist im ersten Semester des Projektes entstanden. Im zweiten Semester wurde dieser dann einer gründlichen Revision unterzogen. Die größten Änderungen ergaben sich bei der Darstellung von Mengen transitionsbewirkender Zeichen.

Wir wenden uns nun dem Gerüst des AAA-Automaten zu.

3.2.1 Aufbau eines Automaten

Alle Automaten verwenden Zustände und Transitionen. Daher erben alle Implementierungen von der abstrakten Klasse `aaa.automaton.Automaton`, deren Aufgabe es ist, eine Verwaltung dieser Objekte anzubieten. Alles, was spezifischer ist, insbesondere Operationen, wird in abgeleiteten Klassen implementiert.

Es besteht die Möglichkeit, andere ähnliche Automatenmodelle durch Vererbung zu realisieren. Geändertes und neues Verhalten kann dann implementiert oder überschrieben werden.

Ein Automat ist eine graphenähnliche Struktur mit Zuständen als Knoten und Transitionen als Kanten, die mit **Triggern** beschriftet sind. Ein **Trigger** repräsentiert die Menge der Symbole, die eine Transition auslösen können. Das Trigger-Konzept wurde in der zweiten Phase des Projekts stark verallgemeinert, weshalb an dieser Stelle für eine detail-

lierte Darstellung auf Abschnitt 3.2.3.1 verwiesen sei. Die Automaten sind in der Regel unvollständig, das heißt, dass nicht für jeden Zustand und für jedes Symbol des Alphabets eine Transition existiert, die diesen Zustand erreicht oder verlässt und die durch dieses Symbol geschaltet wird.

3.2.1.1 Zustände

Automaten bestehen aus Zuständen, die durch Instanzen der Klasse `State` repräsentiert werden. Da dies alle Automaten betrifft, ist die Verwaltung der Zustände in der abstrakten Klasse `aaa.automaton.Automaton` implementiert. Dies geschieht mit Hilfe der internen Liste `states`. Zustände werden entweder explizit durch Aufruf der Methode `addState()` zu dieser Liste hinzugefügt oder implizit beim Hinzufügen einer Transition, die einen noch nicht im Automaten enthaltenen Zustand betrifft.[6] Dabei ist darauf zu achten, dass ein Zustand nicht mehrfach hinzugefügt wird, ansonsten führt dies zu einer Ausnahme.

Ein Zustand hat folgende Eigenschaften:

1. `owner` referenziert den Automaten, zu dem der Zustand gehört.
2. `index` ist die Nummer des Zustands in der `states`-Liste seines Automaten.
3. `key` dient zur eindeutigen internen Identifikation (siehe Abschnitt 3.2.1.4), er induziert außerdem eine totale Ordnung auf den Zuständen.
4. `string` ist eine textuelle Beschreibung.
5. `label` ist ein beliebiges mit einem Zustand assoziiertes Objekt.
6. `accepting` und/oder `initial` oder keines von beiden

Der Index wird intern neu gesetzt, wenn sich die `states`-Liste ändert, das `owner`-Attribut kann nur einmal intern beim Hinzufügen eines Zustands zu einem Automaten gesetzt werden. Dabei ist die Existenz eines Startzustandes für einen Automaten eine Minimalanforderung. Die anderen Attribute sind auch (nachträglich) durch einen Benutzer veränderbar. Bei der Veränderung des `key`-Attributs muss natürlich die Eindeutigkeit erhalten bleiben.

Es existieren vier verschiedene Konstruktoren, die einen neuen Zustand erzeugen:

1. `State(Object label, String string, boolean accepting, boolean initial)`: Der neue Zustand erhält genau die übergebenen Eigenschaften.

[6] In der Klasse OBDD, die ebenfalls von Automaton erbt, wird allerdings erwartet, dass die beteiligten Zustände dem OBDD bereits bekannt sind.

2. `State(State state)`: Dieser Konstruktor kopiert den übergebenen Zustand und erstellt einen Klon von ihm, alle Eigenschaften bis auf das `owner`-Attribut sind identisch.

3. `State(String string)`: Hier wird ein Zustand einfach durch eine textuelle Beschreibung erzeugt; `label` wird auf `null` und `accepting` und `initial` werden auf `false` gesetzt.

4. `State(String string, boolean initial, boolean accepting)`: Bei dieser Variante besteht zusätzlich zu 3. die Möglichkeit mit `initial` festzulegen, ob der neue Zustand Startzustand sein soll bzw. mit `accepting` festzulegen, ob der neue Zustand akzeptierend sein soll.

Zusätzlich kann durch die Methode `getOutgoingTransitions()` die Menge der vom Zustand ausgehenden Transitionen ermittelt werden. Die Methode `getAdjacentStates()` dagegen gibt alle unmittelbaren Folgezustände eines Zustands zurück.

3.2.1.2 Transitionen

Genauso wie es bei Zuständen der Fall ist, besitzt jeder Automat eine Menge von Transitionen, die durch Instanzen der Klasse `Transition` dargestellt werden. Intern sind alle Transitionen eines Automaten in der Liste `transitions` von `Automaton` zusammengefasst. Diese Liste enthält für jeden Zustand z eine `Map`, die die zu z adjazenten Zustände als Schlüssel enthält. Ist ein Zustand z' solch ein Schlüssel, so bildet ihn die `Map` auf das `Transition` Objekt von z nach z' ab. Diese Darstellung ermöglicht es, effizient zu überprüfen, ob bereits eine Transition zwischen zwei gegebenen Zuständen existiert.

Eine Transition hat folgende Eigenschaften:

1. `from` ist der Zustand, von dem die Transition ausgeht. Die Methoden `Transition.setFrom(State)` kann nur aufgerufen werden, wenn kein Eigentümer der Transition existiert.

2. `trigger` ist ein `Trigger` Objekt, welches die Menge der Symbole repräsentiert, die die Transition feuern. Die Automatenbibliothek stellt verschiedene Triggertypen zur Verfügung, die sich in ihren Vor- und Nachteilen für unterschiedliche Anwendungfälle unterscheiden. Für eine detaillierte Darstellung des überarbeiten Triggerkonzepts sei auf Abschnitt 3.2.3.2 verwiesen.

3. `to` ist der von der Transition erreichte Zustand. Ebenso kann die Methode `Transition.setTo(State)` nur aufgerufen werden, wenn kein Eigentümer der Transition existiert.

4. `owner` ist der mit dem Trigger assoziierte Automat. Der `owner` wird beim Hinzufügen der Transition zum Automaten automatisch gesetzt.

Die Übertragung der formalen Darstellung von Transitionen in eine konkrete Implementierung erfordert es, sich auf Konventionen im Umgang mit ihnen zu einigen.

Zuerst einmal wird zwischen zwei Transitionstypen unterschieden: *leere* und *markierte* Transitionen. Für leere Transitionen gilt `getTrigger() == null`, markierte sind immer mit einem `Trigger`- Objekt assoziiert. Wenn für eine markierte Transition `getTrigger().isFireable() == false` gilt, d. h. wenn der Trigger kein Zeichen enthält (wie z. B. das OBDD, welches nur aus der 0-Senke besteht), dann heißt diese Transition *tote* Transition. Tote Transitionen können z. B. auch vorübergehend bei der graphischen Eingabe entstehen. Durch Aufruf der Methode `removeDeadTransitions()` werden sie entfernt.

ε-Transitionen werden sowohl in endlichen Automaten als auch in Büchiautomaten als *leere* Transitionen modelliert. Es ist jederzeit möglich, leere, markierte oder tote Transitionen zum Automaten hinzuzufügen. Intern sind pro Zustandspaar aber höchstens eine markierte und höchstens eine leere Transition vorhanden. Eine markierte Transition, die dem Automaten hinzugefügt werden soll, wird mit einer evtl. vorhandenen markierten Transition verschmolzen, das heißt die Symbolmengen (Trigger), mit denen die Transitionen markiert sind, werden vereinigt. Soll eine leere Transition (oder auch ε-Transition) hinzugefügt werden und es existiert zwischen dem jeweiligen Zustandpaar schon eine leere Transition, dann wird die neue ignoriert.

Methoden, die beliebige Transitionen liefern (z. B. `Automaton.getAllTransitions()` oder `State.getOutgoingTransitions()`), liefern ungefiltert alle vorhandenen Transitionen (auch tote). Die Methode `State.getAdjacentStates()` liefert allerdings nur über nicht tote Transitionen erreichbare Zustände. Die Methoden in der Hilfsklasse `GraphAlgorithms` berücksichtigen keine toten Transitionen.

Durch den Aufruf der Methode `removeTransition(Transition)` können Transitionen aus dem Automaten entfernt werden. Diese liefert die entfernte Transition zurück, ihr Eigentümer ist nun `null`. Man kann nun die Zustände verändern und die Transition später wieder einfügen.

Es existieren verschiedene Konstruktoren, die eine neue Transition erzeugen.

- `Transition (State from, State to)` hier wird zunächst kein übergangsbewirkendes Zeichen angegeben; die Transition ist zunächst einmal eine leere Transition, ihr `trigger` Attribut ist `null`.

- `Transition (State from, Trigger trigger, State to)` hier wird der Transition ein `Trigger` Objekt mit übergeben, welches die Menge der Symbole repräsentiert, die die Transition feuern.
- Mit `Automaton.addTransition(Transition newTransition)` kann man einem Automaten eine Transition hinzufügen.
- Analog zum zweiten Transitionskonstruktor, fügt `Automaton.addTransition-(State from, Trigger trigger, State to)` dem Automaten die entsprechende Transition hinzu.
- Die Methode `Automaton.addTransition(State from, State to, Symbol symbol)` fügt dem Automaten eine Transition, welche durch ein einzelnes Symbol getriggert wird, hinzu.
- Die Methode `Automaton.addTransition(String symbol, State from, State to)` generiert aus dem String ein neues Symbol und fügt die entsprechende Transition dem Automaten hinzu.

Folgende Methoden stehen in der Klasse `Transition` zur Verfügung:

- `getTrigger()` liefert das `Trigger`-Objekt der Transition zurück, das die Menge von Zeichen repräsentiert, die die Transition auslösen können. Durch Aufruf der Methode `Trigger.covers(Symbol)` kann überprüft werden, ob ein bestimmtes Zeichen die Transition auslösen kann. Die Methode `Trigger.getCovered-Symbol()` liefert irgend ein Symbol, das die Transition schaltet. Mit der Methode `Trigger.isFireable()` kann überprüft werden, ob die Transition überhaupt von einem Zeichen ausgelöst werden kann.
- `Transition.setTrigger(Trigger)` setzt das `trigger`-Attribut neu. Der Automat, der die Transition enthält, wird intern darüber benachrichtigt, so dass Maßnahmen eingeleitet werden können, um die oben beschriebenen Bedingungen zu erhalten. Dies ist z. B. nötig, um einer Transition neue akzeptierende Symbole hinzuzufügen. Da `Trigger`-Objekte unveränderlich sind, muss der Transition ein neues Objekt assoziiert werden. Dies geschieht durch den Aufruf `trans.setTrigger(trans.get-Trigger().join(newTrigger))`. Die `join` Methode von `trigger` gibt ein neues `Trigger` Objekt, welches die Symbole beider Trigger akzeptiert, zurück.
- `getFrom()` und `getTo()` liefert der Start- bzw. Zielzustand der Transition.
- `toString()` liefert eine Stringdarstellung der Transition bestehend aus der `toString()` Methode der Start- und Zielzustände und der spezifischen `toString()`-Methode des Trigger-Objekts.

- `isEmpty()` gibt Auskunft darüber, ob der Transition ein `Trigger` Objekt assoziiert ist.

3.2.1.2.1 Behandlung von ε-Transitionen Eine ε-Transition vom Zustand a zum Zustand b ermöglicht einen Zustandsübergang von a nach b, ohne dass der Automat ein Eingabezeichen liest. ε-Transitionen sind allerdings nicht in jedem Modell sinnvoll, das durch eine von `Automaton` erbende Klasse dargestellt wird. Z. B. sind in einem Kripkemodell alle Transitionen gewissermaßen ε-Transitionen , also leere Transitionen. Auch für OBDDs machen ε-Transitionen keinen Sinn. ε-Transitionen sind aus diesem Grund nur eine andere semantische Sicht auf leere Transitionen. Sie werden äquivalent behandelt.

In `FiniteAutomaton` und Büchiautomaten erlauben ε-Transitionen es, gewisse Automatenoperationen wie z. B. Vereinigung, Konkatenation und Kleeneschen Abschluss auf eine sehr intuitive Weise zu realisieren. Aus diesem Grund werden ε-Transitionen in einem `FiniteAutomaton` und Büchiautomaten standardmäßig *nicht* aufgelöst. Der Benutzer soll die Struktur seiner Automaten auch noch nach einer Automatenoperation einfach und intuitiv nachvollziehen können.

Wenn der Benutzer also beispielsweise eine ε-Transition mittels `fAutom.addEpsilonTransition(start,to)` einfügt, dann wird diese ε-Transition auch tatsächlich im Automaten erzeugt.[7]

3.2.1.2.2 Die `emptyTransitionsAllowed`-*flag* Das Berechnen eines Automaten ohne ε-Transitionen , der zu einem gegebenen Automaten mit ε-Transitionen äquivalent ist, ist relativ rechenintensiv. Effizienter ist es, die bei den oben genannten Operationen verwendeten ε-Transitionen sofort beim Einfügen ε-Transitionen lokal durch gewöhnliche Transitionen zu ersetzen, da nicht alle Transitionen des Automaten, sondern nur diejenigen betrachtet werden, die zu dem Zielzustand der ε-Transition inzident sind. Dies wird in der AAA-Bibliothek durch das `emptyTransitionsAllowed`-*flag* der `Automaton`-Klasse ermöglicht. Hierzu muss die *flag* mittels `setAllowEmptyTransitions(false)` gesetzt werden. Dies kann auch durch die Wahl eines geeigneten Konstruktors des endlichen Automaten geschehen.

Während `Automaton.emptyTransitionsAllowed() == false` ist, wird jede hinzugefügte ε-Transition durch normale Transitionen ersetzt und ggf. werden weitere Zustände auf akzeptierend gesetzt. Beim Setzen der *flag* werden bestehende ε-Transitionen aufgelöst; es wird die Methode `removeEpsilonTransitions()` implizit aufgerufen, in der alle ε-Transitionen resp. leeren Transitionen aufgelöst werden. Der Automat wird dabei

[7] Wie in 3.2.1.2 beschrieben wird ein `Transition` Objekt mit leerem Trigger erzeugt.

allerdings nicht deterministisch gemacht, da sich die Trigger zweier ausgehender Transitionen des selben Zustands weiterhin überlappen können und mehrere Startzustände unberücksichtigt bleiben (siehe hierzu Abschnitt 3.2.1.7.6 Determinisierung).

Es stehen folgende Methoden der Klasse `FiniteAutomaton` zur Erstellung von ε-Transitionen zur Verfügung:

- `addAndSubstituteEpsilonTransition(State from, State to)` löst die einzufügende ε-Transition sofort auf, d. h. die Methode erstellt für jede ausgehende Transition des Zielzustands `to` eine Transition vom Startzustand `from` mit dem selben Ziel und dem gleichen Trigger.

- `addEpsilonTransition(State from, State to)` fügt in Abhängigkeit der `emptyTransitionsAllowed-`*flag* entweder eine ε-Transitionen ein oder substituiert diese direkt.

3.2.1.3 Alphabete und Symbole

Die Realisierung von Alphabeten soll zum einen möglichst flexibel sein, das heißt offen bezüglich der Gestalt der Objekte, aus denen Alphabete bestehen. Zum anderen ist jedoch darauf zu achten, dass intern eine effiziente Verarbeitbarkeit gewährleistet ist, auch soll nicht mehr Speicherplatz als notwendig benutzt werden.

Folgender Kompromiss wird beiden Anforderungen gerecht: Alphabete können aus Instanzen beliebiger Klassen bestehen, die das Interface `Symbol` implementieren. Dieses Interface fordert die Realisierung einer bijektiven Abbildung von Symbol-Instanzen auf Zahlen vom Typ `BigInteger`.

So kann intern, wo die Semantik eines Symbols nicht interessiert, einfach mit diesen Zahlen gerechnet werden, die man durch Aufruf der `getValue()`-Methode einer Symbol-Instanz erhält.

Nach außen hin ist ein Objekt sichtbar, das die Semantik des entsprechenden Symbols wiedergibt. Es ist nicht notwendig, alle Objekte, aus denen ein Alphabet (potentiell) besteht, irgendwo zu speichern. In aller Regel wird man zur Repräsentation nach außen nur Objekte für einzelne Zeichen benötigen, und diese können bei Bedarf schnell konstruiert werden. Dies geschieht für einen gegebenen `BigInteger`-Wert durch Aufruf der Methode `Alphabet.getSymbol(BigInteger)`.

Die geforderte Korrespondenz von Symbol-Instanzen und `BigInteger`-Zahlen stellt keine Einschränkung dar, da Alphabete aufzählbare und ausgesprochen strukturierte Mengen

sind. In der Regel lässt sich die geforderte Abbildungsvorschrift direkt aus der Definition eines Alphabets ableiten.

Folgende Implementierungen von Alphabeten stehen bislang zur Verfügung:

- **BinaryAlphabet**: besteht nur aus 0 und 1
- **CharacterAlphabet**: bietet Unicode-Zeichen an, der zu einem Zeichen gehörige `BigInteger`-Wert entspricht dem `char`-Wert
- **BitvectorAlphabet**: bietet alle Bitvektoren zu einer festen, aber beliebig großen Bitlänge an.
- **AssignmentAlphabet**: Unterklasse von **BitvectorAlphabet**, die Bits werden hier als Variablen interpretiert, die mit Strings benannt sind. Ein Symbol dieses Alphabets repräsentiert eine Variablenbelegung.

Symbole eines Alphabets können zu Mengen zusammengefasst werden. Das geschieht beispielsweise mit Objekten vom Typ `SymbolRangeList`, die eine Anzahl von `SymbolRange`-Objekten enthalten, die Intervalle repräsentieren. Eine einzelne `SymbolRange` ist charakterisiert durch ein Start- und ein Endsymbol und sie repräsentiert die Menge aller Symbole, die zwischen diesen beiden Grenzen liegen. In der zweiten Projektphase wurde auch eine Möglichkeit geschaffen, Symbolmengen in einer nicht intervallorientierten Weise zu repräsentieren. Vgl. dazu Abschnitt 3.2.3.2 auf Seite 129.

3.2.1.4 Schlüssel

In der AAA-Automatenbibliothek besitzen Zustände (`State`) ein Schlüssel-Attribut (`key`-Attribut). Von diesem Attribut wird gefordert, dass es 1). eine Instanz einer Klasse ist, die das Interface `java.util.Comparable` implementiert und 2)., dass es eindeutig ist. Wenn das Schlüssel-Attribut beim Hinzufügen eines Zustands zu einem Automaten nicht gesetzt ist, dann weist der Automat dem Zustand automatisch einen eindeutigen Schlüssel vom Typ `Integer` zu. Da es prinzipiell aber erlaubt ist, beliebige Objekte als Schlüssel-Attribut zu verwenden und die Schlüssel der Zustände innerhalb eines Automatens nicht zwangsläufig vom selben Typ sein müssen, benötigt ein `Automaton`-Objekt einen `java.util.Comparator` um diese vergleichen zu können.

Der Comparator kann explizit im Konstruktor angegeben werden (z. B. `FiniteAutomaton(Alphabet alphabet, Comparator<Comparable> stateKeyComparator)`) ansonsten wird automatisch `aaa.automaton.util.DefaultComparator` benutzt. Im Normalfall muss sich der Benutzer also nicht um die Vergleichbarkeit von Zustandsschlüssel kümmern, die Bibliothek verwaltet die Schlüssel der Zustände automatisch.

Der `aaa.automaton.util.DefaultComparator` kann also auch Instanzen verschiedener Klassen vergleichen. Hierzu benutzt er den Klassennamen der Objekte und berechnet das Ergebnis gemäß ihrer lexikographischen Ordnung. Wenn die Objekte derselben Klasse angehören, findet der Vergleich über die `compareTo` Methode der beiden Objekte statt – dies ist immer möglich, da Schlüssel-Attribute vom Typ `Comparable` sein müssen.

Wenn in einem Automaten also sowohl Zustände mit Integer-Schlüsseln als auch mit String-Schlüsseln vorkommen, dann würde der Vergleich „Integer-Schlüssel“ < „String-Schlüssel“ mit dem `DefaultComparator` immer den Wahrheitswert `true` haben. Innerhalb der „Integer-“ und „String-Zustände“ bleibt die Ordnung der Zustände allerdings arithmetisch bzw. lexikographisch.

Der Vorteil einer Ordnung über der Zustandsmenge ist, dass bei einer geschickten Wahl der Ordnung, die Determinisierung in linearer Zeit durchgeführt werden kann. Dies ist z. B. bei der Konstruktion von Automaten aus Presburger Formeln der Fall.

3.2.1.5 Beispielhafte Konstruktion eines `FiniteAutomaton`

Nachdem alle wesentlichen Bausteine zum Erstellen eines endlichen Automaten vorgestellt wurden, wird im Folgenden eine kleine Nutzung der Bibliothek demonstriert. Wir wollen einen einfachen Automaten konstruieren.

Zunächst wird ein Alphabet benötigt. Ein recht gängiges Alphabet ist ein einfaches Alphabet, das aus normalen Buchstaben besteht. Also wird ein solches referenziert via

```
Alphabet myAlphabet = CharacterAlphabet.getInstance();
```

Dieses Alphabet ist unveränderlich. Man beachte an dieser Stelle, dass Alphabete sehr leicht austauschbar sind, ohne dass weitere Änderungen nötig sind. Nun kann auch ein endlicher Automat erstellt werden.

```
Automaton myAutomaton = new FiniteAutomaton(myAlphabet);
```

Auch bei Aufruf des parameterlosen Konstruktors (`new FiniteAutomaton()`) wird ein `CharacterAlphabet` gewählt, denn das ist das *default*-Alphabet. Ein anderes kann mit der statischen Methode `Automaton.setDefaultAlphabet(Alphabet)` eingestellt werden.

Der Automat ist nun komplett leer – er besitzt keinen einzigen Zustand und somit auch keinen Startzustand. In dieser Situation *soll nicht dauerhaft* verweilt werden. Ein neuer Startzustand wird also erstellt:

```
State initialState = new State("initial");
initialState.setInitial(true);
```

Dieser Startzustand gehört noch zu keinem Automaten. Mittels

```
myAutomaton.addState(initialState);
```

wird dieser Zustand als neuer Startzustand *explizit* zum Automaten hinzugefügt.

Der bisher konstruierte Automat ist noch nicht sehr interessant und erfüllt nur minimale Anforderungen. Es gibt noch keinerlei Transitionen oder akzeptierende Zustände. Nun soll folgendes geschehen:

```
State finalState = new State("final");
finalState.setAccepting(true);
Symbol someSymbol = myAlphabet.createSymbol("a");
Transition someTransition = new Transition
        (initialState, someSymbol, finalState);
myAutomaton.addTransition(someTransition);
```

Ein neuer akzeptierender Zustand und ein Symbol für das Zeichen „a“ werden erzeugt. Dieses Zeichen löst die neu konstruierte Transition vom ersten Zustand zu dem neu erzeugten Zustand aus. Diese Transition wird nun zu dem Automaten hinzugefügt.

Dem Automat wurde unser neuer Zustand `finalState` noch nicht explizit hinzugefügt, doch dies geschieht *implizit* beim Einfügen der Transition.

Es sei noch erwähnt, dass es auch eine Methode mit der Signatur `addTransition(String,State,State)` gibt, deren erstes Argument eine String-Repräsentation eines Symbols ist. Das Alphabet erzeugt aus diesem String eine `Symbol`-Objekt. In dem Beispiel wird allerdings ein Symbol (`someSymbol`) explizit aus einem String erzeugt. Der Automat wird hinter den Kulissen eine Symbolmenge (siehe dazu auch Abschnitt 3.2.3.2 auf Seite 129) erzeugen, die nur aus diesem einzelnen Symbol besteht.

Es gilt auf einiges zu achten. Die Zustände sollten eindeutig einem Automaten gehören. Es sollte nicht unachtsam damit umgegangen werden. Das Eingabezeichen bzw. Symbol sollte passend zu dem Alphabet sein, das bei der Konstruktion des Automaten verwendet wurde. Das sollte aber unter normalen Umständen kein Problem sein, weil man die `createSymbol(String)`-Methode des Alphabets nutzt, um das Symbol zu erstellen.

3.2.1.6 Läufe

Endliche Automaten sollen endlich lange Wörter aus der Sprache, die sie repräsentieren, akzeptieren. Sie lesen dabei jedes einzelne Zeichen eines Wortes nacheinander. Intern nehmen sie dabei nichtdetermistisch einen Zustand bzw. deterministisch mehrere Zustände an. Nach dem Lesen des letzten Zeichens ist der Automat dann in einem akzeptierenden Zustand, wenn das Wort zu der von ihm repräsentierten Sprache gehört, andernfalls in einem nicht akzeptierenden Zustand.

Instanzen der Klasse `aaa.automaton.FiniteRun` sind dazu da, einem `FiniteAutomaton` ein Wort einzugeben und (schrittweise) zu beobachten, wie sich der Automat beim Lesen dieses Wortes verhält.

Der Konstruktor nimmt als Argumente einen Automaten und eine Eingabe entgegen. Danach kann schrittweise der zu dem eingegebenen Wort korrespondierende Lauf mittels der Methode `step()` durchgeführt werden. Da `step()` `true` zurückliefert, solange noch Zeichen aus der Eingabe zu lesen sind, kann man z. B. einfach mit einer `while`-Schleife einen kompletten Lauf durchführen.

Da ein Lauf auch auf nichtdeterministischen endlichen Automaten stattfinden kann, d. h. auf solchen, die nicht vorher deterministisch gemacht wurden, gibt die Methode `getCurrentStates()` alle möglichen Zustände zurück, in denen der Automat sich nach dem Lesen eines Präfix der Eingabe gerade befinden kann.

Ferner kann man einen Lauf mit `reset` zurücksetzen, oder man kann prüfen, ob der Lauf akzeptierend (`isAcceptingRun()`) ist oder im aktuellen Schritt akzeptierend ist (`isCurrentlyAccepting()`).

3.2.1.7 Operationen

Erst Operationen bewirken, dass man Automaten mächtig und sinnvoll nutzen kann. Ihre Implementierung erfordert aber manchmal größere Mengen von Programmzeilen.

Um hier nicht die Übersicht zu verlieren, wurden alle höheren Operationen in separaten Klassen verwirklicht, die den Charakter von Plugins besitzen. Ein weiterer Vorteil dieser Vorgehensweise ist, dass verschiedene Operationen leicht ausgewechselt werden können und darüber hinaus leicht erweiterbar sind.

Alle Operationen befinden sich in dem Paket `aaa.automaton.operation` und Unterpaketen davon. Aufrufe von Methoden zur Ausführung komplexer Operationen in der Klasse `Automaton` und in ihren Unterklassen werden dorthin delegiert.

Bei der Namensgebung der Automaten gibt es eine Konvention. Eine Methode mit einem Präfix `render` verändert den Automaten, auf den die Methode selber aufgerufen wird. Solche Methoden gibt es für Operationen, die durch lokale Änderungen eines Automaten realisiert werden können (z. B. Komplement, Umkehrung, Totalisierung). Die andere Variante ohne dieses Präfix gibt einen neuen Automaten zurück und belässt den ursprünglichen Automaten unverändert. Diese Variante wird bei allen Operationen benutzt, die es erfordern, einen neuen Automaten aufzubauen (z. B. Determinisierung, Schnitt, Minimierung).

3.2.1.7.1 Komplementbildung Die Bildung des Komplements eines Automaten steht nur für deterministische Automaten zur Verfügung. Die Methode `renderComplementary()` macht aus akzeptierenden Zustände nicht akzeptierende und umgekehrt.

3.2.1.7.2 Vereinigung Die Operation `automaton.union(Automaton)` erlaubt es, zwei `Automaton`-Objekte zu vereinigen. Wenn $L(A)$ die vom Automaten A erkannte Sprache ist und $L(B)$ die von B erkannte Sprache, dann akzeptiert der Automat A.`union`(B) die Sprache $L(A) \cup L(B)$.

3.2.1.7.3 Vereinigung von FiniteAutomaton-Objekten. Zwei `FiniteAutomaton`-Objekte können vereinigt werden, falls sie äquivalente Alphabete benutzen. Die Methode `FiniteAutomaton.union(Automaton)` verändert beide Automaten nicht, sondern gibt ein neues `FiniteAutomaton`-Objekt zurück.

Es wird ein neuer Startzustand erzeugt, vom dem ε-Transitionen auf alle Startzustände der beiden Automaten zeigen. Der vereinigte Automat ist sicher nichtdeterministisch.

3.2.1.7.4 Schnitt Bei der Schnittbildung soll zu Automaten A und B, die Sprachen $L(A)$ und $L(B)$ erkennen, der die Sprache $L(A) \cap L(B)$ erkennende Automat konstruiert werden.

Dies wird von der Methode `intersect()` realisiert. Zu dem Automaten, auf dem diese Methode aufgerufen wird und zu dem als Parameter angegebenen Automaten wird der "parallele Automat" berechnet, der alle auf parallelen Läufen erreichbaren Paare von Zuständen aus den beiden ursprünglichen Automaten enthält. In diesem Automaten ist ein Zustand genau dann akzeptierend, wenn die beiden Zustände aus dem entsprechenden Paar es auch sind.

`intersect` überführt die Automaten, zu denen der Schnittautomat zu berechnen ist, nach

Bedarf in deterministische Automaten. Zurückgeliefert wird ein neues `FiniteAutomaton`-Objekt, der oben beschriebene parallele Automat.

3.2.1.7.5 Umkehrung Die Umkehrung eines Automaten bezeichnet die Operation, die bei einem endlichen Automaten einen endlichen Automaten zurückgibt, der genau die umgekehrte bzw. gespiegelte Sprache des ursprünglichen Automaten akzeptiert. So würde bei der Umkehrung eines Automaten, der das Wort „Hallo“ akzeptiert, ein Automat entstehen, der das Wort „ollaH“ akzeptiert.

Wenn `a` ein endlicher Automat vom Typ `aaa.automaton.FiniteAutomaton` ist, so wird mittels `a.renderReversed()` der Automat umgekehrt.

Intern arbeitet die Umkehrung so, dass alle Transitionen umgedreht werden (dazu besitzen Transitionen eine eigene interne Methode). Danach werden alle Anfangszustände zu akzeptierenden Zuständen und alle akzeptierenden Zustände zu Anfangszuständen.

3.2.1.7.6 Determinisierung Möchte man den zu einem NFA äquivalenten DFA berechnen, so muss der NFA determinisiert werden. Da NFAs in der Regel eine kompaktere Beschreibung regulärer Sprachen ermöglichen, ist dabei mit einer Zunahme der Zustandsanzahl zu rechnen, welche im worst-case exponentiell sein kann. In vielen Fällen beobachtet man jedoch nur ein mildes Wachstum des Automaten.

Die Methode `determinize()` determinisiert einen gegebenen (nichtdeterministischen) endlichen Automaten per Potenzmengenkonstruktion. Im resultierenden DFA entspricht ein Zustand einer Zustandsmenge des NFA. Für alle Wörter w, die einen Zustand s dieses DFA erreichen, ist die Menge der im NFA beim Lesen von w erreichbaren Zustände identisch mit der s entsprechenden Zustandsmenge.

Für genauere Details zu dieser Konstruktion, insbesondere zu dem Problem der Berechnung von Symbolmengen, die mehrere Transitionen gemeinsam schalten, sei auf Abschnitt 3.2.2 auf Seite 122 verwiesen.

3.2.1.7.7 Totalisierung Oft ist es beim Spezifizieren eines Automaten mühselig und auch unübersichtlich, einen Automaten komplett zu spezifizieren. Man stelle sich vor, man möchte lediglich modellieren, dass ein einziger Buchstabe in einem Zustand zu einem gewissen anderen führt. Alle anderen Eingaben sollen dazu führen, dass der Automat nicht akzeptiert.

Man müsste also für das komplette Alphabet einen Übergang von dem Ausgangszustand zu einem nicht akzeptierenden (Ausnahme-)Zustand, der nicht verlassen werden kann,

haben.

Dies ist aber nicht notwendig, da ein Ausnahmezustand immer (implizit) vorhanden ist, für jeden Zustand existiert in der Vorstellung eine Transition dorthin, die von allen Symbolen geschaltet wird, für die keine andere Transition vorhanden ist.

Man möchte allerdings manchmal diese kompakte Form auch expandieren können, um den vollständigen Automaten zu bekommen. Dies ist z. B. beim Komplementieren sogar nötig.

Mittels der Methode `renderTotalized()` lässt sich diese Totalisierung auf einem Automaten durchführen. Ein Aufruf dieser Methode führt dazu, dass der Ausnahmezustand und die zu ihm führenden Transitionen explizit erzeugt und zu der Menge der Zustände bzw. Transitionen hinzugefügt werden.

3.2.1.7.8 Minimierung Oft werden bei Operationen oder bei anderen automatischen Konstruktionen (z. B. bei Analyseverfahren) Automaten berechnet, die nicht minimal sind. Das heißt, dass sie eine größere Anzahl von Zuständen besitzen als der eindeutige minimale DFA für die Sprache, die sie erkennen.

Um solche Automaten so weit wie möglich zu verkleinern, steht die Minimierungsoperation zur Verfügung, die von den Methoden `renderMinimized()`, `renderBrzozowski-Minimized()`, `renderHuffmanMinimized()` und `renderHopcroftMinimized()` implementiert wird.

Es stehen drei Minimierungsverfahren zur Verfügung.

- **Huffman-Minimierung**: Hierbei handelt es sich um das wohl bekannteste Minimierungsverfahren, das – startend bei der Partition, die für jeden Zustand eine Menge enthält – diese schrittweise vergröbert, so dass am Ende alle bezüglich ihres Akzeptanzverhaltens äquivalenten Zustände in einer Menge sind. Aus diesen Mengen werden die Zustände des resultierenden minimalen Automaten konstruiert. Schlüssel und Beschreibung eines Zustands des minimalen Automaten bestehen aus den Indizes der Zustände in der entsprechenden Menge.

  Die Huffman-Minimierung funktioniert nur auf deterministischen Automaten, so dass für NFAs ein preprocessing notwendig ist, bei dem deterministiert wird. Laufzeit und Speicherplatzbedarf der Huffman-Minimierung liegen (bezüglich der Zustandsanzahl des DFA) in $\theta(n^2)$.

- **Brzozowski-Minimierung**: Dieses Verfahren berechnet einen minimalen Automaten, indem der ursprüngliche Automat zweimal hintereinander umgekehrt und deterministiert wird. Für NFAs ist es nicht notwendig, sie vorher zu deterministieren, dies

geschieht implizit während der Minimierung. Daher empfiehlt sich die Brzozowski-Minimierung vor allem für diesen Automatentyp.

Das Verfahren ist bezüglich Rechenzeit und Speicherplatzbedarf heuristisch. Beide sind im worst-case im Verhältnis zur DFA-Zustandsanzahl exponentiell (da determinisiert wird), in vielen Fällen beobachtet man jedoch erstaunlich gute Rechenzeiten.

- **Hopcroft-Minimierung**: Bei diesem Verfahren wird ebenfalls eine Partition der Zustände berechnet, allerdings startet man dabei mit der Partition, die nur aus zwei Mengen besteht, nämlich der Menge akzeptierender und der Menge nicht akzeptierender Zustände. Diese Partition wird schrittweise verfeinert, so dass am Ende die gleiche Partition entsteht wie bei der oben erwähnten Huffman-Minimierung. Es wird aber bei n Zuständen nur Rechenzeit $O(n \cdot \log n)$ benötigt. Die Alphabetgröße bzw. die Anzahl verschiedener Symbolmengen in der Eingabe geht als Konstante in die Laufzeit ein.

3.2.1.8 Weitere Funktionalität

Eine wichtige Anforderung für Bibliotheken ist, dass sie flexibel sind. Dazu gehört, dass sie mit universellen Formaten umgehen können. Eine Interaktion mit schon vorhandenen System ist wünschenswert.

3.2.1.8.1 XML Für viele Anwendungen reicht es nicht aus, dass Automaten nur temporär existieren, es ist häufig notwendig, sie weiterzuverarbeiten oder sie auf einem Sekundärspeicher persistent zu machen.

Für diese beiden Zwecke steht die Möglichkeit zur Verfügung, zu einem Automaten eine XML-Repräsentation zu berechnen. Diese kann (in Form eines `org.w3c.dom.Document`-Objektes) entweder direkt weiterverarbeitet oder als Textdatei gespeichert werden.

Die dabei entstehenden Ergebnisse sollen möglichst redundanzfrei, aber noch von Menschen lesbar sein. Daher wurde keine Standardlösung gewählt, sondern ein eigener Weg beschritten. Die wichtigste Idee dabei ist, Objekte als Teil-XML-Bäume darzustellen, wobei einfache Attribute eines Objekts (z. B. vom Typ String oder int) als Attribute des Wurzelknoten modelliert werden und komplexere member-Objekte bzw. Sammlungen davon als Unterbäume repräsentiert werden. Diese Lösung ist keine allgemeine, sondern wurde nur für Objekte erdacht, die Teile von Automaten sind (Zustände, Transitionen, ...).

Für ein Objekt, das auf diese Weise dargestellt werden soll, muss eine `Converter`-Implementierung zur Verfügung stehen (siehe Abbildung 3.8). `Converter` ist eine ab-

Abbildung 3.8: Die abstrakte Klasse `Converter`

Methoden	Beschreibung
`(abstract) convertObject`	erzeugt einen XML-Teilbaum, der das via `setObject` gesetzte Objekt repräsentiert.
`(abstract) convertElement`	erzeugt ein Objekt aus dem via `setElement` gesetzten XML-Teilbaum.
`(protected) addCollection, addObject`	hängt unter den XML-Knoten `getElement` XML-Teilbäume für mehrere oder ein einzelnes Objekt (durch rekursive Converter-Aufrufe bzw. durch Benutzung der XStream-Bibliothek, die XML-Teilbäume aus beliebigen Objekten erzeugen kann).
`(protected) fetchCollection, fetchObject`	erzeugt mehrere oder ein einzelnes Objekt aus Teilbäumen, die unter `getElement` hängen (intern realisiert durch durch rekursive Converter-Aufrufe oder mit XStream).
`getObject, setObject`	gibt das zu konvertierende Objekt zurück bzw. setzt es.
`getElement, setElement`	gibt die Wurzel des das zu konvertierende Objekt repräsentierenden Teilbaums zurück bzw. setzt sie.
`getAutomaton, setAutomaton`	gibt den Automaten zurück bzw. setzt den Automaten, der das zu konvertierende Objekt enthält.
`getAttribute, setAttribute`	holt oder setzt einfache Attribute.

strakte Klasse, die Methoden anbietet, mit denen Objektattribute gelesen und geschrieben werden können und mit denen member-Objekte in einen Baum eingefügt werden bzw. aus diesem „herausgeholt“ werden können.

Um Objekte einer bestimmten Klasse zu konvertieren, muss eine für diese Klasse spezialisierte Implementierung eines `Converter` zur Verfügung stehen. Solch eine Implementierung muss

- zum Speichern des Objekts als XML-Baum die einfachen Attribute unter dem richtigen Namen per `setAttribute` setzen und die komplexeren *member* des Objekts unter einem eindeutigen Namen per `addObject` bzw. `addCollection` ablegen.
- zum Wiederherstellen des Objekts die Attribute wieder lesen per `getAttribute` und die komplexeren *member* per `fetchObject` bzw. `fetchCollection` einlesen und sie dem wiederherzustellenden Objekt (vorzugsweise per Konstruktoraufruf) zu

übergeben.

Für alle Automatentypen stehen Converter-Implementierungen zur Verfügung. Für die Transformation eines Automaten nach XML und umgekehrt gibt es in der Klasse `AutomatonSerializer` die Methoden `createDocumentFromAutomaton(Automaton)`, `createAutomatonFromDocument(Automaton)`, `writeDocument(Document,String,Writer)`, `readDocument(Reader)`, `saveAutomaton(Automaton,String)`, `loadAutomaton(String)`. Für eine Spezifikation dieser Methoden kann die API-Dokumentation herangezogen werden.

`Document`-Objekte, die aus Automaten erzeugt wurden, können von Instanzen der Klasse `XSLTransformer` weiterverarbeitet werden. Eine Instanz korrespondiert jeweils zu einem xsl-Stylesheet, das Transformationsanweisungen enthält, welche festlegen, wie Knoten oder Teilbäume eines `Document`-Baums in einer resultierenden Textdatei oder einem XML-Dokument darzustellen sind. Als Beispiel steht das Stylesheet `mkdot.xsl` zur Verfügung, das Textdateien im `dot`-Format erzeugen kann. Der gesamte Transformationsprozess von einem Automaten über XML nach `dot` kann durch Aufruf der Methode `AutomatonSerializer.saveAsDot(Automaton,String)` angestoßen werden.

3.2.1.8.2 dk.Brics Die Automatenbibliothek `dk.brics` ist eine offene Automatenbibliothek, die durch ihre Einfachheit und Kompaktheit zu überzeugen weiß. Alle grundlegenden Operationen für endliche Automaten werden angeboten. Daher bietet die AAA-Automatenbibliothek für diese Bibliothek eine Import- und eine Exportfunktion an. Bei der Entwicklung dieser Automatenbibliothek hat es sich bei der Fehleranalyse mehrmals als nützlich erwiesen, die berechneten Ergebnisse von Operationen mit denen von `dk.brics` vergleichen zu können.

3.2.1.8.3 Nutzung Im Paket `aaa.automaton.util` befindet sich eine Klasse `AutomatonUtilities` mit mehreren hilfreichen Funktionen. Die Verwendung der Import- und Exportfunktionen für `dk.brics` ist denkbar einfach gehalten.

Mittels eines einfachen statischen Imports `import static aaa.automaton.util.AutomatonUtilities.*` lassen sich die Funktionen sofort nutzen:

- `aaa.automaton.FiniteAutomaton importBric(dk.brics.automaton.Automaton);`
- `dk.brics.automaton.Automaton exportToBrics(aaa.automaton.FiniteAutomaton);`

dienen dazu, ein Automatenobjekt aus der einen Bibliothek in ein Automatenobjekt der anderen umzuwandeln.

3.2.1.8.4 Hilfsfunktionen In der Klasse `aaa.automaton.AutomatonUtilities` befinden sich eine Reihe nützlicher statischer Methoden, die bisher noch keinerlei Erwähnung fanden. Diese dienen dazu, häufig benötigte und gängige Automaten zu erzeugen. So gibt es eine Methode `createAutomatonForWord(String)` für die Erstellung von Automaten, die genau ein bestimmtes Wort erkennen, das mittels eines `String` ausgedrückt wird. Andere gängige Automaten sind der Automat, der alles akzeptiert, derjenige, der nichts akzeptiert und schließlich der, der nur das leere Wort akzeptiert.

Zuletzt sei noch erwähnt, dass die Möglichkeit besteht, einen Automaten aus einem regulären Ausdruck zu konstruieren. Intern wird dabei die entsprechende Funktionalität in der Bibliothek `dk.brics` benutzt und der resultierende Automat importiert (s. o). Dies leistet die Methode `fromRegExp(String)`. Diese Methode nutzt die Import- und Exportmöglichkeiten von `dk.brics` und kann daher nur endliche Automaten konstruieren. Es gibt jedoch auch einen eigenen Parser für reguläre Ausdrücke, der auch die Konstruktion von Büchiautomaten aus ω-regulären Ausdrücken erlaubt. Zu diesem Thema sei auf Abschnitt 3.3.4 auf Seite 157 verwiesen.

3.2.2 Berechnung von Transitionen bei Automatenoperationen

Bei der Implementierung von Operationen auf endlichen Automaten sind Ergebnisautomaten zu konstruieren, für die Transitionen berechnet werden müssen. Beispielhaft sei die Potenzmengenkonstruktion genannt, bei der die Zustände des deterministischen Ergebnisautomaten Mengen von Zuständen des NFAs entsprechen. Sei $(Q, \Sigma, q_0, F, \delta : Q \times \Sigma \longrightarrow 2^Q)$ ein NFA und $(2^Q, \Sigma, \{q_0\}, \delta' : 2^Q \times \Sigma \longrightarrow 2^Q)$ der dazu äquivalente DFA. δ' ist wie folgt definiert: $\delta'(S, a) = \bigcup_{s \in S} \delta(s, a)$.

Liegen die Transitionen in tabellarischer Form vor, stellt die Berechnung von δ' kein Problem dar, man muss lediglich für alle $s \in S$ an der Stelle für a nachschauen und die entsprechende Menge bilden. Die Laufzeit ist linear in der Eingabelänge.

In der vorliegenden Automatenbibliothek sind die transitionsbewirkenden Zeichen mengenweise spezifiziert. Natürlich ist es auch hier möglich, für jedes einzelne Zeichen zu prüfen, in welchen Mengen es liegt. Die Laufzeit eines solchen Verfahrens wäre stets mindestens so groß wie die Größe des Alphabets. Wünschenswert ist aber eine Laufzeit, die asymptotisch nicht viel größer ist als die Länge der Darstellung der Symbolmengen in der Ein- bzw. Ausgabe.

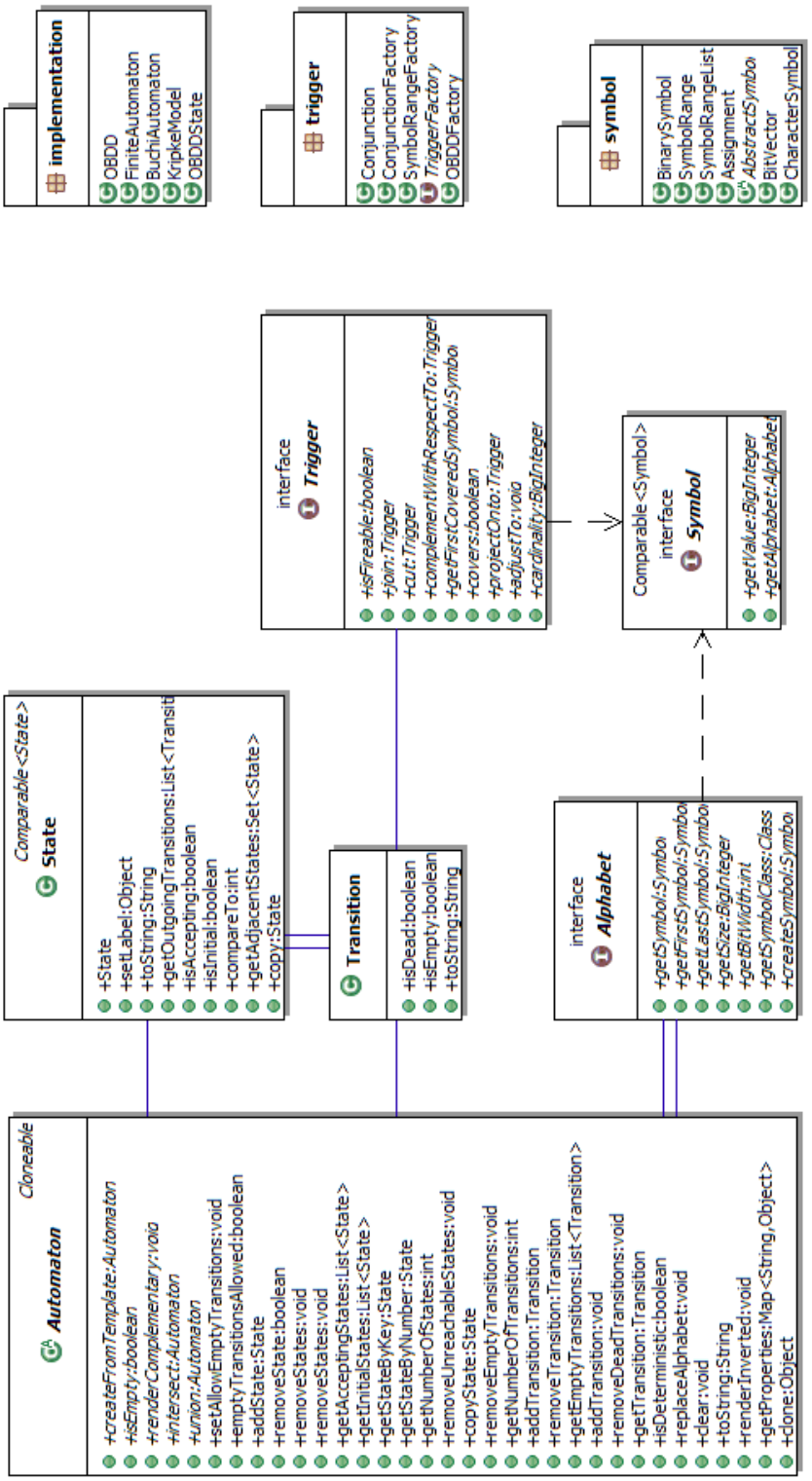


Abbildung 3.9: Diagramm zu dem endgültigen Automatenkern

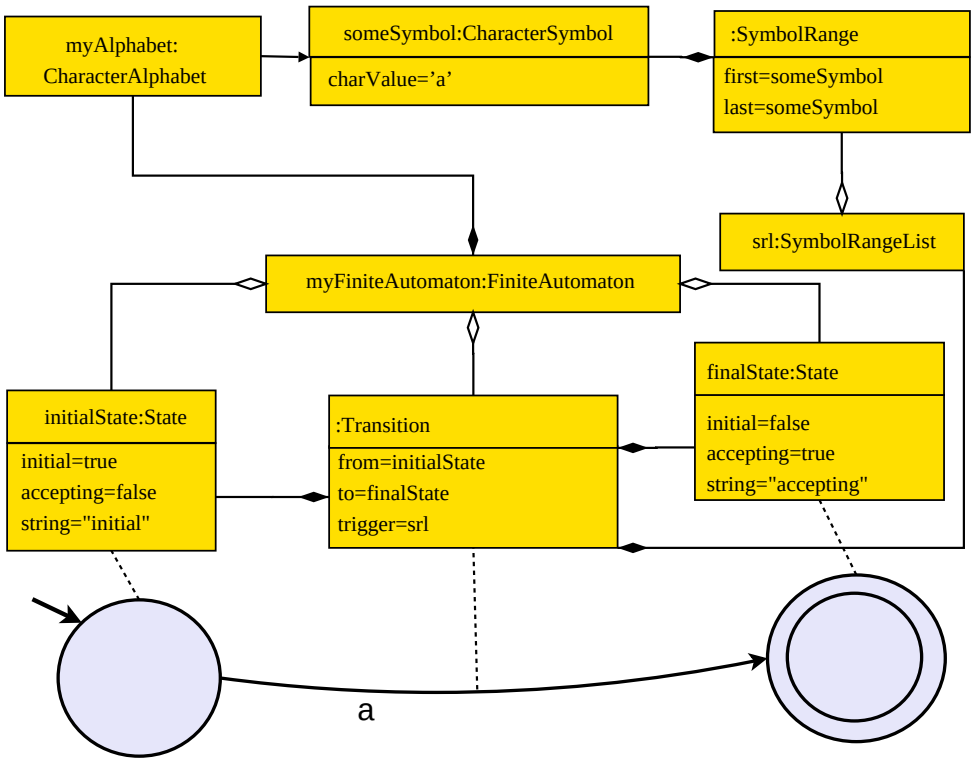


Abbildung 3.10: Objektdiagramm zur beispielhaften Konstruktion in 3.2.1.5

Die Ausgabe besteht aus neu berechneten Transitionen mit Symbolmengen als Markierung, bei der Potenzmengenkonstruktion sind das Transitionen zwischen Zustandsmengen, z. B. zwischen A und B. Eine Transition von A nach B kann ausgelöst werden von allen Zeichen im Schnitt der Symbolmengen, mit denen die Transitionen markiert sind, die im Originalautomaten in A beginnen und die einen Zustand in B erreichen. In den Symbolmengen der übrigen in A startenden Transitionen darf kein Zeichen vorkommen, das sich in diesem Schnitt befindet.

Es ist also algorithmisch zu analysieren, wie solche Schnittmengen aussehen, und es sind entsprechende Transitionen auszugeben. Die abstrakte Klasse `TransitionAnalyzer` stellt eine allgemeine Schnittstelle für solche Transitionsanalysen zur Verfügung. Die Eingabe ist eine Menge von Transitionen, die Ausgabe ist formatiert als Liste von Objekten des Typs `TransitionAnalyzer.SharedTransition`. Ein solches Objekt referenziert eine Menge M von Startzuständen, eine Menge M' von erreichten Zuständen und ein `Trigger`-Objekt, das genau die gemeinsamen Zeichen der Transitionen von M nach M' enthält.

Eine mögliche Darstellung von Symbolmengen sind Intervalle (`SymbolRanges`). In Abbildung 3.11 ist ein einfaches Beispiel dargestellt. Hier sind die in $S = \{q_0, q_2\}$ startenden Transitionen des Potenzmengenautomaten zu berechnen. Von q_0 startet eine Transition nach q_1, von q_2 starten Transitionen nach q_3 und q_4. Die Transitionen sind als Linien dargestellt, auf denen graue Rechtecke liegen, die den Symbolintervallen entsprechen, die

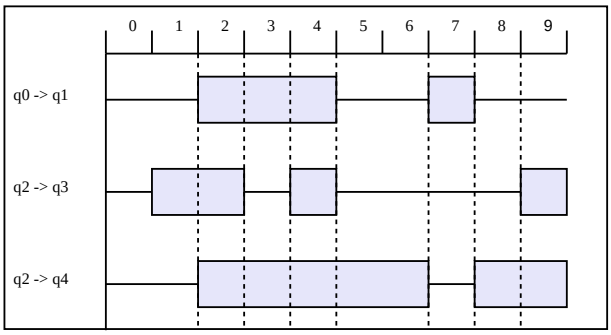


Abbildung 3.11: Situation bei der Potenzmengenkonstruktion

die Transitionen schalten.

Das Problem kann durch schrittweises Identifizieren von jeweils einem Symbolintervall I und einer zugehörigen Transitionsmenge $\mathcal{T}$ mit den folgenden Eigenschaften gelöst werden.

a) Alle in $\mathcal{T}$ enthaltenen Transitionen werden von allen Zeichen in I geschaltet.

b) Alle von S ausgehenden Transitionen, die nicht in $\mathcal{T}$ enthalten sind, werden von keinem Zeichen in I geschaltet.

c) I ist maximal, das heißt würde man I um ein benachbartes Zeichen $a \in \Sigma$ erweitern, wäre eine der ersten beiden Eigenschaften nicht mehr gegeben.

In dem vorliegenden Beispiel wären $(I = [1,1], \mathcal{T} = \{q_2 \rightarrow q_3\})$ und $(I = [2,2], \mathcal{T} = \{q_0 \rightarrow q_1, q_2 \rightarrow q_3, q_2 \rightarrow q_4\})$ geeignete Paare von Symbolintervall und Transitionsmenge.

Hat man solch ein Intervall I und eine Menge $\mathcal{T}$ gefunden, kann eine Transition konstruiert werden, die von I geschaltet wird und die die Menge von Zuständen erreicht, die von Transitionen in $\mathcal{T}$ erreicht werden. Die beiden erstgenannten Eigenschaften sichern, dass genau die richtigen in S startenden Transitionen berücksichtigt werden. Die dritte Eigenschaft sichert, dass nicht mehr Transitionen als nötig konstruiert werden.

Folgender Algorithmus berechnet Paare $(I, \mathcal{T})$ in der gewünschten Weise.

1. Initialisiere für jede betrachtete Transition einen Zeiger, der anfangs auf das erste Zeichen des ersten Symbolintervalls der Transition zeigt. $\mathit{Position}(z)$ gibt die aktuelle Position, $\mathit{Transition}(z)$ die zugehörige Transition und $\mathit{Intervallende}(z)$ das Ende des aktuellen Symbolintervalls für solch einen Zeiger z an. $\mathit{Erreicht}(z)$ ist der Zustand, der von der zu z gehörigen Transition erreicht wird.

2. Initialisiere einen MIN-HEAP h, in dem die Zeiger gemäß ihrer Position aufsteigend geordnet sind.

3. Entferne alle minimalen Zeiger aus h, nenne ihre Position *minSymbol* und die entsprechende Zeigermenge Z.

4. - Falls h nicht leer ist, nenne das nunmehr minimale Element in h (Kopf des MIN-HEAP) z', setze $maxSymbol := min\{Intervallende(z) \mid z \in Z\} \cup \{Position(z') - 1\}$.
   - Falls h leer, setze $maxSymbol := min\{Intervallende(z) \mid z \in Z\}$.

5. Gebe aus $I = [minSymbol, maxSymbol]$, $\mathcal{T} = \{Transition(z) \mid z \in Z\}$.

6. Versuche, alle Zeiger in Z zur Position $maxSymbol+1$ zu bewegen. Ein z kann genau dorthin bewegt werden, wenn $maxSymbol+1$ noch auf dem aktuellen Intervall liegt, ansonsten wird z zum Anfang des nächsten Intervalls bewegt. Existiert kein nächstes Intervall mehr, kann z nicht bewegt werden und wird nicht wieder in h eingefügt. Falls z erfolgreich bewegt wurde, füge z wieder in h ein.

7. Falls h nicht leer, fahre bei 3. fort.

Die ausgegebenen Paare $(I, \mathcal{T})$ haben die gewünschten Eigenschaften:

a) I endet spätestens dort, wo das erste der aktuellen Intervalle in $\mathcal{T}$ endet, nämlich an $min\{Intervallende(z) \mid z \in Z\}$.

b) Weiterhin endet I spätestens an der Position, die vor derjenigen liegt, an der ggf. ein Intervall einer Transition beginnt, die nicht in $\mathcal{T}$ ist. Denn in die Minimumsberechnung geht auch $Position(z') - 1$ ein, falls ein Zeiger z' für eine weitere Transition existiert.

c) I kann nicht in Richtung höherwertiger Zeichen verlängert werden, denn ansonsten wäre das Minimum anders berechnet werden. I kann auch nicht in Richtung niederwertigerer Zeichen verlängert werden, denn ansonsten hätte in der vorausgehenden Iteration das gleiche $\mathcal{T}$ ausgegeben werden müssen, und das Minimum wäre anders berechnet worden.

Die Anzahl der Iterationen entspricht der Anzahl ausgegebener `SymbolRanges`. Diese lassen sich nicht zu größeren Intervallen zusammenfassen. Also ist die Ausgabe optimal formatiert. Die Kosten einer Iteration werden dominiert durch die Kosten der MINHEAP-Operationen, sind damit logarithmisch bezüglich der Anzahl betrachteter Transitionen; also höchstens logarithmisch bezüglich der Ausgabelänge. Damit erfüllt die Laufzeit des Algorithmus die oben genannten Anforderungen.

Der Algorithmus ist in der Klasse `aaa.automaton.util.SymbolRange-TransitionAnalyzer` implementiert.

3.2.3 Einleitung und Übersicht zur Weiterführung

Die abstrakte Klasse `Automaton` wurde als gemeinsame Basis für verschiedene Automatenmodelle konzipiert. Die wichtigste und zugleich auch einfachste Klasse eines Automaten ist der endliche Automat. Dieser wurde bereits in der ersten Hälfte der Projektgruppe in der Klasse `FiniteAutomaton` vollständig realisiert. Als weiteres Modell sollte nun der Büchiautomat realisiert werden. Da der Büchiautomat noch immer ein sehr verwandtes Modell ist, ist die Implementierung nicht aufwändig. Durch das Ableiten von `Automaton` stehen sofort die grundlegenden Automatenfunktionalitäten zur Verfügung. Jedoch geht das Modell etwas weiter und Operationen müssen anders implementiert werden und als eine Erweiterung wird noch das Modell der generalisierten Büchiautomaten bedacht. Der Kern bleibt aber gleich.

Dass Büchiautomaten als Automatenmodell gewählt wurden, geschah mit der Absicht, mit diesen Automaten *Model Checking* zu betreiben, und zwar Model Checking mit LTL-Formeln. Es war daher auch nötig, *Kripkestrukturen* zur Verfügung zu haben, um darauf als elementare Struktur operieren zu können. Eine Kripkestruktur kann als eingeschränkter Automat aufgefasst werden, und somit liegt es nahe, dass auch diese Datenstruktur auf `Automaton` fußen kann.

Insgesamt wurde also auf dem Gerüst des offenen Automatenbaukastens, der vorhanden war, neben der Referenzimplementierung eines endlichen Automaten mit sämtlichen Funktionen ein weiteres Automatenmodell hinzugefügt. Zusätzlich wurden zwei weitere Modelle auf einen Nenner gebracht, da man diese auf ein Automatenmodell reduzieren und ebenfalls mithilfe dieses Baukastens erstellen kann. Dies hat mehrere angenehme Nebeneffekte mit sich gezogen. So kann beispielsweise das GUI-Frontend (der *AAA Workspace*) von diesem gemeinsamen Modell profitieren, da die gemeinsame Darstellung dazu führt, dass man nur Typen von Automaten auswechseln muss und sich Funktionen übertragen. Dies ergänzt sich gut zu dem pluginorientierten Konzept des GUI Frontends.

Abschließend lässt sich sagen, dass sich die Nutzung des endlichen Automaten `FiniteAutomaton` nur wenig geändert hat. Es hat sich sogar herausgestellt, dass sich die Bedienung durch neu eingeführte Konstrukte vereinfachen ließ. Somit stellt die AAA-Automatenbibliothek nicht nur eine sehr umfangreiche und leistungsfähige Klassenbibliothek dar, sondern kann darüber hinaus auch intuitiv und leicht benutzt werden, wenn man davon absieht, dass eine riesige Fülle an Klassen und Methoden zur Verfügung steht. In den folgenden Abschnitten werden die im zweiten Semester vorgenommenen Umstrukturierungen der Automatenbibliothek und die Realisierung der neu hinzugekommenen Automatenmodelle Büchiautomaten, OBDDs und Kripkestrukuren [8] beschrieben.

[8] Kripkestrukturen sind streng genommen keine Automaten.

3.2.3.1 Überholung des Transitions- und Alphabetkonzept

Nach der ersten Projektphase standen Automaten zur Verfügung, die reguläre Sprachen über beliebigen Alphabeten darstellen können. Diese Sprachen können nicht nur aus Wörtern von Bits oder druckbaren (Unicode-)Zeichen bestehen, sondern auch aus Wörtern von Bitvektoren, wie es bereits bei der Analyse der Presburger Arithmetik der Fall war.

Um die Menge von Bitvektoren darzustellen, die eine Transition zwischen zwei Zuständen auslösen können, war es zwingend notwendig, diese Menge in Intervalle zu zerlegen und die Transition für jedes dieser Intervalle mit einer entsprechenden `SymbolRange` zu markieren.

In der zweiten Projektphase sollen Automaten dazu benutzt werden, (unerwünschte) Berechnungspfade von Transitionssystemen darzustellen (vgl. dazu Abschnitt 3.1.3 auf Seite 86). Die Transitionen solcher Automaten sind mit Mengen von Mengen atomarer Propositionen markiert, die intern als Mengen von Bitvektoren dargestellt werden sollen. Es ist nicht zu erwarten, dass die dabei auftretenden Bitvektormengen sich in wenige Intervalle zerlegen lassen. Angemessener ist es, eine solche Menge durch ihre charakteristische Funktion anzugeben. Zur Darstellung von Booleschen Funktionen haben sich `ordered binary decision diagrams` (OBDDs) bewährt – vgl. dazu Abschnitt 3.2.4 auf Seite 131. Diese Datenstruktur wurde in der zweiten Projektphase als alternative Darstellung von Symbolmengen implementiert.

Diese Darstellung ist der alten – intervallorientierten – *nicht* diametral entgegengesetzt. Vielmehr erkennt man Gemeinsamkeiten. In beiden Fällen werden Mengen dargestellt, die sich vereinigen, schneiden und komplementieren lassen. Bei OBDDs wird das durch Synthesen (vgl. 3.2.4.3 auf Seite 134) bzw. Komplementbildungen (vgl. 3.2.4.4 auf Seite 134) realisiert, bei `SymbolRanges` erreicht man dies durch entsprechende Intervalloperationen.

Beide Darstellungen können beliebige Teilmengen von Alphabeten repräsentieren. Denn jedes Element eines Alphabets, also jedes Symbol, korrespondiert zu einem `BigInteger`-Wert, der als Bitvektor interpretiert werden kann. Man kann Mengen solcher Zahlen zu Intervallen (`SymbolRanges`) zusammenfassen. Ebenso kann man eine charakteristische Funktion finden, deren erfüllende Belegungen genau zu den in einer Menge enthaltenen Bitvektoren korrespondieren, und diese Funktion kann als OBDD dargestellt werden.

OBDDs sind eher geeignet, wenn solche Mengen groß sind und sie eine einfache charakteristische Funktion haben. `SymbolRanges` eignen sich, wenn die Mengen typischerweise nur aus einem oder wenigen Zeichen bestehen. Dann genügen wenige Intervalle, während ein OBDD in diesem Fall für (fast) jedes vorkommende Bit einen Knoten benötigt.

Es gibt Symbolmengen, die sich mit einem OBDD konstanter Größe darstellen lassen, die

aber in exponentiell viele (bzgl. der Variablenanzahl) Intervalle zerfallen. Man denke an die Menge, die einen Bitvektor genau dann enthält, wenn sein am wenigsten signifikantes Bit gesetzt ist. Umgekehrt gibt es aber keine Menge, die sich mit Hilfe von Intervallen wesentlich effizienter darstellen lässt als mit einem OBDD. Denn die charakteristische Funktion einer durch Intervalle beschriebenen Menge ist die Disjunktion von Konjunktionen jeweils zweier Vergleichsfunktionen ($x \geq$ Intervallanfang $\wedge\ x \leq$ Intervallenende). Bei einer geeigneten Variablenordnung ist diese Funktion durch ein OBDD darstellbar, das höchstens um einen linearen Faktor (bzgl. der Anzahl der Bits) größer ist als die Anzahl der Intervalle.

3.2.3.2 Trigger

Die bereits vorhandenen Realisierungen von Alphabeten und Symbolen können weitgehend unverändert aus der ersten Projektphase übernommen werden. Für Klassen, die Symbolmengen darstellen, wurde das Interface `Trigger` eingeführt, das alle bei der Realisierung von Automatenoperationen benötigten Manipulationen von Symbolmengen ermöglicht.

Die Klassen `OBDD` und `SymbolRangeList` (repräsentiert eine Menge von Symbolintervallen, also `SymbolRanges`) implementieren dieses Interface. Die Signatur enthält die oben genannten Mengenoperationen (`join()`, `cut()`, `complementWithRespectTo()`). Außerdem soll eine `Trigger`-Implementierung prüfen können, ob ein bestimmtes Symbol in der dargestellten Symbolmenge enthalten ist (`covers()`), es soll auch ein beliebiges enthaltenes Zeichen geliefert werden können (`getCoveredSymbol()`). Eine weitere wichtige Methode ist `isFireable()`, mit der entschieden werden kann, ob eine Symbolmenge nicht leer ist, also eine damit markierte Transition „gefeuert“ werden kann.

3.2.3.2.1 `TriggerFactory` Manchmal kommt es vor, dass Standard-Trigger (z. B. leere Symbolmenge, vollständige Symbolmenge) erzeugt werden sollen, oder dass zu Standard-Darstellungen korrespondierende Trigger benötigt werden, ohne dass man sich darum kümmern möchte, welche konkrete Darstellung verwendet wird, z. B. wenn man eine Symbolmenge erzeugen möchte, die nur aus einem einzigen Symbol besteht. Daher gehört zu jeder `Trigger`-Implementierung auch die Implementierung einer `TriggerFactory` mit den Methoden `createEmptyTrigger(), createFullTrigger(), createSingleSymbolTrigger()`. Jeder Automat ist mit einer seinem Triggertyp entsprechenden `TriggerFactory` assoziiert. Beim Instanziieren eines Automaten wird der zu benutzende Triggertyp angegeben, so dass im Konstruktor die passende `TriggerFactory` ausgewählt werden kann.

3.2.3.2.2 Transitionanalysen mit OBDDs Die bereits genannten Methoden des Interface `Trigger` reichen aus, um viele Probleme zu lösen, z. B. kann damit ein Produktautomat berechnet werden – dabei wird jeweils per `Trigger.cut()` geprüft, ob zwei Transitionen von gemeinsamen Symbolen geschaltet werden. Falls ja, ist eine mit dem Rückgabewert von `Trigger.cut()` markierte Transition des Produktautomaten zu konstruieren. Auch kann überprüft werden, ob ein Automat deterministisch ist. Dabei werden die `Trigger` der von einem Zustand ausgehenden Transitionen paarweise geschnitten. Falls einer dieser Schnitte nicht leer ist, ist der Automat nicht deterministisch. Für manche Operationen, insbesondere die Determinisierung, ist es sinnvoll, noch weitere Funktionen zur Untersuchung von Transitionen zur Verfügung zu haben. In Abschnitt 3.2.2 auf Seite 122 wurde daher der `TransitionAnalyzer` eingeführt.

Auch für OBDDs existiert eine Implementierung eines `TransitionAnalyzer`, die jedoch weniger effizient ist, da nun die Zeichenmengen nicht mehr als geordnete Intervalle vorliegen, über die man iterieren und in denen man nach lokalen Überschneidungen suchen kann.

Um herauszufinden, in welcher Weise sich bei einer Potenzmengenkonstruktion die Symbolmengen von m in einem Zustand startenden Transitionen überschneiden, wird eine m-äre Synthese der m OBDDs, mit denen die Transitionen markiert sind, durchgeführt. Das ist eine Verallgemeinerung der in Abschnitt 3.2.4.2 beschriebenen binären Synthese von OBDDs. Dabei entsteht ein Zwischen-OBDD O_Z, dessen Senken allerdings nicht mit 0 oder 1 markiert werden. Die Senken von O_Z heißen im Folgenden „Pseudo-Senken“.

Die Synthese ähnelt m parallelen Läufen durch die m OBDDs, und wann immer dabei m Senken erreicht werden, konstruiert man eine „Pseudo-Senke“ von O_Z. Diese Senke wird markiert mit einem Bitvektor der Länge m, in dem der i-te Eintrag angibt, welche Senke des i-ten OBDDs erreicht wurde. Alle „Pseudo-Senken“, die mit dem gleichen Bitvektor markiert sind, werden ähnlich wie bei der normalen Synthese „verschmolzen“.

Nach der Konstruktion von O_Z wird für jede „Pseudo-Senke“ P, in deren zugehörigem Bitvektor mindestens ein Bit gesetzt ist, ein neues OBDD O^P erzeugt, das eine Kopie von O_Z ist. Dabei wird P 1-Senke von O^P. Alle anderen „Pseudo-Senken“ werden zur 0-Senke von O^P verschmolzen.

Sei $\mathbf{O_1}$ die Menge der OBDDs, die zu den 1-Senken in P korrespondieren und sei $\mathbf{O_0}$ die Menge der übrigen OBDDs. Die von O^P dargestellte Menge enthält genau die Symbole, die in allen von den $\mathbf{O_1}$-OBDDs dargestellen Mengen enthalten sind, und in keiner der Mengen, die von den $\mathbf{O_0}$-OBDDs dargestellt werden. Es kann also eine Transition des Potenzmengenautomaten konstruiert werden, die die Menge der Zustände erreicht, die Ziele der Transitionen sind, die mit den OBDDs in $\mathbf{O_1}$ markiert sind.

3.2.4 OBDDs

In Abschnitt 3.2.3.1 auf Seite 128 wurde beschrieben, warum OBDDs zur Darstellung von Symbolmengen implementiert wurden.

OBDDs sind eine in der Schaltkreisverifikation häufig gebrauchte Datenstruktur. Sie waren in den letzten 20 Jahren, und sind es noch heute, Gegenstand der theoretischen Forschung, die eine Vielzahl von Ergebnissen zutage gefördert hat. Insbesondere sind effiziente Algorithmen für die üblichen Mengenoperationen auf OBDDs bekannt.

Zunächst soll eine formale Definition von OBDDs vorgenommen werden[9]:

Definition 1. *Ein π-**OBDD** zu einer Variablenordnung $\pi = (x_1, \ldots, x_n)$ ist ein azyklischer, gerichteter Graph $G = (V, E)$ mit einer Quelle.*

- *Der Graph enthält nur innere Konten mit Ausgangsgrad 2 und Senken mit Ausgangsgrad 0.*
- *Innere Knoten sind mit einer Variable x_i markiert.*
- *Die beiden ausgehenden Kanten eines inneren Knotens sind mit 0 bzw. 1 markiert.*
- *Senken sind mit 0 oder 1 markiert*
- *Auf jedem Pfad kommt eine Variable x_i höchstens einmal vor.*
- *Wenn auf einem Pfad die Variable x_i mit der Ordnungsnummer $\pi(i)$ vorkommt, dann folgen auf dem Pfad nur Variablen x_j deren Ordnung bzgl. der Variablenordnungen größer sind, d. h. es gilt $\pi(i) < \pi(j)$.*

Ein Problem von OBDDs ist, dass ihre Größe entscheidend von der gewählten Variablenordnung abhängt. Die Wahl der Variablenordnung kann zwischen polynomieller und exponentieller Größe des OBDDs entscheiden. Auf der anderen Seite ist es ein NP-schwieriges Problem, für eine als Formel oder als OBDD gegebene Funktion f die optimale Variablenordnung zu berechnen. Es gibt allerdings Heuristiken, mit denen sich zumeist „gute“ Ordnungen berechnen lassen.

Bei der Implementierung der OBDD-Datenstruktur haben wir uns auf die feste kanonische Variablenordnung beschränkt, da die Synthese zweier OBDDs die gleiche Variablenordnung voraussetzt und das Vertauschen zweier Variablen x_i und x_j in der Ordnung alleine schon quadratische Rechenzeit (bzgl. der OBDD-Größe) benötigt. Es wurden allerdings

[9] Die Definition folgt dem Skript der Vorlesung „Theorie des Schaltkreisentwurf und der Schaltkreisverifikation“ von Beate Bollig [**?**]

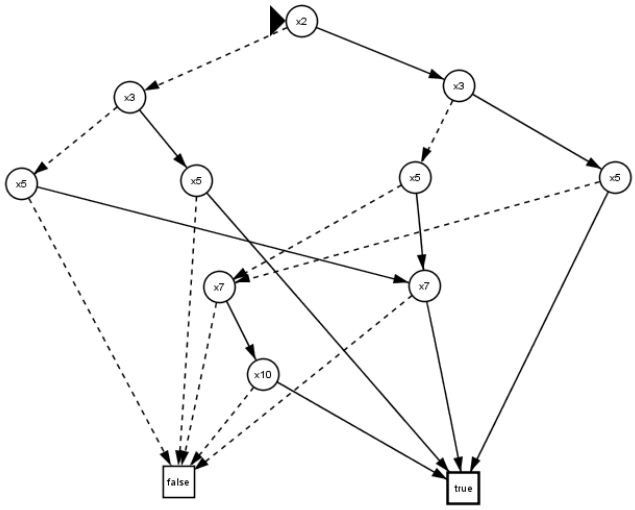


Abbildung 3.12: Ein OBDD für die Funktion $f = (x_2 \wedge x_7 \wedge x_{10}) \vee (x_5 \wedge (x_3 \vee x_7))$

Vorkehrungen getroffen, die es ermöglichen, auch OBDDs mit einer anderen Variablenordnung als der kanonischen zu konstruieren.

3.2.4.1 Reduzierung

Es gibt in der Bibliothek keine echte Minimierung für OBDDs, nur eine Reduzierung. Dazu benutzen wir den Algorithmus von Bryant, der im Skript von Beate Bollig ([**?**] S. 15-18) detailliert beschrieben ist. Dieser Algorithmus benutzt zwei verschiedene Regeln: die *Deletion Rule*, die besagt, dass ein Knoten eliminiert werden kann, falls 0- und 1-Nachfolger identisch sind; und die *Merging Rule*, nach der zwei Knoten „verschmolzen“ werden können, falls sie mit der gleichen Variable markiert sind und die 1-Nachfolger sowie die 0-Nachfolger beider Knoten identisch sind.

Für jeden Knoten wird ein Repräsentant berechnet; zu Beginn des Verfahrens ist jeder Knoten sein eigener Repräsentant. Sollte es im weiteren Verlauf zu einer Anwendung der *Deletion Rule* kommen, so wird der Repräsentant des Nachfolgers des eliminierten Knotens zu dessen Repräsentanten. Bei einer Anwendung der *Merging Rule* wird einer der beiden Repräsentanten der betroffenen Knoten Repräsentant beider Knoten.

Da es stets nur eine 0-Senke und eine 1-Senke gibt, müssen diese bei einer Reduzierung nicht behandelt werden. Für die übrigen Knoten wird zunächst bottom-up überprüft, ob die *Deletion Rule* anwendbar ist. Danach wird für jede Variable x_i eine Liste aufgebaut, die alle Knoten enthält, die mit x_i markiert sind. Die Knoten werden in der Liste gemäß ihrer Nachfolger sortiert, so dass Knoten, auf die die *Merging Rule* anwendbar ist, in der Liste benachbart sind. Es genügt, die sortierte Liste einmal zu durchlaufen, um alle

Verschmelzungen von mit x_i markierten Knoten durchzuführen.

Nachdem alle Repräsentanten berechnet wurden, wird ein neues OBDD aufgebaut, das für jeden Repräsentanten nur einen Knoten enthält.

Die Reduzierung wird in der Automatenbibliothek durch die Klasse `aaa.automaton.operation.obdd.Reducer` realisiert und kann durch die Methode `reduce()` in der Klasse `aaa.automaton.implementation.obdd` aufgerufen werden.

3.2.4.2 Synthese

Die Synthese-Operation ermöglicht es, zwei OBDDs durch einen Booleschen Operator zu verknüpfen. Seien G_a und G_b zwei OBDDs, die mit dem Operator $\otimes$ verknüpft werden sollen. Die Idee ist nun folgende: für eine Eingabe $x \in \{0,1\}^n$ werden G_a und G_b parallel durchlaufen und (an den Senken) die beiden Funktionswerte $a(x)$ und $b(x)$ durch den Operator $\otimes$ verknüpft. Es wird davon ausgegangen, dass die beiden OBDDs die gleiche Variablenordnung $x_1, \ldots, x_n$ besitzen, wobei anzumerken ist, dass nicht notwendigerweise in beiden jede Variable auf allen Pfaden getestet werden muss. Die Berechnung startet an den Quellen, und man kann sich vorstellen, dass ein Zeiger auf den aktuellen Knoten v_a bzw. v_b der beiden OBDDs gerichtet ist. Während der Berechnung wird das OBDD $G_{a \otimes b}$ erzeugt, dessen Knoten mit (v_a, v_b) bezeichnet werden. Es können sechs Fälle auftreten:

1. Fall: v_a und v_b sind beide mit der Variable x_i markiert. Es wird der Knoten (v_a, v_b) erzeugt und mit der Variable x_i markiert. Der 0-Nachfolger ist der Knoten (v_{a_0}, v_{b_0}) wobei v_{a0} der 0-Nachfolger von Knoten v_a ist. Der 1-Nachfolger von (v_a, v_b) ist analog (v_{a_1}, v_{b_1}). Die Berechnung wird dann rekursiv an den beiden Nachfolgern fortgesetzt.

2. Fall: v_a ist mit der Variable x_i und v_b mit x_j markiert, wobei $i < j$ ist. Die Variable x_i steht also in der Ordnung vor x_j; das OBDD G_b muss auf G_a warten, der Zeiger bleibt also auf v_b ruhen, während er in G_b den 0- bzw. 1-Nachfolger folgen muss. Es wird der Knoten (v_a, v_b) erzeugt und mit x_i markiert. Sein 0-Nachfolger wird (v_{a_0}, v_b) und sein 1-Nachfolger wird (v_{a_1}, v_b) (G_b muss warten).

3. Fall: v_a ist mit x_i und v_b mit x_j markiert, wobei $i > j$ ist. Der Fall ist analog zu Fall 2. Es wird der Knoten (v_a, v_b) erzeugt und mit x_j markiert. Sein 0-Nachfolger wird (v_a, v_{b_0}) und sein 1-Nachfolger wird (v_a, v_{b_1}) (G_a muss warten).

4. Fall: v_a ist eine Senke und v_b ist mit x_i markiert. Auch hier muss das OBDD G_a auf G_b warten. Es wird der Knoten (v_a, v_b) erzeugt und mit x_i markiert. Sein 0-Nachfolger wird (v_a, v_{b_0}), sein 1-Nachfolger (v_a, v_{b_1}).

5. Fall: v_a ist mit x_i markiert und v_b ist eine Senke. Der Fall ist analog zu Fall 4. Es wird der Knoten (v_a, v_b) erzeugt und mit x_i markiert. Sein 0-Nachfolger wird (v_{a_0}, v_b) und sein 1-Nachfolger wird (v_{a_1}, v_b).

6. Fall: v_a und v_b sind Senken. Es wird die Senke (v_a, v_b) erzeugt. Die Markierung der neuen Senke ergibt sich aus der Markierung von v_a und v_b verknüpft mit $\otimes$.

Das entstandene OBDD $G_{a\otimes b}$ ist nach dieser Berechnung in der Regel nicht minimal bzgl. der Variablenordnung. Der Algorithmus von Bryant (1986) vermeidet die Erzeugung von überflüssigen Zuständen und berechnet das reduzierte OBDD $G_{a\otimes b}$, sofern die OBDDs G_a und G_b reduziert sind. Die Rechenzeit für die Syntheseoperation ist $O(|G_a| \cdot |G_b|)$.

In der Automatenbibliothek ist die Synthese-Operation durch die Klasse `aaa.automaton.operation.obdd.Synthesizer` realisiert. Einem Synthesizer-Objekt werden mit der Methode `Synthesizer.setup(OBDD obdd1, OBDD obdd2, Byte operation)` die beiden OBDDs und die entsprechende Operation als `Byte` übergeben. Für die gängigen Operationen stehen in der Klasse `OBDD` entsprechende Konstanten zur Verfügung; für die AND-Operation wäre der `Byte`-Wert beispielsweise $8_{10} = 1000_2$, für OR wäre er $14_{10} = 1110_2$ gemäß den Funktionstabellen der AND und OR Funktion.

3.2.4.3 Vereinigung und Schnitt via Synthese

Das logische Äquivalent der Vereinigung bzw. des Schnitts ist das OR bzw. AND. Somit wird die Vereinigung der beiden OBDDs `obdd1` und `obdd2` durch die Synthese beider realisiert. Der Aufruf `obdd1.union(obdd2)` liefert ein OBDD für die Vereinigung zurück; analog der Aufruf `obdd1.intersect(obdd2)` für den Schnitt.

3.2.4.4 Komplement

Wenn man ein OBDD G_f als eine kompakte Darstellung der Menge der erfüllenden Belegungen für die Formel f auffasst, dann ist das Komplement $\overline{G_f}$ die Menge der nicht-erfüllenden Belegungen von f (oder die Formel $\neg f$). Das Komplement von einem `OBDD` Objekt wird durch `OBDD.renderComplementary()` berechnet. Dabei wird das Objekt wie bei allen `render`-Methoden selbst verändert. Es ist nichts weiter zu tun, als die beiden Senken zu vertauschen.

3.2.4.5 Ersetzung durch Konstanten und Quantifizierung

Die Operation „Ersetzung durch Konstanten“ erlaubt es, eine Variable x_i in dem OBDD konstant auf 1 oder 0 zu setzten. Durch den Methodenaufruf `OBDD.assignConstant (int i, boolean constant)` wird ein reduziertes OBDD berechnet, in dem die Variable x_i auf den Booleschen Wert `constant` gesetzt ist. Alle Vorkommen von mit x_i markierten Knoten werden eliminiert und ihre eingehenden Kanten auf den entsprechenden `constant`-Nachfolger des x_i-Knotens gesetzt.

Durch diese Methode ist es leicht, in einem OBDD die Variable x_i zu quantifizieren. Hierzu wird das OBDD G_{a0} für die Funktion $a_{|x_i=0}$ und das OBDD G_{a1} für die Funktion $a_{|x_i=1}$ berechnet. Falls x_i allquantifiziert werden sollen, werden die beiden OBDDs G_{a0} und G_{a1} mit AND synthetisiert. Die Funktion soll alle möglichen Belegungen von x_i erfüllen. Falls x_i existenzquantifiziert werden soll, werden die OBDDs mit OR synthetisiert. Die Bibliothek stellt hierzu die Methoden `OBDD.allQuantify (int variable)` und `OBDD.existsQuantify (int variable)` zur Verfügung. Die Rechenzeit der Operation „Ersetzung durch Konstanten“ ist $O(|G|)$. Dementsprechend benötigt die Quantifizier Operation auf Grund der Laufzeit der Synthese $O(|G|^2)$ Rechenschritte.

3.2.4.6 Aufbau aus Formeln

Die Automatenbibliothek bietet die Möglichkeit, OBDDs aus einer Präfixformel zu konstruieren. Die Formel muss der folgenden Grammatik genügen:

BOOLEXP ::= “atom“ | “*“ | “#“ | “&(BOOLEXP,BOOLEXP)“ | “|(BOOLEXP,BOOLEXP)“ | “!(BOOLEXP)“

Das Zeichen „*“ entspricht hier dem *true* und das Zeichen „#“ dem *false*. Die Formel muss mit dem `aaa.automaton.util.obdd.formula.TinyParser` in einen `BooleanExpression`-Baum transformiert werden. Anschließend übergibt man den Baum einem `aaa.automaton.util.obdd.formula.OBDDBuilder`-Objekt, das mit dem Methodenaufruf `OBDDBuilder.createFromFormula( BooleanExpression root, AssignmentAlphabet alphabet)` das zu der Formel entsprechende OBDD konstruiert. Das übergebene Alphabet muss die in der Formel enthaltenen Atome kennen.

Beispiel:

```
AssignmentAlphabet psa =
              new AssignmentAlphabet ("x1", "x2", "x3");
String formula = "|(x1,&(x2,x3))";
```

```
TinyParser tp = new TinyParser ();
BooleanExpression boolexp = tp.parse(formula);
OBDD obdd = OBDDBuilder.createFromFormula(boolexp, psa);
```

Umgekehrt lässt sich ein OBDD auch wieder zurück in eine Formel umwandeln. Es wird immer die Shannon-Zerlegung berechnet. Diese ist wie folgt definiert:

Definition 2. *Die von einem Knoten v dargestellte Funktion f_v ergibt sich wie folgt:*

- *falls v eine Senke ist, dann ist die dargestellte Funktion f_v der Wert der Markierung von v.*
- *falls v ein innerer Knoten mit Beschriftung x_i ist, dann ist $f_v(x) = (x_i \wedge f_{v_0}) \vee (\overline{x}_i \wedge f_{v_1})$. Wobei f_{v_0} die vom 0-Nachfolger von v dargestellte Funktion ist.*

Der `aaa.automaton.util.obdd.formula.OBDD2BooleanExpressionConverter` berechnet aus einem OBDD einen `BooleanExpression`-Baum, der dann mit dem `InfixConverter` im selben Package zu einer Formel in Infixnotation konvertiert werden kann.

Beispiel:

```
OBDD2BooleanExpressionConverter conv =
              new OBDD2BooleanExpressionConverter(obdd);
BooleanExpression converted = conv.computeResult();
InfixConverter infix = new InfixConverter(converted);
System.out.println(infix.computeResult());
```

Eleganter löst diese Aufgabe allerdings die `toString()` Methode der `OBDD`-Klasse.

3.2.4.7 Nutzung

Während bei endlichen Automaten sogar nicht vorhandene Knoten durch eine neue Transition implizit erzeugt und verbunden werden können, kann dies bei OBDDs nur explizit durch die `connect` - Methode mit vorhandenen Knoten geschehen. Durch diese Methode werden zwei Zustände `from` und `to` durch eine Kante mit dem Booleschen Wert `value` (`true` entspricht 1, `false` entspricht 0) verbunden, falls es von `from` noch keine ausgehende Kante mit `value` gibt. Das Gegenstück zu `connect` ist `disconnect`. Diese Methode entfernt alle direkten Kanten zwischen einem `from` und einem `to` Zustand. `removeTransition` in der Klasse `OBDD` benutzt genau diese Methode.

3.2.5 Büchiautomaten

Das bisher betrachtete Modell endlicher Automaten ist in mancherlei Hinsicht sehr eingeschränkt. Endliche Automaten sind nicht in der Lage, Wörter aus übermäßig ausdrucksstarken Sprachen zu erkennen. Es lassen sich immer nur endliche Wörter erkennen, und es ist auch nicht möglich, zu „zählen“, da der Zustandsraum dieser Automaten endlich ist. Schon kontextfreie Sprachen können nicht mehr erkannt werden.

Allerdings gibt es eine andere, besondere Automatenklasse, die strukturell den endlichen Automaten gleicht, aber so interpretiert wird, dass damit reguläre Sprachen unendlicher Wörter repräsentiert werden können. Das ist der gewöhnliche Büchiautomat.

Damit erschließen sich eine große Reihe neuer Nutzungsmöglichkeiten. Im Gegensatz zu der Klasse der endlichen Automaten ist es nicht unbedingt wünschenswert, Läufe auf unendlichen Eingaben zu simulieren, da diese kein Ende hätten. Jedoch ist die Analyse der grundlegenden Mengeneigenschaften der erkannten Sprache dieser Automatenklasse interessant: Vereinigung, Schnitt, Komplement und Leerheitstest.

Schon mit wenigen Analyseoperationen, insbesondere dem Schnitt und dem Leerheitstest, lassen sich Analysen auf Strukturen durchführen, die überaus interessant und nützlich sind, nicht nur im theoretischen Sinne, sondern auch für praktische Anwendungen.

Eingaben eines Büchiautomaten sind unendliche Wörter, die man als zeitliche Abläufe interpretieren kann, die einen Anfang, aber kein bekanntes Ende haben. So kann man beispielsweise einen Programmablauf modellieren, der nach dem Start unbeschränkt weiterläuft. Der Programmlauf besteht aus einer Folge von Zuständen, dargestellt als Folge von Zeichen aus dem Alphabet des Automaten.

Kripkemodelle sind Graphen, deren Pfade solche Programmabläufe repräsentieren. Sie können unmittelbar in Büchiautomaten überführt werden. Da sich alle Aussagen über Büchiautomaten auf unendliche Wörter beziehen, sind die Methoden der Analyse dieser Automaten nur dazu geeignet, Aussagen über lineare Programmabläufe in Kripkemodellen zu machen. Verzweigungen müssen unberücksichtigt bleiben.

Die Konsequenz ist, dass mittels linearer Temporallogik Kripkemodelle auf gewisse Eigenschaften überprüft werden können. Es ist also denkbar, dass ein Kripkemodell ein komplexeres System modelliert und dieses mit einer temporallogischen Formel auf die Erfüllung gewisser Eigenschaften geprüft werden kann, um das durch das Kripkemodell modellierte System zu verifizieren.

Die AAA-Automatenbibliothek bietet neben den bekannten endlichen Automaten eben dieses Automatenmodell der Büchiautomaten an, um damit Analysen durchführen zu

können.

3.2.5.1 Komplement

Das Komplement von Büchiautomaten ist zwar konstruierbar, aber für die gegebenen Ressourcen nur für extrem kleine und damit seltene Büchiautomaten durchführbar. In [**?**] wird eine Konstruktion beschrieben, die $4^{n^2} \cdot (4^{n^2} + 1)$ Zustände benötigt. Deshalb hat sich die Projektgruppe gegen die Implementierung entschlossen und wird sie hier auch nicht weiter betrachten.

3.2.5.2 Vereinigung `union`

Die Vereinigung bei Büchiautomaten funktioniert analog zu den Automaten auf endlichen Worten (siehe 3.2.1.7.2). Auch hier müssen die Automaten dieselben Alphabete haben und beide Automaten werden nicht verändert, sondern ein neuer erzeugt. Wenn der Automat generalisiert ist, wird er degeneralisiert. Ebenso wie bei Automaten auf endlichen Worten, ist der resultierende Automat nichtdeterministisch. Da aber die Klasse der von nichtdeterministischen Büchiautomaten erkennbaren Sprachen größer ist als die der von deterministischen erkennbaren, ist er auch nicht notwendigerweise in einen deterministischen Büchiautomaten überführbar.

Das Resultat des Methodenaufrufs lässt sich in maximal linearer Laufzeit berechnen.

3.2.5.3 Schnitt `intersect`

Die Schnittoperation liefert einen Automaten, der den Schnitt der Sprachen der beiden Automaten A und B akzeptiert ($L(A) \cap L(B)$).

Im Gegensatz zur Klasse `FiniteAutomaton` werden die Automaten nicht determinisiert, aber degeneralisiert. Die Operation hat keinen Einfluss darauf, ob der Automat deterministisch ist oder nicht. Sie verändert beide Automaten nicht, sondern erzeugt einen neuen. Die Alphabete der Automaten müssen gleich sein, andernfalls wird eine Exception geworfen.

Die Details der Konstruktion kann man in Abschnitt 2.1 auf Seite 12 nachlesen. Bei dieser expliziten Konstruktion ist verhältnismäßig viel Speicher und Rechenzeit erforderlich (es kommt zu einem quadratischen blow-up). Wenn man sich nur dafür interessiert, ob der die Sprache $L(A) \cap L(B)$ erkennende Automat leer ist, sollte man sich der in Abschnitt 3.2.5.5 beschriebenen Methoden bedienen.

3.2.5.4 Leerheitstest `isEmpty()`

Ein Büchiautomat akzeptiert (irgend) ein unendliches Wort genau dann, wenn es einen zu dem Wort korrespondierenden Run gibt, bei dem ein akzeptierender Zustand unendlich oft besucht wird. Ein hinreichendes und notwendiges Kriterium dafür ist die Existenz eines vom Startzustand erreichbaren Kreises, der einen akzeptierenden Zustand enthält.

Solch ein Kreis kann wie folgt mit einer doppelten Tiefensuche gefunden werden (siehe auch Abschnitt 2.1.1.3.3 auf Seite 13). Die erste Tiefensuche beginnt am Startzustand des Automaten. Jedesmal, wenn ein akzeptierender Zustand (a) abgearbeitet wurde, wird von (a) aus eine zweite Tiefensuche gestartet. Diese bricht mit der Ausgabe „Der Automat ist nicht leer“ ab, wenn sie einen Zustand (b) erreicht, der sich auf dem von der ersten Tiefensuche benutzten Stack befindet.

Wenn bei keiner dieser Tiefensuchen ein Zustand gefunden wird, der auf dem Stack der ersten Tiefensuche ist, wird „Der Automat ist leer.“ ausgegeben.

Eine Eingabe, die vom Automaten akzeptiert wird, erhält man im Fall der Nicht-Leerheit wie folgt. Das endliche Präfix dieser Eingabe ist ein Wort, über das man vom Startzustand zum Zustand (b) gelangt. Der unendlich oft wiederholte, zyklische Teil der Eingabe besteht erstens aus dem Wort, über das man vom Zustand (b) zum Zustand (a) gelangt, und zweitens aus dem Wort, über das man von (a) zu (b) zurück gelangt. Die Teilwörter lassen sich durch Speichern von Zusatzinformationen auf den Stacks der Tiefensuchen leicht rekonstruieren.

Die zweite Tiefensuche wird zwar unter Umständen mehrmals gestartet, aber jeder Zustand wird höchstens einmal dabei besucht. Trotzdem findet das Verfahren stets einen Kreis mit einem akzeptierenden Zustand, falls ein solcher existiert.

Das kann wie folgt per Widerspruchsbeweis gezeigt werden. Sei q der erste akzeptierende Zustand, von dem aus die zweite Tiefensuche einen Kreis nicht findet, weil ein auf diesem Kreis liegender Zustand r von einer bei q' startenden zweiten Tiefensuche bereits besucht wurde. q' liegt auf keinem Kreis (sonst wäre schon die bei q' startende zweite Tiefensuche gescheitert) und q ist über r von q' aus erreichbar. Dann müsste aber die erste Tiefensuche q abgearbeitet haben, bevor q' abgearbeitet wurde. Das ist ein Widerspruch zu der Annahme, dass die zweite Tiefensuche zuerst bei q' begann. Also wird ein Kreis in jedem Fall gefunden.

Das Verfahren macht nur linear viele Schritte und die Laufzeit ist um einen logarithmischen Faktor größer als linear, wenn Hash-Tabellen verwendet werden für das Speichern bereits besuchter Zustände sowie als Indikator dafür, welche Zustände auf dem Stack liegen.

3.2.5.5 Impliziter Leerheitstest des Automaten für $L(A) \cap L(B)$ `findCommonRun()`, `findCommonWord()`

Oft interessiert man sich nur dafür, ob die von zwei gegebenen Büchiautomaten A, B erkannten Sprachen einen leeren Schnitt haben. Dies kann auch entschieden werden, ohne dass der Automat, der diesen Schnitt erkennt, explizit konstruiert wird. Dafür stehen die Methoden `findCommonRun` und `findCommonWord` zur Verfügung, die auf dem Automaten A aufgerufen werden und die den Automaten B als Parameter haben.

Um die in Abschnitt 3.2.5.4 beschriebene doppelte Tiefensuche durchzuführen, werden lediglich die Adjazenzlisten des gerade besuchten Zustands und der Zustände auf dem Stack benötigt. Ist ein Zustand dieses Automaten (implizit) gegeben – das ist ein Tripel aus zwei Zuständen der Originalautomaten A, B und einem Index –, so können, wie in Abschnitt 2.1 auf Seite 12 beschrieben, alle im Automaten für $L(A) \cap L(B)$ dazu adjazenten Zustände ebenfalls durch solche Tripel dargestellt werden, ohne dass eine Datenstruktur für den gesamten Ergebnisautomaten existiert. An einem dieser Zustände wird die Tiefensuche fortgesetzt, die übrigen Zustände werden zusammen mit dem aktuellen Zustand auf dem Stack gespeichert.

Beginnt man also bei den Startzuständen des Automaten für $L(A) \cap L(B)$ (die Tripel bestehen aus zwei Startzuständen und dem Index 0), so kann alles erledigt werden, was im Abschnitt 3.2.5.4 beschrieben wird. Die Methode `findCommonWord` liefert, wie dort erwähnt, ein im Schnitt von $L(A)$ und $L(B)$ liegendes Wort, falls er nicht leer ist. Die Methode `findCommonRun` liefert eine zu diesem Wort korrespondierende Zustandsfolge des Automaten, auf dem die Methode aufgerufen wurde. Rückgabewerte der Methoden sind Paare von Wörtern bzw. Paare von Zustandsfolgen, wobei die erste Komponente das endliche Präfix der akzeptierten Eingabe/des akzeptierenden Runs ist und die zweite Komponente der zyklische Teil.

Der Vorteil dieses impliziten Leerheitstests besteht darin, dass nur Speicher für die Stacks der Tiefensuchen benötigt wird. Von selten vorkommenden worst-case-Eingaben (z. B. Automaten, die nur aus langen Ketten bestehen) abgesehen, ergibt sich ein viel geringerer Platzbedarf als bei der expliziten Konstruktion des Ergebnisautomaten. Außerdem entfällt die für diese Konstruktion benötigte Rechenzeit. So können auch noch Eingaben bewältigt werden, für die sonst wegen Speicherplatzmangels kein Leerheitstest mehr durchgeführt werden kann.

3.2.5.6 Beispiele

Nun folgen Beispiele für die oben beschriebenen Operationen :

```
/* Erzeugt Symbole. */
TriggerFactory tf = new SymbolRangeFactory();
CharacterAlphabet ca = CharacterAlphabet.getInstance();
Trigger a = tf.createSingleSymbolTrigger(ca.createSymbol("a"));
Trigger b = tf.createSingleSymbolTrigger(ca.createSymbol("b"));
Trigger c = tf.createSingleSymbolTrigger(ca.createSymbol("c"));
Trigger abc = a.join(b.join(c));
Trigger bc = b.join(c);
Trigger ab = a.join(b);

/* Konstruiert Automat A. */
BuchiAutomaton baA = new BuchiAutomaton(SymbolRangeList.class,
                                                    ca);

State a1 = new State("A1");
State a2 = new State("A2");
a1.setInitial(true);
a2.setAccepting(true);
baA.addState(a1);
baA.addState(a2);

baA.addTransition(a1, abc, a1);
baA.addTransition(a1, a, a2);
baA.addTransition(a2, a, a2);
baA.addTransition(a2, bc, a1);

/* Konstruiert Automat B. */
BuchiAutomaton baB = (BuchiAutomaton) baA.createFromTemplate();
State b1 = new State("B1");
State b2 = new State("B2");
State b3 = new State("B3");

b1.setInitial(true);
b3.setAccepting(true);
baB.addState(b1);
baB.addState(b2);
baB.addState(b3);
baB.addTransition(b1, abc, b1);
baB.addTransition(b1, b, b2);
```

```
baB.addTransition(b2, c, b3);
baB.addTransition(b3, abc, b1);

/* Vereinigung von Automat A und B */
BuchiAutomaton a = baA.union(baB);

/* Schnitt von Automat A und B */
BuchiAutomaton a = baA.intersect(baB);

/* Sucht gemeinsames Wort von Automat A und B */
Pair<List<Symbol>, List<Symbol>> commonWord =
        baA.findCommonWord(baB);

/* Testet ob Automat A leer ist. */
boolean result = baA.isEmpty();
```

3.2.5.7 Degeneralisierung `degeneralize`

Ein generalisierter Büchiautomat (GBA) hat nicht nur *eine* Menge akzeptierender Zustände, sondern *mehrere* „Akzeptanzmengen“, die nicht paarweise disjunkt sein müssen. Er akzeptiert ein Wort genau dann, wenn beim Lesen des Worts mindestens ein Zustand aus jeder Akzeptanzmenge unendlich oft eingenommen wird.

Nach Aufruf der Methode `renderGeneralized()` stellt ein `BuchiAutomaton`-Objekt einen generalisierten Büchiautomaten dar. Falls vorher akzeptierende Zustände vorhanden waren, bilden diese nun die erste Akzeptanzmenge des GBA (mit Index 0). Weitere Akzeptanzmengen können durch Aufruf der Methode `addAcceptanceSet(Set<State>)` hinzugefügt werden. Diese Methode liefert den Index der hinzugefügten Akzeptanzmenge zurück. Einzelne Zustände können per `addStateToAcceptanceSet(State,int)` zu einer durch ihren Index beschriebenen Akzeptanzmenge hinzugefügt werden. Informationen zu bestehenden Akzeptanzmengen erhält man über die Methoden `getAcceptanceSets()`, `getAcceptanceSet(int)` und `getAcceptanceSetsOf(State)`, die jede oder eine bestimmte Akzeptanzmenge liefern bzw. die Indizes der Akzeptanzmengen, in denen sich ein Zustand befindet.

Mit einem GBA lassen sich ω-reguläre Sprachen unter Umständen kompakter darstellen, die Ausdrucksstärke nimmt aber nicht zu. Denn zu jedem GBA A lässt sich leicht ein äquivalenter BA A' berechnen, der das Verhalten von A simuliert. A' enthält für jede Akzeptanzmenge eine Kopie von A. Wenn in einer Kopie ein Zustand aus der zu der

nächsten Kopie korrespondierenden Akzeptanzmenge erreicht ist, wird in diese Kopie gewechselt:

Seien $F_0, \ldots, F_{n-1}$ die Akzeptanzmengen von A. A' besteht aus n Kopien von A: $A_0, \ldots, A_{n-1}$. In A_i wird für jeden Zustand aus F_{i+1} jede ausgehende Transition wie folgt geändert. Wenn vorher das Ziel von t der Zustand q in A_i ist, so ist das Ziel nachher der q entsprechende Zustand in $A_{i+1 \bmod n}$. Die Startzustände von A' sind die Startzustände in A_0. Akzeptierende Zustände von A' sind alle Zustände in A_{n-1}. Die Größe von A' nimmt gegenüber der von A um den Faktor n zu.

Zu einem `BuchiAutomaton`-Objekt, das einen GBA A darstellt, kann das beschriebene Verfahren durch Aufruf der Methode `degeneralize()` ausgeführt werden. Die Methode liefert den äquivalenten Büchiautomaten A' zurück.

```
CharacterAlphabet ca = CharacterAlphabet.getInstance();
BuchiAutomaton ba = new BuchiAutomaton(SymbolRangeList.class,
                                                    ca);
ba.renderGeneralized();

State q1 = new State("q1");
State q2 = new State("q2");
State q5 = new State("q5");
q1.setInitial(true);

SymbolRangeList as = new SymbolRangeList(ca.createSymbol("a"));
SymbolRangeList bs = new SymbolRangeList(ca.createSymbol("b"));

ba.addTransition(q1, as, q2);

ba.addTransition(q2, bs, q5);

Set<State> m0 = new HashSet<State>();
Set<State> m1 = new HashSet<State>();
m0.add(q2);
m0.add(q5);
m1.add(q5);

ba.addAcceptanceSet(m0);
ba.addAcceptanceSet(m1);
```

```
BuchiAutomaton nonGBA = ba.degeneralize();
```

3.2.6 Kripkestruktur

Kripkestrukturen (oder Kripkemodelle) sind einfache Strukturen, mit denen Transitionssysteme beschrieben werden können. Zur Erinnerung, formal ist eine Kripkestruktur K ein Tupel (S_0, S, R, L, AP). Dabei ist S eine beliebige Menge, die Zustände beschreibt, z. B. $\mathbb{N}$ und $S_0 \subseteq S$ ist die Menge der Anfangszustände. R ist eine linkstotale Übergangsrelation $\subseteq S \times S$. Die Abbildung $L : S \to 2^{AP}$ bildet Zustände auf Mengen von *atomaren Propositionen* aus AP ab. K lässt sich also als Graph mit Knotenbeschriftungen von Elementen aus AP auffassen. Atomare Propositionen sind dabei aufzufassen als nullstellige Prädikate, deren Knotenbeschriftungen praktisch Vollkonjunktionen sind. Nicht vorhandene Elemente aus AP können als negiert in S aufgefasst werden.

Da die abstrakte Klasse `Automaton` ein Grundgerüst für graphenartige Objekte bietet, liegt es nahe, dass auch Kripkemodelle diese Klasse als gemeinsames Modell nutzen. Somit muss kaum Code geschrieben werden, da Kripkemodelle eine sehr einfache Datenstruktur sind und viel eingeschränkter als normale Automaten. Die Menge der atomaren Propositionen, die in einem Zustand gelten, werden einfach an das Label eines Zustandes gehängt, und zwar als `BitVector` (bzw. `Assignment`, um ausdrucksstärker zu sein, siehe Abschnitt zu Alphabeten), der eine Vollkonjunktion repräsentiert.

Da Zustände (theoretisch) immer akzeptierend sind und die Beschriftung (der *Trigger*) einer Transition belanglos, brauchen wir speziell eingeschränkte Methoden. Diese überschreiben die herkömmlichen Methoden so, dass sie keinerlei andere Zustände oder Transitionen, die dagegen verstoßen, annehmen; sie werfen Exceptions.

Zusätzlich sollte der Benutzer nicht unnötig komplexe Schnittstellen für so einen einfachen Aufbau nutzen müssen. Zu diesem Zwecke wurden die beiden Methoden

- `void addTransition(int fromKey, int toKey);`
- `void addState(int key, boolean isInitial, Assignment atomicPropositons);`

eingeführt. Zustände werden nur noch über Zahlen referenziert, Zustände können nur initial oder nicht sein und erhalten sofort ihre AP. Transitionen werden nur als Paar von Zuständen angegeben.

3.2.6.1 Umwandlung in Büchiautomaten

Die Konstruktion eines angemessenen Büchiautomaten aus einem Kripkemodell wird wie folgt umgesetzt: Ein neuer, einziger Initialzustand zeigt auf alle bisherigen Initialzustände des Kripkemodells. Die atomaren Propositionen eines Zustandes werden zu `Trigger` an den Transitionen, die als Zielzustand diesen Zustand haben. Die Methode trägt die Signatur `BuchiAutomaton KripkeModel.toBuchiAutomaton()`.

3.2.6.2 CTL Model Checking

Die Klasse `CTLModelChecker` erlaubt durch die Methode `check(String)` die Erfüllbarkeit einer CTL Formel f auf einem Kripkemodell K zu testen, d. h. ob K ein Modell von f ist, bzw. ob das System, dem K unterliegt, gewisse Eigenschaften, die f beschreibt, erfüllt. Dabei wird f als String angegeben und von einem Parser intern umgewandelt in eine interne Repräsentation der CTL Formel. `check` liefert `true` gdw. K ein Modell von f ist, d. h. die Formel in allen Startzuständen von K gilt. Alternativ kann man durch `getValidStates(String)` auch die Menge aller Zustände erhalten, in denen f gilt.

Das CTL Model Checking verläuft dabei mit einer simplen Fixpunktberechnung. Das unterliegende System wurde in der Vorlesung Softwarekonstruktion im Wintersemester 2005/2006 an der Universität Dortmund erläutert. Die Idee ist atomare Formeln mit der Menge der Zustände, in denen sie gelten, zu identifiziert und Konjuktionen, Disjunktionen und Komplementbildung auf Mengenoperationen überträgt. Die Temporaloperatoren werden durch die Auswertung monotoner Fixpunktberechnungen aufgelöst und das System wird induktiv angewendet.

3.2.6.3 Beispiel

Folgendes kleine Beispiel konstruiert ein Kripkemodell, das eine Kette bildet, in dem am Anfang alle Propositionen gelten, und dann jeweils immer eine Proposition weniger gilt.

```
String[] propositions = {"a", "b", "c"};
AssignmentAlphabet alphabet =
        new AssignmentAlphabet(propositions);

KripkeModel model = new KripkeModel(alphabet);
model.addState(0, true,
        alphabet.createFromNameArray("a","b","c"));
model.addState(1, false,
        alphabet.createFromNameArray("a","b"));
```

```
        model.addState(2,false,
                alphabet.createFromNameArray("a"));

        model.addTransition(0, 1);
        model.addTransition(1, 2);
        model.addTransition(2, 2);
```

Und so könnte man Model Checking betreiben:

```
        CTLModelChecker mc = new CTLModelChecker(model);

        String f1s = "(a && b) EU (! c)";
        String f2s = "EG a";
        try {
                System.out.println(mc.check(f1s));
                System.out.println(mc.check(f2s));
        } catch (ParseException e) {
                e.printStackTrace();
        }
```

Bei beiden Beispielen ist es ratsam, wenn sich der Leser als kleine Denkübung klar macht, was der Code bezweckt.

3.3 Parser

Ein Parser überführt bzw. zerlegt eine Eingabe, anhand der vorgegebenen Grammatik, in eine Baumstruktur.

Der Parser behandelt den kontextunabhängigen Teil einer Sprache. Das dabei verwendete übliche automatentheoretische Modell ist ein einfacher Stackautomat.

Je nach Struktur der verwendeten Grammatik bzw. der Einsatzumgebung oder sonstiger Randbedingungen, können sich verschiedene Arten von Verfahren mehr oder weniger eignen. Im Folgenden seien ein paar genannt:

- Parser mit Sprung-/ bzw. Steuertabelle: Dies ist möglich, da Stack und Sprungtabelle funktional-äquivalent sind. Die Sprungtabelle versetzt den Parser jeweils in verschiedene Zustände, die verschiedenen Stackkonfigurationen entsprechen. Parser, die mit solchen Tabellen arbeiten, nennt man tabellengesteuert. Als Konsequenz ist der Rahmen eines solchen Parsers immer gleich und wird auch "Treiberprogramm" genannt.

- "Vorausschauender"[10] Parser: Dies ist die einfachste Möglichkeit einen Parser zu implementieren, sowie auch von der Ausführungsgeschwindigkeit die schnellste. Die Grammatiken, die verwendet werden können, sind allerdings eingeschränkt. Auf die Einschränkungen soll an dieser Stelle jedoch nicht weiter eingegangen werden. Die genaue Vorgehensweise ist ähnlich dem des rekursiv-absteigenden Parsers und genauer in 3.3.3 beschrieben.

- Rekursiv-absteigende Parser mit Backtracking: Ist eine erweiterte Variante des "vorhersehbaren" Parsers, jedoch um die Möglichkeit des Backtrackings[11] erweitert. Das Konzept dieses Types wird in 3.3.2 mit einem Beispiel erläutert.

3.3.1 Parser Infrastruktur

Die Parser-Infrastruktur ist eine Kombination mehrerer Klassen, die den Bau, Auswertung und interne Verwaltung eines Parsers vereinfachen.

[10] im engl. predictive
[11] eine lose Übersetzung wäre: Zurückgehen oder "Durchprobieren"

3.3.1.1 Token (`aaa.parser.util`)

Token ist eine Containerklasse, die Informationen über eine Zeichenkette bereithält. Im Einzelnen sind dies: Typ und Supertyp, sowie die eigentliche Zeichenkette und die Position in der Eingabe.

Typ und Supertyp werden vom jeweiligem Parser als `enum` bereitgestellt. Im Allgemeinen kann davon ausgegangen werden, dass ein Supertyp eine Oberklasse wie "binärer Operator" darstellt und Typ ein konkretes Element der Klasse wie "Plus".

Bei der Verwendung der Informationen innerhalb eines Tokens ist zu beachten, dass die Zeichenkette innerhalb eines Tokens nicht unbedingt mit der Repräsentation innerhalb der Eingabe übereinstimmen muss. Das Gleiche gilt für die Position.

Der Grund ist darin zu suchen, dass Methoden oder Programme im allgemeinen bei der Auswertung nicht an der Repräsentation eines z. B. Operators interessiert sind, sondern an dem, was dieser Operator bedeutet, also an der Semantik. Eine Vermischung von Syntax und Semantik führt zwangsläufig dazu, dass Methoden und Programme bei der geringsten Änderung der Repräsentation eines Operators angepasst werden müssen. Es führt also dazu, dass diese Programme schwerer zu warten sind.

Nur in Fällen, wo die Semantik durch die Auswertung definiert wird - also im Allgemeinen bei Variablen und Literalen - darf gefahrlos auf die Repräsentation zugegriffen werden.

3.3.1.1.1 Kompatibilitätsinterface Älterer Code, oder Parser, die schnell auf die Verwendung von Tokens umgestellt werden sollen, können die Methode `Token.getMinToken(String i)` verwenden. Diese Methode liefert ein Token vom Typ *`Token`*<*`Token.NullType, Token.`*`NullType`> mit Repräsentation `i` zurück. Der (Super)-Typ `NullType` ist semantiklos.

3.3.1.1.2 InputStream (`aaa.parser.util`) InputStream ist eine allgemeine Managementklasse zur Behandlung von Zeichenstrings im Kontext einen Parsers. Die allgemeinen Bedürfnisse, die diese Klasse abdeckt, sind sowohl das Halten des Eingabestrings, als auch das Verwalten der Position in der Eingabe und Bereitstellung von Methoden zum Zeichenkettenvergleich.

Die gebräuchliche Initialisierung erfolgt über den Konstruktor `public InputStream(String i)`, wobei `i` die Eingabe repräsentiert. Ein späteres Zurücksetzen geschieht über `void reset()`. Ein Eingabestring kann dann nachträglich über `void setInput(String i)` gesetzt werden.

Weitere gebräuchliche Methoden:

3.3.1.1.3 `void advancePointer()` Setzt den internen Zeiger um eine Stelle nach vorne.

3.3.1.1.4 `void skipSpaces() throws StringIndexOutOfBoundsException` Setzt den internen Zeiger so weit nach vorne, bis das nächste Zeichen im Eingabestring kein Leerzeichen ist. Sollte der Zeiger über das Ende der Eingabe hinaus gesetzt werden, wird eine `StringIndexOutOfBoundsException` geworfen. Die Zeigerposition sollte dann als undefiniert gelten.

3.3.1.1.5 `boolean match(String s) throws ParseException` Setzt den internen Zeiger so weit nach vorne, bis das nächste Zeichen im Eingabestring kein Leerzeichen ist und vergleicht die nächsten Zeichen in der Eingabe mit `s`. Im Fehlerfall wird eine `ParseException` geworfen und der interne Zeiger zeigt auf das Zeichen, das den Vergleich fehlschlagen lässt.
Bei der Verwendung ist zu beachten, dass diese Methode im Fehlerfall immer eine Exception wirft und niemals den boolean Wert false zurückgibt.

3.3.1.1.6 `boolean lookahead(String s) throws ParseException` Identisch zu `match(String s)`, mit dem Unterschied, dass der interne Zeiger nicht modifiziert wird.

3.3.1.1.7 `void consume(Token t) throws ParseException` Setzt den internen Zeiger so weit nach vorne, dass das Token `t` übersprungen wird.

3.3.1.1.8 PNode (`aaa.parser`) PNode ist eine Containerklasse, die einen Ausdrucksbaum[12] repräsentiert.

Anwendungen sollten Ausdrucksbäume über die folgenden Methoden auswerten:

3.3.1.1.9 `int numNodes()` Gibt die Anzahl der Kinder des Knotens zurück.

3.3.1.1.10 `PNode getNode(int num)` Gibt das `num`-te Kind des Knotens zurück.

[12]Ein Baum, dessen innere Knoten jeweils einen Operator und dessen Blätter Operanden eines Ausdrucks repräsentieren.

$$\begin{array}{lcl} \underline{S} & \rightarrow & cAd \\ A & \rightarrow & ab \mid a \end{array}$$

Abbildung 3.13: eine einfache Grammatik (BNF)

3.3.1.1.11 `Token getToken()` Gibt das Token des Knotens zurück.

Soll ein Parser einen Ausdrucksbaum konstruieren, sollte dies ausschließlich über einen der Konstruktoren wie `PNode(PNode leftNode, PNode rightNode, Token t)` (für zweistellige Operatoren) geschehen. Müssen Knoten mit mehr Kindern erzeugt werden, ist die übliche Vorgehensweise, einen leeren Knoten mit `PNode(int numNodes)` zu erzeugen und den Knoten mit `void setNode(int num, PNode node)` und `void setToken(Token t)` mit Daten zu füllen.

3.3.1.1.12 Kompatibilitätsinterface Ältere Codeteile, oder Parser, die schnell auf die Verwendung von Tokens und PNodes umgestellt werden sollen, können die statische PNode `PNode NullNode = new PNode(Token.getMinToken(\\0\))` verwenden, um den Ausdrucksbaum zu konstruieren.

3.3.1.1.13 ParseException (`aaa.parser`) Exceptions vom Typ `ParseException` zeigen Fehler in der Eingabe an. Eine Anwendung sollte eine der Methoden `String toString()` (nur Fehlermeldung und Eingabe in einzeiligen String) oder `String toStream()` (Fehlermeldung und vorformatierter mehrzeiliger String mit Zeiger auf die Fehlerposition) zur Weitergabe an den Benutzer verwenden.

Beachtet werden muss, dass das Feld `position` abhängig vom Parsertyp u. U. keinen gültigen Wert enthält.

3.3.2 First-Generation (Presburger-Parser)

Der Parser der ersten Generation ist vom Konzept her ein modifizierter rekursiv-absteigender Parser mit Backtracking. Anhand eines Beispiels soll das Konzept hier kurz erläutert werden:

Sei die Grammatik aus Abbildung 3.13 gegeben und das Eingabewort sei *cad*. Der Parser wird nun auf folgende Art und Weise vorgehen, das Wort zu zerlegen:

1. Da S das Startsymbol ist, wird S nach cAd abgeleitet.

2. Der erste Buchstabe der Ableitung stimmt mit dem ersten Buchstaben aus dem Ein-

F	→	ATOM | "/not" "(" F ")" | "(" F "/or" F ")" | "(" F "/and" F ")" | "/all" V ":" "(" F ")" | "/ex" V ":" "(" F ")" | "(" F ")"
ATOM	→	TRT
T	→	L | V | T "+" T | T "-" T | L "*" T
R	→	"=" | "<" | ">" | "<=" | ">="
V	→	ALPHAALPHANUMSMALL
L	→	POSINTNULL | NEGINT

ALPAHAALPHANUMSMALL: alle klein geschriebenen alphanumerischen Folgen, die mit einem kleinen Buchstaben beginnen
POSINTNULL: positive Integer und die Null
NEGINT: negative Integer
Verwendet das Kompatibilitätsinterface von `Token` und `PNode`.

Abbildung 3.14: Presburger (BNF)

gabewort überein. Der Parser wird nun das erste Nichtterminal (A) nach ab ableiten.

3. Der expandierte Ausdrucksbaum enthält nun an allen Blättern Terminale und repräsentiert das Wort *cabd*, welches nicht mit dem Eingabewort überein stimmt. Der Parser wird nun die andere Alternative ausprobieren: *cad*, was auch mit dem Eingabewort übereinstimmt.

Das Prinzip ist also das Folgende: leite so lange ab, bis ein Fehler gefunden wird und gehe dann zurück und probiere eine Alternative, bis keine Alternativen mehr vorhanden sind (woraus ein Fehler folgt) oder bis der Ausdrucksbaum das Eingabewort repräsentiert.

3.3.2.1 Grammatik

Die implementierte Grammatik ist in Abbildung 3.14 zu sehen.

Die Grammatik wurde in der Implementierung von der Linksrekursion befreit, jedoch hier im Originalzustand belassen. Bei näherer Betrachtung fällt auf, dass die Grammatik in diesem hier gezeigten Zustand nicht eindeutig ist. Durch das Entfernen der Linksrekursion und durch Behandlung der komplexeren Tokens, wie z. B. "/not", als ein einziges Zeichen im Eingabestrom, wird die Grammatik wieder eindeutig.

3.3.2.2 Grundlegende Implementierung

Einfache Implementierungen dieses Parsertyps folgen der Struktur der zugrunde liegenden Grammatik. Für jedes Nichtterminal wird eine eigene Methode implementiert, die eigenständig die in einer Alternative vorkommenden Terminale prüft und weiterführende

Nichtterminale (Methoden) aufruft oder bei Misserfolg aller Alternativen einen Fehler zurück liefert.

Weiterhin sei noch an dieser Stelle anzumerken, dass aufgrund der kleinen Menge an Tokens, die zu erkennen ist, das Erkennen von Zeichenketten direkt im Parser implementiert ist.

3.3.2.3 Entwicklung des Parsers

Nach Abschluss des ersten Prototyps war festzustellen, dass bei großen Eingabelängen die Laufzeit extrem anstieg. Der Grund hierfür war, dass beim Zerlegen sehr viel Zeit zur Zeichenerkennung verloren ging und sehr viele Backtracks gemacht wurden[13]. Als weiterer Grund dieser langen Zerlegungsdauer wurden die zweistelligen Operatoren identifiziert, die den Ausdrucksbaum vor allem in die Breite expandieren ließen. Bei genauerer Analyse des Zeichenerkennungsproblems wurde festgestellt, dass aufgrund der häufigen Backtracks die langen Variablennamen und Literalausdrücke, die vor allem bei automatischen Tests entstanden, immer und immer wieder neu analysiert wurden. Als Abhilfe wurden sukzessive Mechanismen eingeführt, um diese Missstände zu beheben:

1. Ein Caching-System wurde eingeführt. Aufgrund des Umstandes, dass an einer Position in der Eingabe für jedes Nichtterminal nur eine richtige Ableitungsmöglichkeit existiert (gegeben, dass die Grammatik eindeutig ist) kann ein positives oder ein negatives Ergebnis, respektive im positivem Fall, der dort abgeleitete Ausdrucksbaum, zwischengespeichert werden. Im Falle von Variablen bzw. Literalen kann im positivem Ergebnisfall der jeweils andere Fall ausgeschlossen werden, also dessen Cache negativ belegt werden. Im Falle der Nichtterminale gilt dies nicht, da dieselbe Eingabeposition unter Umständen noch von einem anderen Nichtterminal erreichbar ist und dort eine mindestens einschrittige, richtige Ableitung machen könnte. Bekannt ist dieses Verfahren unter dem Namen Packrat (siehe [**?**]).

2. An oberster Stelle der Grammatik wurde ein Lookahead-System eingeführt. Anhand des Zeichens in der Eingabe kann die darauf folgende Ableitung in drei Klassen unterteilt und die richtige Klasse ausgewählt werden.

Diese beiden Mechanismen führten zu einem dramatischen Geschwindigkeitsanstieg.

Mit Einführung der Parserinfrastruktur (siehe 3.3.1) im Zuge der Entwicklung der Parser der zweiten Generation (siehe 3.3.3) wurde dieser Parser ebenfalls, mit Hilfe der Kompa-

[13]Bei einer Formel Länge vom 3200 Zeichen wurden 5,6 Mio. Backtracks durchgeführt. Bei einer Ausdrucksbaumtiefe von 22 und mit 107 unterschiedlichen Elementen dauerte das auf einem Athlon64 2800+ 38 sek.

tibilitätsmethoden aus den Infrastrukturklassen, auf den aktuellen Stand gebracht.

3.3.2.4 Anwendung

Der Parser befindet sich in der Klasse `aaa.parser.PresburgerParser` und wird mit der Methode `PNode parse(String)` aufgerufen. Der Methode wird ein Ausdruck der Presburger Logik übergeben und gibt den dazu gehörenden Ausdrucksbaum als PNode zurück.

Dabei ist zu beachten, dass dieser Parser das Kompatibilitätsinterface für Tokens (siehe 3.3.1.1) und PNodes (siehe 3.3.1.1.8) verwendet. Zu beachten ist, dass das Feld `position` einer `ParseException` (siehe 3.3.1.1.13) u. U. nicht den genauen Fehlerort anzeigt.

3.3.2.5 Kritik

Einer der Vorteile dieses Parsertyps ist es, weit mehr Grammatiken parsen zu können, als übliche tabellengesteuerte LL(k)-Parser. Weiterhin ist die Optimierung und die eigentliche Implementierung einfach und schematisch zu verwirklichen.

Als Nachteil stellt sich allerdings heraus, dass es sehr schwer ist, Aussagen darüber machen zu können, an welcher Stelle sich ein syntaktischer Fehler im Eingabestrom befindet. Das resultiert aus der Tatsache, dass die Technik verlangt, unter Umständen an einer Stelle viele mögliche Ableitungsalternativen auszuprobieren, die jeweils korrekte Teilresultate liefern können, ohne dass auch nur eine zum Erfolg führt. Als weiterer kleinerer Nachteil, der aus der Top-Down-Analyse herrührt, ist, dass alle Grammatiken von Linksrekursion befreit sein müssen, als auch, dass Operatorpräzedenzen nur über eine Änderung der Grammatik zu verwirklichen sind. Wird dies nicht getan, liegt es am Auswerter, eine solche, durch Änderung der Auswertungsreihenfolge und -richtung, zu verwirklichen.

Gerade Letzteres führte dazu, dass davon abgesehen wurde, diese Art des Parsers weiter zu verwenden.

3.3.3 Second-Generation (LTL, CTL, While)

Die Parser der zweiten Generation sind allesamt vorausschauende Parser mit Shunting-Yard (siehe 3.3.3.1), zur Umsetzung von Operatorenpräzedenzen.

Die Technik, die ein vorausschauender Parser verwendet ist ähnlich dem des rekursiv-absteigenden Parsers mit Backtracking (siehe 3.3.2) mit dem Unterschied, dass keine Rücksprünge erfolgen: Der Parser erkennt die richtige Ableitung anhand des nächsten Tokens im Eingabestrom.

CTL	→	P “(“ BINOP P “)“*
P	→	OPD | “(“ CTL “)“ | UNOP P
BINOP	→	“&&“ | “ | | “ | “AU“ | “EU“ | “< −>“ | “->“
OPD	→	“true“ | “false“ | PROP
UNOP	→	“!“ | “AX“ | “AF“ | “AG“ | “EX“ | “EF“ | “EG“

PROP: atomare Propositionen (alle Kombinationen aus Zahlen, kleinen Buchstaben und nicht-reservierten Zeichen)

Abbildung 3.15: Infix CTL (PEG)

CTL	→	BINOP P P | UNOP P | OPD
BINOP	→	“&&“ | “ | | “ | “AU“ | “EU“ | “< −>“ | “->“
OPD	→	“true“ | “false“ | PROP
UNOP	→	“!“ | “AX“ | “AF“ | “AG“ | “EX“ | “EF“ | “AG“

PROP: atomare Propositionen (alle Kombinationen aus Zahlen, kleinen Buchstaben und nicht-reservierten Zeichen)

Abbildung 3.16: Prefix CTL (PEG)

Eine eingehende Analyse der Laufzeit fand für diesen Parsertyp nicht statt, da sie auf Grund der Einfachheit dieses Typus und der erwarteten Formellängen als nicht erforderlich erachtet wurde. Ein weiterer Grund ist, dass die erwartete Laufzeit der Analysen, die auf einem der Parser der zweiten Generation zugreifen, um ein Vielfaches über der Laufzeit eines Zerlegungsvorganges des Parsers liegen.

Für die folgenden Sprachen stehen Parser zur Verfügung : Infix-/Prefix-CTL (Abbildungen 3.15, 3.16), Infix-/Prefix-LTL (Abbildungen 3.3.3, 3.18) und While (Abbildung 3.19).

Es muss beachtet werden, dass die hier abgebildeten Grammatiken jeweils als (siehe [**?**]) PEG[14] angegeben sind.

Die Anwendung eines der Parser ist analog zu dem Parser der ersten Generation (siehe 3.3.2.4). Zur Auswertung des Zerlegunsvorganges siehe 3.3.1.1 und 3.3.1.1.8.

3.3.3.1 Shunting-Yard

”Shunting Yard” ist eine Technik zur Implementation von Operatorpräzedenzen innerhalb eines Parsers. Kern des Ganzen ist

- ein Stack für Operatoren

[14]Eine Mischung aus regulären Ausdrücken und BNF

LTL	→	P “(“ BINOP P “)“ *
P	→	OPD | “(“ LTL “)“ | UNOP P
BINOP	→	“&&“ | “ | | “ | “U“ | “R“ | “< −>“ | “->“
OPD	→	“true“ | “false“ | PROP
UNOP	→	“!“ | “X“ | “F“ | “G“

PROP: atomare Propositionen (alle Kombinationen aus Zahlen, kleinen Buchstaben und nicht-reservierten Zeichen)

Abbildung 3.17: Infix LTL (PEG)

LTL	→	BINOP LTL LTL | UNOP LTL | OPD
BINOP	→	“&&“ | “ | | “ | “U“ | “R“ | “< −>“ | “->“
OPD	→	“true“ | “false“ | PROP
UNOP	→	“!“ | “X“ | “F“ | “G“

PROP: atomare Propositionen (alle Kombinationen aus Zahlen, kleinen Buchstaben und nicht-reservierten Zeichen)

Abbildung 3.18: Prefix LTL (PEG)

- ein Stack für Operanden
- eine Präzedenzmethode, die einem Operator eine Präzendenz zuordnet
- zwei Verwaltungsmethoden ”popOperator” und ”pushOperator”.

Die beiden letzteren Methoden implementieren folgendes Verhalten :

3.3.3.1.1 `pushOperator(op)` Solange auf dem Operatorenstack ein Operator mit höherer Präzedenz ist als op, wird `popOperator()` aufgerufen. Danach wird der Operator `op` auf den Operatorenstack gelegt.

3.3.3.1.2 `popOperator()` Nimmt den obersten Operator vom Operatorenstack, je nach Stelligkeit des Operators entsprechend viele Operanden vom Operandenstack und konstruiert daraus einen Knoten des Ausdrucksbaums. Der Knoten wird danach auf den Operandenstack gelegt.

Der Parser verwendet jeweils `PushOperator`, um einen erkannten Operator zu verarbeiten. Operanden werden einfach auf den Operandenstack gebracht.

PRG	→	“Decl“ VARDECL“;“ {!GLUE! VARDECL “;“}* “endDecl“ “;“ !GLUE! S
VARDECL	→	VAR “[“ INT “,“ INT “]“
S	→	METHOD {!GLUE! METHOD}* !GLUE! INIT
METHOD	→	“Proc“ VAR {PARALIST}? “{“ MBODY “}“ “endProc“ “;“
PARALIST	→	“(“ “)“
MBODY	→	I“;“ {!GLUE! I“;“}*
INIT	→	“Init“ “{“ MBODY “}“ “endInit“ “;“
I	→	(ASS | IF | WHILE | SKIP | SPAWN | ATOMIC | CALL | BLOCK | NBLOCKQW | NBLOCKQR)
ATOMIC	→	“atomic“ “{“ MBODY “}“
CALL	→	“call“ VAR PARALIST
SPAWN	→	“spawn“ VAR PARALIST
BLOCK	→	(“block“ | “wait“) “until“ “(“ BOOL “)”
SKIP	→	“skip“
ASS	→	VAR “=“ ARITH
NBLOCKQW	→	VAR < − ARITH
NBLOCKQR	→	VAR -> VAR
IF	→	“if“ “(“ BOOL “)“ “{“ MBODY “}“ {“else“ “{“ MBODY “}“ }?
WHILE	→	“while“ “(“ BOOL “)“ “{“ MBODY “}“
ARITH	→	P1 {ARITH_BINOP P1}*
P1	→	“(“ ARITH “)“ | (INT | VAR)
ARITH_BINOP	→	“+“ | “-“ | “*“ | “/“
BOOL	→	P2 {LOG_BINOP P2}*
P2	→	“(“ BOOL “)“ | (ARTIH) BOOL_BINOP (ARITH)
LOG_BINOP	→	“ | | “ | “&&“
BOOL_BINOP	→	“<“ | “>“ | “==“ | “>=“ | “<=“ | “!=“
SHADOWS	:	!GLUE!

INT: positive und negative ganze Zahlen

VAR: alle Kombinationen aus Zahlen, kleinen Buchstaben und nicht-reservierten Zeichen

!GLUE!: spezielle Ableitung ohne Zeichenrepräsentation

Abbildung 3.19: While (PEG)

3.3.4 Parser für reguläre und ω-reguläre Ausdrücke

3.3.4.1 Motivation

Dieser Abschnitt befasst sich mit der Umwandlung von regulären Ausdrücken in Automaten, wie sie in AAA verwirklicht wurde. Die Vorgehensweise für reguläre Ausdrücke ist [**?**] entnommen. Sie wurde später auf ω-reguläre Ausdrücke erweitert. Am Anfang verwendeten wir die BRICS-Automatenbibliothek, um dies zu bewerkstelligen. Da wir unsere eigene Automatenbibliothek im Verlauf des zweiten Semesters, um Büchiautomaten erweiterten, war es nun auch wünschenswert, aus ω-regulären Ausdrücken Büchiautomaten erzeugen zu können. Dies zu leisten, ist aber die BRICS-Automatenbibliothek nicht in der Lage. So wurde es erforderlich, eine eigene Umwandlung von regulären Ausdrücken zu Automaten zu implementieren. Desweiteren sprach die Unabhängigkeit von `dk.brics` für eine Eigenimplementierung.

Der nun folgende Text befasst sich mit den für diese Aufgabe notwendigen Komponenten. Der erste Schritt ist die Festlegung einer Grammatik für reguläre Ausdrücke und die Erweiterung auf ω-reguläre Ausdrücke. Danach werden die Anforderungen an die Grammatik erklärt werden. Dann wird der Parser näher erläutert und ein Beispiel dazu gegeben.

3.3.4.1.1 Grammatik Nun wird die Entwicklung der Grammatik skizziert. Ziel ist es, eine Grammatik anzugeben, die es erlaubt, von einem einfachen Parsertyp, der von Hand implementiert werden kann, analysiert zu werden. Dies soll in linearer Laufzeit möglich sein. Ein Parser, der diesen Anforderungen genügt, ist ein prädiktiver Parser, der ohne Rücksetzen (backtracking) arbeitet. Damit ein prädiktiver Parser seine Arbeit verrichten kann, ist es nötig, dass die Grammatik frei von Linksrekursionen ist.

Eine Linksrekursion liegt vor, wenn für ein Nichtterminal A eine Ableitung $A \rightarrow^* Aa$ existiert. Da ein prädiktiver Parser top-down arbeitet, löst dieser die Aufgabe, indem er versucht beginnend mit dem Startsymbol, eine Ableitung nach der anderen durchzuführen. Wenn nun eine Herleitung eine Linksrekursion enthält, so wie im Beispiel, dann versucht der Parser, von A zunächst nach A abzuleiten, um in A wiederum nach A abzuleiten und immer so weiter. Der Parser gerät in eine Endlosschleife. Grammatiken für einen prädiktiven Parser müssen also von Linksrekursion befreit werden.

Damit vermieden wird, dass der Parser bei seinem rekursiven Abstieg in den Syntaxbaum zurückgesetzt wird, muss der Parser an jeder Stelle im Syntaxbaum genau wissen, welche Ableitung zu nehmen ist. Dies ist der Fall, wenn die Grammatik für jedes eingelesene Zeichen nur eine Ableitung zulässt. Würden mehrere Ableitungen zugelassen, so müsste der Parser diese Ableitungen nacheinander durchprobieren und bei einem Fehlschlag sowohl

an diese Stelle im Syntaxbaum als auch im Eingabestrom zurückgesetzt werden. Damit dies nicht passiert, muss sichergestellt sein, dass für eine Ableitung niemals zwei oder mehr Produktionen in Frage kommen. In folgender Grammatik kommt so ein Fall vor:

$$\begin{array}{lcc} S & \rightarrow & iEtS \mid iEtSeS \mid a \\ E & \rightarrow & b \end{array}$$

Hier ist es nicht möglich, beim Einlesen des Zeichens i zu entscheiden, welche Produktion, entweder $iEtS$ oder $iEtSeS$, zur Ableitung zu nehmen ist. Gelöst wird ein solches Problem, indem man die Entscheidung, welche Produktion zur Ableitung zu nehmen ist, auf einen späteren Zeitpunkt verschiebt. Dazu führt man ein neues Nichtterminalzeichen ein, welches man hinter den gleich lautenden Teil der betreffenden Produktionen setzt. Im Beispiel sieht dies wie folgt aus:

$$\begin{array}{lcc} S & \rightarrow & iEtSS' \mid a \\ S' & \rightarrow & eS \mid \varepsilon \\ E & \rightarrow & b \end{array}$$

Somit wird zunächst $iEtS$ eingelesen und erst nach der Ableitung zu S' wird entschieden, ob weiter nach eS abgeleitet wird oder nicht. Mit dieser Technik, die man Linksfaktorisierung nennt, kann man meistens dieses Problem beheben.

Mit diesen beiden Techniken kann man eine Grammatik erhalten, die vom gewünschten Parsertyp verarbeitet werden kann. Die nun von uns tatsächlich verwendete Grammatik für reguläre Ausdrücke ist in Abbildung 3.3.4.1.1 dargestellt.

Reguläre Ausdrücke bestehen aus drei verschiedenen Operationen:

1. Konkatenation: Dies ist die Aneinanderreihung von regulären Ausdrücken. Diese Operation wird in der Ableitung $A \rightarrow TC$ durchgeführt.

2. Disjunktion $(r1 \,|\, r2)$: Diese Operation beschreibt, dass ein regulärer Ausdruck $r1$ oder $r2$ erlaubt sein soll. Die Ableitung $B \rightarrow |R \mid \epsilon$ behandelt dies.

3. Kleenescher Abschluss: Sei L eine Sprache, so ist der Kleenesche Abschluss von L, geschrieben als $L\star$, die null oder mehrfache Konkatenation von L. Behandelt wird dies in der Ableitung $D \rightarrow \star \mid + \mid ? \mid \epsilon$. Die Operationen + und ? sind Abkürzungen: $r+$ ist $rr\star$ und $r?$ ist $\epsilon|r$.

$$
\begin{array}{lcll}
R & \rightarrow & AB & \\
B & \rightarrow & \text{“}|\text{“}R & \mid \epsilon \\
A & \rightarrow & TC & \\
C & \rightarrow & A & \mid \epsilon \\
T & \rightarrow & OD & \\
O & \rightarrow & \text{“}(\text{“}R\text{“})\text{“} & \mid \text{“}u\text{“} \\
D & \rightarrow & \text{“}\star\text{“} & \mid \text{“}+\text{“} \mid \text{“}?\text{“} \mid \epsilon
\end{array}
$$

Nichtterminale: $ABCDORT$
Terminale: $|()+?\star$
u: jedes beliebige Unicodezeichen
Startsymbol: R

Abbildung 3.20: Grammatik für reguläre Ausdrücke

3.3.4.1.1.1 Die Bindungspriorität der Operationen Am stärksten bindet der Kleenesche Abschluss, schwächer ist die Bindung der Konkatenation und am schwächsten bindet die Disjunktion. In der Grammatik spiegelt sich das wie folgt wider:

Wird ein regulärer Ausdruck eingelesen, wird zunächst, beginnend vom Startsymbol R nach A, dann T und O abgeleitet. In O wird überprüft, ob die folgende Operation auf einen Klammerausdruck oder ein einzelnes Symbol angewandt wird. Dann wird aus O nach T zurückgegangen und nach D abgeleitet. Hier wird festgestellt, ob ein Kleenescher Abschluss vorliegt. Zurückgesprungen wird dann von D nach T und von T nach A. Abgeleitet wird nun nach C. Hier entscheidet sich, ob nun ein weiterer A Ausdruck angefügt wird oder nicht. Es wird also auf das Vorliegen einer Konkatenation geprüft. Rücksprung erfolgt hiernach von C nach A und von A nach R. Dort wird dann nach B abgeleitet. In B wird auf das Vorliegen einer Disjunktion geprüft.

Der Parser überprüft zuerst die am stärksten bindende Operation, den Kleeneschen Abschluss, dann die Konkatenation und zum Schluss die am schwächsten bindende Operation, die Disjunktion. Auf diese Weise spiegelt die Grammatik die Bindung der Operationen wider.

Da wir nun auch ω-reguläre Ausdrücke ableiten wollen, erweiterten wir die Grammatik um das Terminalsymbol ω. Da das ω ein “unendlicher“ Kleenescher Abschluss ist, bindet es genauso stark wie dieser. Die Grammatik wird, also in der Ableitung D, um die Produktion ω erweitert. D sieht nun so aus:

$$D \rightarrow \star \mid + \mid ? \mid \omega \mid \epsilon$$

```
RegularExpressionParser rep =
        new RegularExpressionParser();
try {
        PNode rep_root = rep.parse("(a|b)+abb");
}
catch (ParseException pe) {}
```

Abbildung 3.21: Verwendung des Parsers für reguläre-Ausdrücke

Die so erweiterte Grammatik erlaubt allerdings “unsinnige“ Ausdrücke, denn sie erlaubt, dass der Ausdruck nach einem ω-abgeschlossenen Teil weiter geht, so dass, nachdem etwas Unendliches eingelesen wurde, weiteres eingelesen werden kann. Um nun dieses Problem in der Grammatik zu behandeln, hätte diese wesentlich umgestellt werden müssen.

Für die regulären Ausdrücke wurde ein prädiktiver Parser, der ohne Rücksetzen auskommt, implementiert. Dieser Parser-Typ arbeitet mit Grammatiken, die frei von Links-Rekursionen sind. Wie die meisten von Hand geschriebenen Parser, nimmt auch dieser eine sogenannte top-down Analyse bei der grammatikalischen Zergliederung der Eingabe vor. Top-down bedeutet, dass ausgehend vom Start-Symbol der gegebenen Grammatik der zur Eingabe gehörende Syntaxbaum expandiert wird. Realisiert wurde dies wie folgt: Für jede Ableitung der Grammatik wurde eine Methode geschrieben. In diesen Methoden wird nun entschieden, abhängig vom nächsten Zeichen in der Eingabe, welche Methode als Produktion aufgerufen wird. Der Syntaxbaum entsteht also durch einen rekursiven Aufruf dieser Methoden. Hier zeigt es sich, warum die zugrundeliegende Grammatik nur eine Produktion pro eingelesenem Zeichen zulassen darf. Würde sie mehrere Produktionen zulassen, müssten diese Produktionen der Reihe nach durchprobiert werden und bei einem Fehlschlagen der Ableitung der Eingabe-Strom zurückgesetzt werden, sowie an die Stelle im Syntaxbaum zurückgesprungen werden, wo die Entscheidung für eine falsche Ableitung getroffen wurde. Wie bereits erwähnt, ist die Grammatik für die regulären Ausdrücke so entworfen worden, dass dieser Fall nicht eintreten kann. Bei der Implementierung des Parsers wurde auf die bereits vorhandene Infrastruktur der in diesem Projekt zuvor implementierten Parser zurückgegriffen. So fanden der Eingabe-Puffer und die Datenstrukturen für den Syntaxbaum (parsetree) auch hier Verwendung. So wurde auch das allgemeine Parser-Interface dem Parser hinzugefügt, sodass einer eventuellen Entwicklung einer Parserfactory nichts im Wege steht.

3.3.4.1.2 Erzeugen eines Syntaxbaums als Beispiel Der Programmcode ist in Abbildung 3.3.4.1.2 zu sehen. Zunächst wird eine Instanz eines RegularExpressionParser erzeugt. Dann wird die Methode `parse` dieser Instanz aufgerufen. Als Parameter wird

ein String mit dem regulären-Ausdruck übergeben. Als Rückgabe erhält man die Wurzel des resultierenden Syntaxbaums. Der Aufruf muss in einen try-catch-Block eingebettet werden, da im Fehlerfall eine ParseException “geworfen“ wird.

3.3.4.2 Die Thompson-Konstruktion

Die Thompson-Konstruktion ist eine einfache Umsetzung des Ausdrucksbaums in einen Automaten. Man kann ihn top-down rekursiv durchlaufen und übergibt in jedem Rekusionsschritt einen Anfangs- und Endpunkt. Zwischen den beiden Knoten wird dann eine Operation eingebaut, die der im Knoten entspricht. Die Operationen sind:

3.3.4.2.1 Konkatenation Die Operation ist sicherlich die häufigste. Sie hängt nur zwei Operationen (die beiden Kinder im Ausdrucksbaumbaum) hintereinander. Sie wird umgesetzt, indem man einen neuen Zustand erzeugt, die Konstruktion rekursiv für die beiden Kinder aufruft und als Startknoten bzw. Endknoten den eigenen Startknoten bzw. den neuen Knoten übergibt. Die zweite Operation bekommt dann den neuen Punkt und den eigenen Endpunkt.

3.3.4.2.2 Disjunktion Die Disjunktion geht in folgende Konstruktion über:

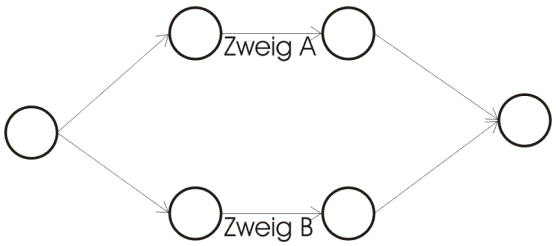


Abbildung 3.22: Disjunktion von zwei regulären Ausdrücken

Die beiden alternativen Zweige des Ableitungsbaums werden mit ε-Transitionen eingebunden.

3.3.4.2.3 Literal Ein Literal erzeugt eine Transition mit dem Literal als Trigger.

3.3.4.2.4 Kleenescher Abschluss Die resultierende Konstruktion sieht etwas so aus:

Es wird ein neuer Zustand eingefügt. Zwischen dem Anfangszustand und dem neuen Zustand wird der Zweig des Ableitungsbaums eingefügt. Entgegengesetzt dazu verläuft

Abbildung 3.23: Kleenescher Abschluss von einem regulären Ausdruck

eine ε-Transition . Außerdem wird der Anfangsknoten noch mit dem Endknoten mit einer ε-Transition verbunden.

3.3.4.2.5 ω-Abschluss Der ω-Abschluss ist dem Kleeneschen Abschluss sehr ähnlich. Es gibt nur 2 Unterschiede. Erstens unterscheidet sich die Semantik. Der Kleenesche Abschluß einer Sprache ist zwar eine beliebig häufige (auch nullfache), aber nicht unendliche Konkatenation. Zweitens muss der ω-Abschluss einen akzeptierenden Zustand beinhalten.

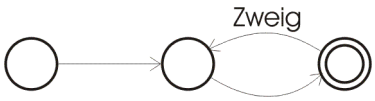


Abbildung 3.24: ω-Abschluß von einem regulären Ausdruck

Wenn die Klasse `ThompsonConstruktion` beim Konstruktor einen String statt einem Ableitungsbaum übergeben bekommt, dann ruft sie den passenden Parser auf. Um die Frage nach dem Typ des Ausdrucks zu klären, wird der Ausdrucksstring nach dem ω-Zeichen durchsucht.

Nachdem der Ausdruck in einen passenden Automaten transformiert wurde, wird der Automat zu Schluss noch mal minimiert. Das bedeutet z. B., dass die ϵ-Kanten entfernt werden.

3.3.4.2.6 Beispielhafter Aufruf Ein Beispiel für einen Aufruf der Thompson-Konstruktion:

```
RegularExpressionParser rep = new RegularExpressionParser();
try {
        new ThompsonConstruction("a(bc)*").computeResult();
}
catch (ParseException pe) {}
```

Es erzeugt einen Automaten, der den regulären Ausdruck a(bc)* abzeptiert.

3.4 Der Workspace - Stand Februar 2006

Die Workspacegruppe beschäftigte sich mit der graphischen Schnittstelle des Projektes.

Der Workspace hat primär die Aufgabe, die Handhabung der Automatenbibliothek, des Parsers und der Transformationsverfahren zu erleichtern und diese graphisch intuitiv nutzbar zu gestalten.

Hierbei bleibt das flexible Gesamtkonzept zu berücksichtigen, das es ermöglicht, den Workspace schnell und unkompliziert zu erweitern. So lassen sich über eine Addon-Schnittstelle neue Automaten oder Transformationsverfahren integrieren.

3.4.1 Planungsphase

Der Implementation des Workspaces ging eine umfangreiche Analysephase voraus, in der wir kurz unsere Vorstellungen und Ziele definiert haben.

Das wichtigste Ziel bestand darin, die Vorgaben eines Baukastenprinzips zu erfüllen. Das heißt, dass der Workspace leicht erweiterbar sein sollte und neue Automaten oder Transformationsverfahren integriert werden können, ohne dafür den eigentlichen Workspace umprogrammieren zu müssen.

In der Interaktion zwischen Benutzern, den Automaten und den Transformationsverfahren sollte der Workspace folgende Punkte erfüllen:

- Es muss eine flexible Eingabemöglickeit geschaffen werden, um eine zu transformierende Aufgabenstellung eingeben zu können.
- Die durch die Transformation entstandenen Automatenmodelle können als Graph angezeigt werden.
- Auf diesen erarbeiteten Ergebnissen können weitere Operationen durchgeführt werden, um ggf. weitere Informationen zu erhalten. Des Weiteren müssen alle automatenspezifischen Operationen durchführbar sein, die von der Automatenbibliothek geliefert werden, wie z. B.: Schnitt, Vereinigung usw.
- Automaten sollen editierbar sein.
- Alle Automaten sollen in einer Projektverwaltung angelegt werden. Projekte sollen abgespeichert und geladen werden können.

Nachdem die Ziele grob definiert worden sind, gab es zwei wichtige offene Fragen, die vor der Implementation zu klären waren. Das erste Problem beschäftigte sich mit der

Plattform unserer Implementation, das heißt, ob wir eine bereits bestehende Plattform wie *Eclipse* oder *jABC* nutzen. Die zweite Problemstellung befasste sich mit dem Layout von Automaten, also der Frage, wie man Automaten übersichtlich in einem Editor anzeigt. Dem Editor ist mit 3.6 ein eigener Abschnitt dieses Berichtes zugeordnet.

3.4.1.1 Plattform

Das Ziel der einfachen Erweiterbarkeit der *Eclipse* Plattform wird durch ein Plug-in-Konzept erreicht. Die Eclipse Plattform besteht aus einem kleinen Kern, der die Basis für Erweiterungen darstellt. Neue Plugins werden in Java und unter Zuhilfenahme der Plugin-Entwicklungsumgebung PDE implementiert. Ein zu Swing und AWT vergleichbares Framework wird mit SWT angeboten. Den Workspace auf Basis eines Eclipse Plugins zu implementieren, verlangt einen Einsatz von Kenntnissen und Einarbeitungszeit, die eine Verzögerung der Implementationsphase bedeutet hätte. Somit hätten die anderen Projektteile länger auf eine funktionsfähige GUI warten müssen. Diese Gründe waren ausreichend, um sich gegen die Realisierung unseres Workspaces durch ein Eclipse Plugin zu entscheiden.

Charakterisierend für *jABC* [**?**] ist die Verwendung einer graphischen Programmierschicht, in der aus den so genannten SIBs hierarchische Graphen erstellt werden. Service Independent Building Blocks (SIBs) werden durch eine einzelne Java-Klasse repräsentiert. Das jABC Modell kann durch Plugins mit einer Ausführungsschicht angereichert werden, die für die Graphen eine Semantik definieren. Gegen jABC haben wir uns entschieden, weil es aus unserer Sicht nur sehr mäßig dokumentiert war und es sich um ein „closed source“ Produkt handelt, was zum Zeitpunkt der Implementation unvorhersagbare Probleme hätte bedeuten können. Außerdem passen Automatengraphen und SIBs nicht so ganz zusammen. Nichtsdestotrotz kann jABC beliebige Graphen verwalten.

An dieser Stelle haben wir uns dann für eine Java Eigenimplementation entschieden, um unsere vorhandenen Kenntnisse in Java, Swing und AWT ausnutzen zu können. Durch die gesparte Einarbeitungszeit war eine schnelle Umsetzung der wichtigsten Funktionen möglich.

3.4.1.2 Graphenbibliothek

Einen Automaten als Graphen „schön“ zu layouten, ist schon manuell eine relativ schwere Aufgabe. Ein solches Layout automatisch berechnen zu lassen, könnte den Umfang unserer PG übersteigen. Aus diesem Grund haben wir uns zu Beginn der PG mit bereits bestehenden freien Graphlayout-Bibliotheken aueinandergesetzt. Drei Bibliotheken haben da-

bei unser Interesse geweckt. Dieses waren das Package *org.eclipse.draw2d.graph* [?], das *JUNG - Java Universal Network/Graph Framework* [?] und *Grappa* [?]. Mit diesen Paketen haben wir experimentiert, um sie auf ihre Tauglichkeit hin zu überprüfen.

Das Package *org.eclipse.draw2d.graph* ist Bestandteil von Draw2D, welches wiederum ein Teil des Graphical Editing Frameworks von Eclipse ist. Der Algorithmus platziert Knoten und verlegt Kanten möglichst kreuzungsfrei. Dabei lassen sich einige Einstellungen bezüglich der Wichtigkeit von Kanten und Knoten vorgeben. Nach einigen Testimplementationen ist uns aufgefallen, das in dieser Bibliothek keine reflexiven Kanten unterstützt werden. Knoten, die über eine Kante mit sich selbst verbunden sind, lassen den Layout-Algorithmus in eine Endlosschleife laufen. Dies lässt sich beheben, indem man diese Kanten einfach vor dem Layouting entfernt und später wieder hinzufügt. Eine Einschränkung stellt dies nicht dar, reflexive Kanten benötigen schließlich kein automatisiertes Layout. Leider stellte die Anordnung der Knoten einen großen Nachteil der Bibliothek dar: Der Layouter platziert die Knoten eher hierarchisch von oben nach unten. Da Automaten aber mit ihren nicht ungewöhnlichen Zyklen eher keine hierachischen Graphen darstellen, war dieser Layoutalgorithmus für unsere Zwecke leider ungeeignet.

Das *Java Universal Network/Graph Framework* bietet einen Rahmen für die Modellierung, Analyse und die Visualisierung von Graphen aller Art. Es beinhaltet Plugin-Schnittstellen z. B. für Algorithmen oder Layouts, sowie Schnittstellen zu anderen Bibliotheken. Sie ist im Rahmen der BSD-Lizenz frei verfügbar. Bei ersten Tests gelang es recht schnell und einfach, Graphen layouten zu lassen. Um die Tauglichkeit mit Automaten zu analysieren, haben wir die Automatenbibliothek *dk.brics* [?] mit JUNG verknüpft und waren mit den graphischen Ergebnissen zunächst zufrieden.

Grappa ist eine auf Java basierende Graphenbibliothek, die von AT&T entwickelt wurde. Hiermit lassen sich Graphen konstruieren und manipulieren. Diese Bibliothek stellte sich generell leider als Enttäuschung heraus, da diese direkt kein automatisches Graph-Layouting unterstützt, sondern lediglich einen GraphViz-Wrapper darstellt. Ein Graphlayout ist leider nur möglich, indem man die integrierten GraphViz [?] Funktionen verwendet, das heißt, einen Graphen im GraphViz-Dot Format exportiert, mit Hilfe von GraphViz layouten lässt und dann wieder einlädt.

Auf Basis dieser Ergebnisse haben wir uns im ersten Schritt dazu entschieden, JUNG zu verwenden. Es sei aber hier schon einmal vorweggenommen, dass wir uns später wieder davon getrennt haben und JUNG durch eine Eigenimplementation (siehe 3.6) ersetzt haben. Dieses wurde nötig, da ein Bearbeiten von Graphen mit JUNG uns vor größere Probleme stellte.

3.4.2 Das Workspacekonzept

Bei der Strukturierung des Workspaces bilden folgende Elemente die Hauptkomponenten:

1. die Automatenliste
2. die Automatenfenster auf der Arbeitsfläche
3. die Konsole

In der Automatenliste werden alle Automaten eines Projektes in einer Baumstruktur, untergliedert nach ihrem Automatentyp, angezeigt. Beim Anlegen eines neuen Automaten wird dieser in die Baumstruktur eingefügt. Hierbei kann es sich um einen neuen Automaten handeln oder um einen Automaten, der auf Basis einer Automatenoperation entstanden ist. Um Automatenoperationen durchzuführen, werden Automaten in der Automatenliste markiert und durch Auswahl einer möglichen Operation im Aktionsmenü diese dann angewendet. Die Operationen werden bei einer Auswahl eines oder mehrerer Automaten gefiltert. Dadurch stehen jeweils nur erlaubte Operationen zur Verfügung. Als Beispiel bei endlichen Automaten darf z. B. eine Komplementbildung nur bei Auswahl eines Automaten angewendet werden. Werden mehrere Automaten ausgewählt, so steht diese Operation nicht mehr zur Verfügung, dafür aber Vereinigung, Schnitt, etc.

Eine weitere wichtige Aufgabe der Automatenliste besteht in der übersichtlichen Auswahl von Automaten. Klickt ein User einen Automaten an, so rückt dieser in den Vordergrund. Über einen Doppelklick in der Automatenliste kann das entsprechende Automatenfenster maximiert oder auf seine letzte Größe minimiert werden.

Die eigentlichen Informationsträger des Workspaces bilden die Automatenfenster. Sie ermöglichen es, in einer geordneten Tabstruktur, zu transformierende Aufgabenstellungen komfortabel einzugeben, Automaten zu betrachten, Automaten zu bearbeiten, sowie Analyseergebnisse einzusehen. Zum damaligen Zeitpunkt kann man in einem Source-Tab eine Formel der Presburger Arithmetik oder einen regulären Ausdruck eingeben. Diese Formel wird durch die Aktion *Create Automaton* transformiert und der Ergebnisautomat in einem Automaten-Tab angezeigt.

Das letzte Element des Workspace bildet die Konsole. In ihr werden alle Aktionen, Informationen und Fehler dokumentiert und angezeigt, so dass ein User diese zu jeder Zeit noch einmal nachlesen kann. Die Konsole ist zusätzlich mit Funktionen zum Löschen und Kopieren des Konsoleninhalts ausgestattet. So können wichtige Inhalte einfach und schnell exportiert werden.

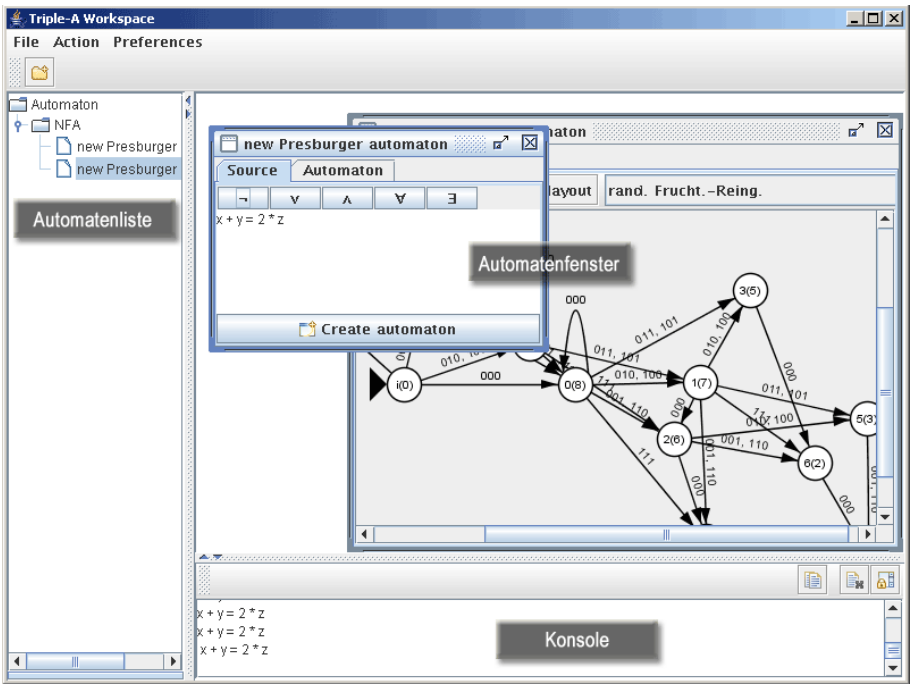


Abbildung 3.25: Triple-A Workspace

3.4.3 Features der Implementation

An dieser Stelle gehen wir auf die Implementation des Workspace ein und beleuchten die Hintergründe und ggf. aufgetretene Probleme und deren Lösungen. Das Ziel dieses Abschnittes ist es, den Leser in die Lage zu versetzen, unter Zuhilfenahme des Codes die Workspace-Implementation und seine Hintergründe zu verstehen.

3.4.3.1 Der Workspace

Abbildung 3.26 zeigt ein abstraktes Klassendiagramm des Workspace. Dieses Schaubild ist nicht vollständig, um es übersichtlich zu halten. Es enthält aber alle nennenswerten Hauptkomponenten des Workspace, deren Funktionen im Weiteren kurz erläutert werden. Einige Punkte werden auch später in diesem Abschnitt noch einmal aufgegriffen, um deren Funktionalität genauer zu erläutern.

WorkspaceControl: Die Klasse `WorkspaceControl` bildet die zentrale Kontrollinstanz des Workspace. Wird diese Klasse instanziiert, so erzeugt sie ein Objekt der Klasse `WorkspaceFrame` und initialisiert somit den Workspace. Ihre weiteren Kontrollaufgaben bestehen darin, alle statischen Aktionen des Workspace zu verwalten. Diese statischen Aktionen bilden innere Klassen und sind in einer `HashMap` abgelegt. Es sind folgende statische Aktionen implementiert:

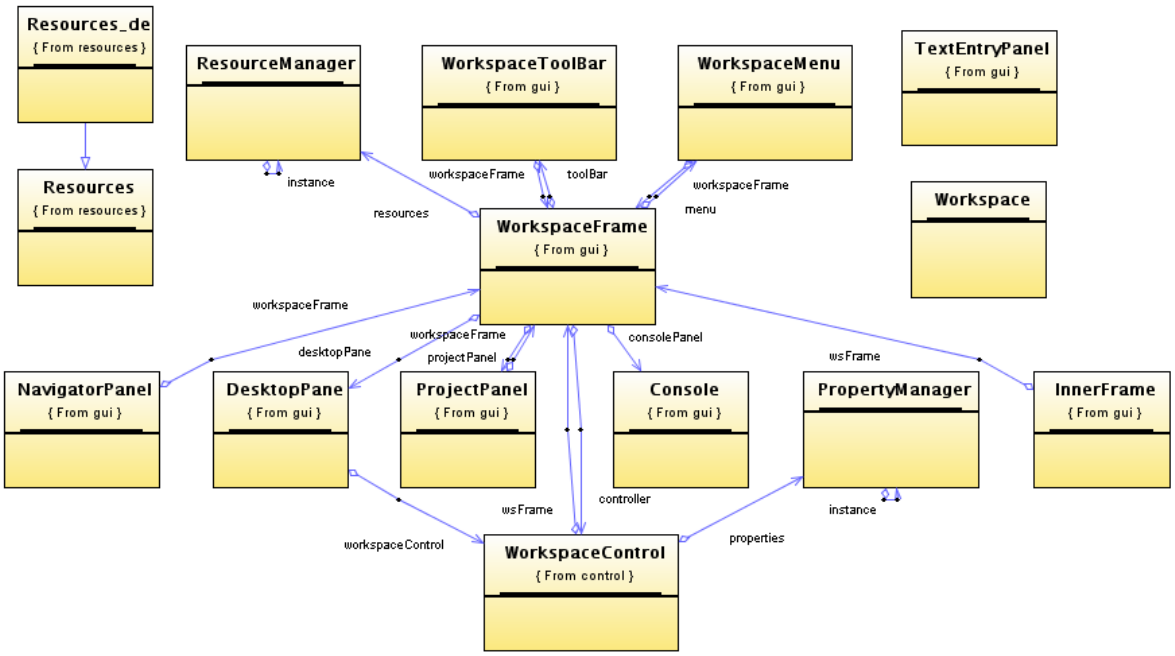


Abbildung 3.26: Klassenübersicht des Workspaces

- den Workspace schließen
- ein neues Projekt anlegen
- das Look&Feel des Workspace ändern
- eine Sprache wählen

Zusätzlich kümmert er sich um die Instanziierung und Verwaltung der Addonmanager, die im Abschnitt Addons näher erklärt werden.

WorkspaceFrame: Der Triple-A Workspace wird zentral durch die Klasse `WorkspaceFrame` gezeichnet. Sie ist von der Klasse *`javax.swing.JFrame`* abgeleitet und bildet das Hauptfenster des Workspace. Im Workspace werden folgende Komponenten initialisiert:

WorkspaceMenu: Diese Klasse bildet das Menü des Workspace, in dem Aktionen abgelegt werden können. Beim Initialisieren des Menüs werden alle statischen Aktionen aus der Klasse `WorkspaceControl` ins Menü aufgenommen. Außerdem enthält die Klasse Methoden, mit denen dynamisch Einträge dem Menü hinzugefügt werden können. Diese Methoden bilden eine Schnittstelle für die Addons, um sich beim dynamischen Laden ins Menü einzutragen. Es gibt zwei Schnittstellen im Menü:

- `addFileNewItem(JMenuItem item)`: Diese Methode ermöglicht, neue Transformationsverfahren dem Menü hinzuzufügen.

- `addFileActionItem(JMenuItem item)`: Über diese Methode können Automatenaktionen ins Menü eingetragen werden.

Die WorkspaceToolBar: Der Workspace verfügt über eine Toolbar, über die schnell und bequem Aktionen aufgerufen werden können. Diese Toolbar wird nur zum Anlegen von neuen Projekten verwendet.

DesktopPane: Die `DesktopPane` bildet die zentrale Arbeitsfläche des Workspace. In ihr können beliebig viele `InnerFrame`-Objekte angelegt werden. Sie ist abgeleitet von der Klasse *javax.swing.JDesktopPane*.

InnerFrame: `InnerFrame`s wurden im vorherigen Abschnitt als Automatenfenster bezeichnet. Hierbei handelt es sich um *javax.swing.JInternalFrame*-Objekte. Ihre Aufgabe besteht darin, Transformations- und Analyseergebnisse anzuzeigen. Dazu verfügt ein `InnerFrame` über eine geordnete Tabstruktur, abgeleitet von *javax.swing.JTabbedPane*.

ProjectPanel: Auf der DesktopPane können eine Vielzahl von InnerFrames erzeugt werden. Das ProjectPanel dient zur strukturierten Anzeige aller vom Benutzer erzeugten InnerFrames. Die Strukturierung erfolgt nach Typ des Automatenergebnisses. Das ProjectPanel erfüllt außerdem die Aufgabe, ein Fenster schneller aufzufinden und Automaten markieren zu können, um auf diese Operation anzuwenden. Beim ProjektPanel handelt es sich um ein *javax.swing.JTree*-Objekt. Dieser `JTree` ist so programmiert, dass ein Baum der Tiefe drei entsteht. Die Wurzel des Baums bildet der aktuelle Projektname. Unter der Wurzel gibt es Verzweigungsknoten, die die Strukturierung der Innerframes nach Automatentypen leisten. Auf der Blattebene entsteht pro Innerframe ein Knoten mit Innerframebezeichnung. Diese Strukturierung ist im Package *aaa.workspace.gui.projecttree.ProjectTree* implementiert.

Console: Die Klasse `Console` ist eine Eigenentwicklung, die es möglich macht, alle ausgeführten Aktionen, Fehlermeldungen und Informationen vom Workspace zu dokumentieren und anzuzeigen. Dabei ist die Konsole so einfach zu verwenden wie ein `System.out-PrintStream`. Sie ist zusätzlich noch mit drei weiteren Funktionen ausgestattet. Zum einen kann der Inhalt mit einem Buttonklick in die Zwischenablage kopiert werden, die Konsole kann gelöscht werden und als besonderes Feature kann das Scrollen in der Konsole aktiviert oder deaktiviert werden. Das heisst, man kann bestimmen, ob die Konsole bei neuen Einträgen automatisch scrollt oder an ihrer aktuellen Position verweilt. Die Konsole ist ein *javax.swing.JPanel*, in dem eine *javax.swing.JTextArea* eingebunden ist. In diese *javax.swing.JTextArea* kann über den `PrintStream Console.out` einfach ein Eintrag hinzugefügt werden.

TextEntryPanel: Um eine Transformation einer Eingabe durchzuführen, bietet die

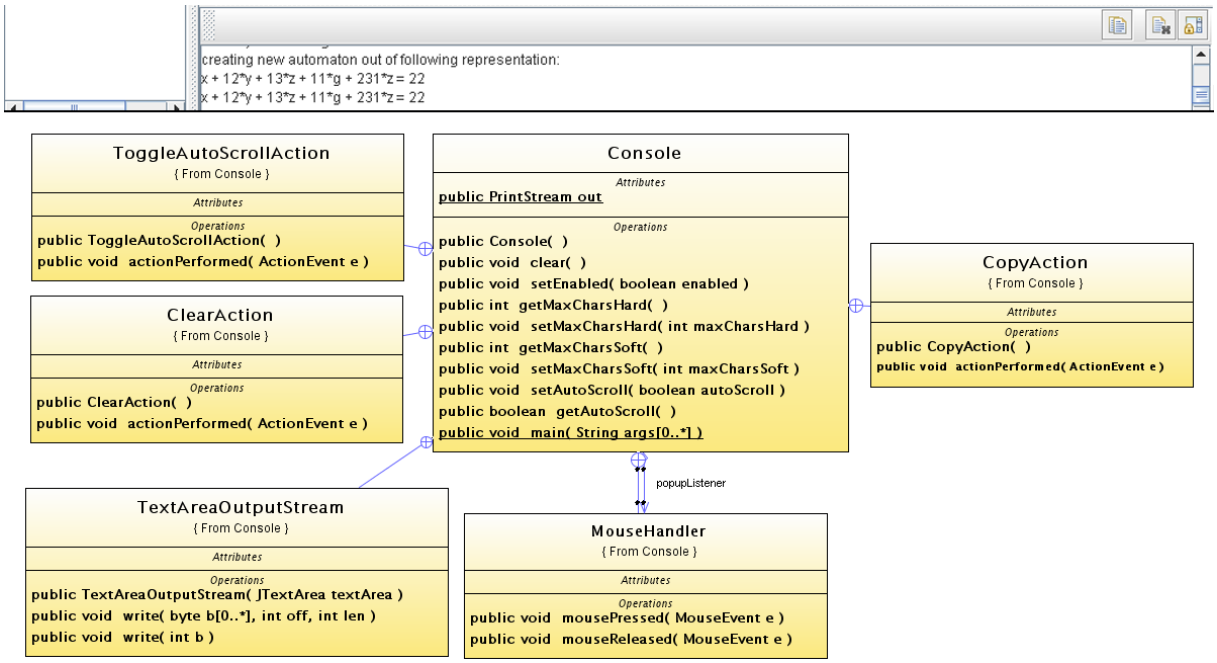


Abbildung 3.27: Die Klasse `Console`

Klasse `TextEntryPanel` eine komfortable Möglichkeit. Es besteht aus drei Komponenten. Es kann eine Buttonleiste mit Sonderzeichen gefüllt werden, die für die Eingabe einer Sprache nötig sind. Im Konstruktor des TextEntryPanels kann eine *`java.util.LinkedList`* mit *`java.lang.Character`*-Objekten übergeben werden. Diese werden dann als Sonderzeichen zur Verfügung gestellt. Wird der leere Konstruktor oder eine leere Liste übergeben, so wird diese nicht angezeigt. Unter der Sonderzeichenleiste befindet sich ein *`javax.swing.JTextArea`* in der die gewünschte Transformationsformel eingegeben werden kann. Das letzte Element des Dialogs bildet ein *`javax.swing.JButton`*, mit dem eine Transformation gestartet wird. Da diese Aktion flexibel gehalten werden muss, ermöglicht die Methode `setOkButtonAction(Action action)` die Transformationsaktion zu verändern. So kann dieser Dialog für mehrere Transformationsverfahren genutzt werden.

PropertyManager: Oft hat ein User den Wunsch, seine graphische Arbeitsumgebung individuell auf sich persönlich zugeschnitten einzurichten. Dieser Wunsch kann durch den Workspace erfüllt werden. Der Workspace ist mit der Klasse `PropertyManager` ausgestattet, die es erlaubt, Einstellungen und Zustände des Workspaces zu speichern und beim erneuten Aufruf des Workspaces wieder einzulesen. Diese Benutzerinformationen werden im Hauptverzeichnis des Workspaces in der Datei *`properties.cfg`* gespeichert. **Datei Properties.cfg**

```
#Triple-A properties file
#Sat Jan 28 15:18:04 CET 2006
locale=en
```

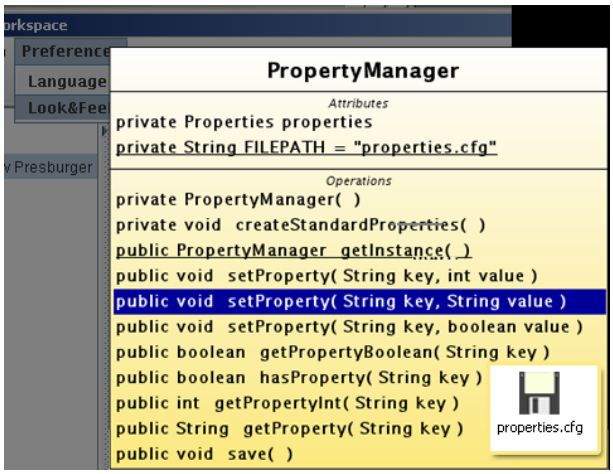


Abbildung 3.28: Die Klasse `PropertyManager`

```
LookAndFeel=
    javax.swing.plaf.metal.MetalLookAndFeel
```

Zum gegenwärtigen Zeitpunkt unterstützt der Workspace zwei Eigenschaften: die Speicherung der Spracheinstellung und ein individuelles Look&Feel.

Je nach gewählter Spracheinstellung wird der Workspace in der gewünschten Sprache gestartet.

Java unterstützt es, das Erscheinungsbild und das Verhalten der GUI mittels sogenannter Look&Feels anzupassen. Auf allen Plattformen stehen sowohl das Motif-Look-and-Feel als auch das Metal-Look-and-Feel zur Verfügung, auf Mac und Windows zusätzlich ein dem systemüblichen angepasstes eigenes Look&Feel. In den Eigenschaften wird das vom User bevorzugte Look&Feel gespeichert und beim Start des Workspaces verwendet. Der PropertyManager ist unter Zuhilfenahme des *java.util.Properties*-Konzeptes implementiert.

RessourceManager: Um viele interessierte Nutzer mit unserem Projekt erreichen zu können, haben wir den Workspace sprachunabhängig programmiert. Dieses Ziel wird effizient erreicht, indem wir auf das *ResourceBundle*-Konzept von Java zurückgegriffen haben. Bei diesem Konzept werden alle angezeigten Texte, Informationen und ggf. Bilder durch einen symbolischen Namen in der Programmierung ersetzt. Diese Namen werden im Workspace in sogenannten Resource-Klassen definiert und übersetzt. Das heißt, in einer Resource-Klasse werden einem symbolischen Namen ein länderspezifischer Text oder eine Bilddatei zugeordnet. Dabei wird die passende Resource-Datei automatisch aufgrund der im Workspace eingestellten Spracheinstellung identifiziert. Die Resourcedateien unterscheiden sich durch eine Erweiterung des Dateinamens um eine Landeskennung. Diese Kennung gibt

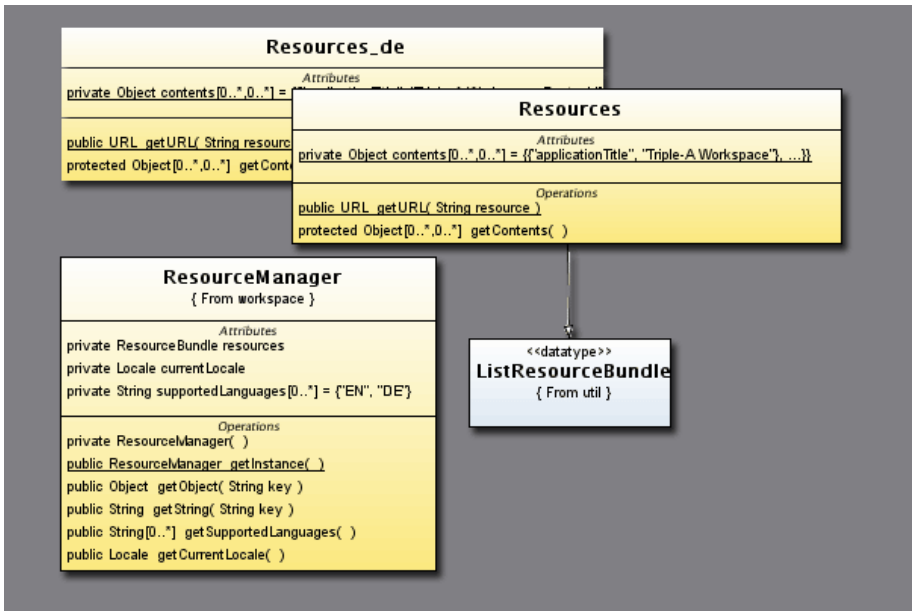


Abbildung 3.29: ResourceManager

an, welche Sprache vorliegt. Z. B. steht resource_de.class für eine deutschsprachige Resource-Datei. Die Übersetzung findet zentral durch den ResourceManager statt.

3.4.3.2 Addons

Um eine zukünftige flexible Erweiterung des Workspaces mit neuen Automatentypen, neuen Transformations- und Analyseverfahren zu gewährleisten, wurde ein Addon-Konzept eingeführt. Mit dieser Funktionalität soll der Workspace dem Konzept eines leicht erweiterbaren Baukastens entsprechen, da durch das Hinzufügen von Addons keine Änderungen am Workspace vorgenommen werden müssen. Somit sind die Funktionen der Addons streng gekapselt und fügen dynamisch ihre Funktionalität dem Workspaces hinzu.

Elemenente des Addonkonzepts: Das Diagramm *Addons in AAA* zeigt alle wichtigen Elemente der Addons und ihr Zusammenspiel mit den anderen Komponenten des Systems: der Automaten, des Parsers und der Transformation. An dieser Stelle werden alle Elemente der Addons beschrieben.

- Die Klasse `AddonWorkspacePreprocess` ist eine abstrakte Klasse, von der alle Addon-Initialklassen abgeleitet werden müssen. Dieses Vorgehen hat den Vorteil, das alle Addons einen gemeinsamen Konstruktor und eine Methode `init(WorkspaceFrameworkspaceFrame)` besitzen. So erhalten alle Addons eine einheitliche Struktur, die zum Initialisieren eines Addons benötigt wird. Die Methode `init(WorkspaceFrame workspaceFrame)` muss in einem Addon so

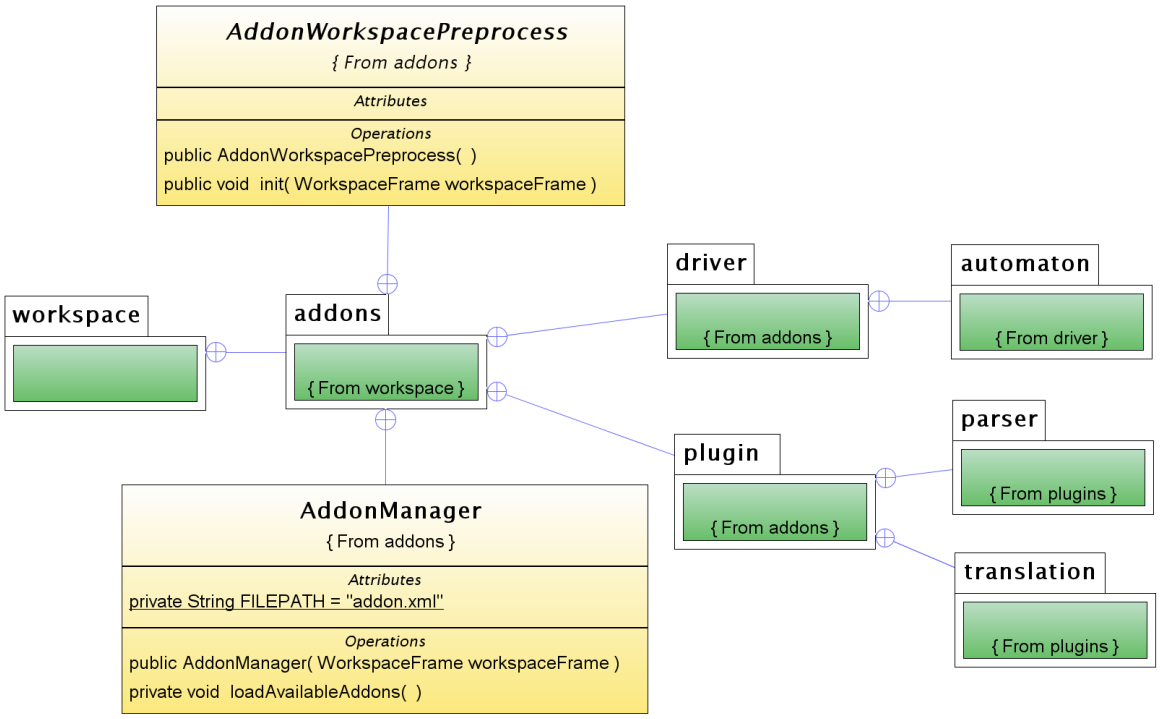


Abbildung 3.30: Addons in AAA

implementiert werden, das sie das Laden des Addons anstößt. So wird garantiert, dass alle Addons auf die selbe Art und Weise dynamisch geladen werden können.

```
package aaa.workspace.addons;

import aaa.workspace.gui.WorkspaceFrame;

public abstract class AddonWorkspacePreprocess {
    protected WorkspaceFrame workspaceFrame;

    public AddonWorkspacePreprocess() {
    }

    public void init(WorkspaceFrame workspaceFrame){
       this.workspaceFrame = workspaceFrame;
    }
}
```

- Die abstrakte Klasse `AddonManager` bildet die Basis für alle weiteren Addon-

Manager. Diese müssen von der abstrakten Klasse abgeleitet werden. Der Grund liegt in der Methode `loadAvailableAddons()`. Diese Methode wird beim Instanziieren des Addon-Managers gestartet und hat die Aufgabe, alle vorhandenen Addons zu ermitteln und dynamisch zu laden. Auf die genauere Funktionsweise kommen wir später noch einmal zurück.

- Einer der zwei implementierten Addontypen wird **Driver** genannt. Driver haben die primäre Aufgabe, eine Automatenbibliothek, die ohne Gedanken an eine GUI konzipiert wurde, in den Workspace einzubinden. Eine einzubindende Bibliothek wird also per Treiber angesprochen und es gibt pro Automatentyp genau einen. Die Driver erfüllen folgende Aufgaben:
  1. Ganz trivial kennt er seine zugehörige Automatenbibliothek und deren aktuellen Standort.
  2. Ein Driver ist in der Lage, einen Automaten im Workspace anzuzeigen und weiß, wie und ob ein Automat graphisch bearbeitet werden kann.
  3. Er stellt Methoden zur Verfügung, um Automaten zu speichern und später wieder einladen zu können.
  4. Er stellt alle Automatenoperationen zur Verfügung und filtert sie. Das heißt, wird z. B. genau ein Automat markiert, so werden nur unäre Operationen im Workspacemenü angezeigt, die zu diesem Automatentyp kompatibel sind.
  5. Er verwaltet alle Analyse-Funktionen und stellt zum Beispiel für endliche Automaten einen Akzeptanz-Test von Wörtern zur Verfügung sowie für endliche Automaten und Büchiautomaten einen Test auf Leerheit der erkannten Sprache.
- Nun braucht der Workspace noch eine Art Addon, die sich um die Einbindung der Transformationverfahren kümmert. Diese Aufgabe wird durch **Plugins** erfüllt. Die Aufgaben eines Plugins sind:
  1. Das Plugin kennt alle benötigten Elemente, wie die Transformationsfunktionen und Parser.
  2. Es können Transformationsverfahren im Workspace zur Verfügung gestellt werden, die über einen „File - New“-Menüeintrag ausgewählt werden können.
  3. Es wird ein Editor zur Eingabe eines Ausdrucks oder einer Formel zur Verfügung gestellt.
  4. Das Plugin kann aus einer eingegebenen Formel eine vollständige Transformation durchführen, deren Ergebnis ein Automat ist. Dieser Automat wird dann an den Driver weitergeleitet.

- Es gibt einen **Driver- und einen Pluginmanager**, der die Schnittstelle zwischen Addons und Workspace bildet. Im ersten Schritt laden sie alle verfügbaren Addons. Der Drivermanager enthält zusätzlich Funktionen, mit denen der Name eines Automatentyps bei Übergabe eines Automaten ermittelt werden kann.
- In der Datei `addon.xml` können einige Informationen abgelegt werden, die später beim Einladen des Addons verwendet werden können. Wichtig für das dynamische Einlesen sind im Moment die Schlüssel `name` und `class`. Der Schlüssel `name` wird beim Laden des Addons in der Konsole angezeigt und informiert somit einen Benutzer beim Start, welche Addons seinem Workspace hinzugefügt worden sind. Die Variable `class` wird zum dynamischen Klassenladen der Addoninitialklasse benötigt. Diese Klasse ist von der oben beschriebenen Klasse `AddonWorkspacePreprocess` abgeleitet.

  Datei addon.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM
    "http://java.sun.com/dtd/properties.dtd">
<properties>
<comment>Presburger Plugin</comment>
<entry key="name">Pluginname</entry>
<entry key="version">version 0.1</entry>
<entry key="class">
aaa.workspace.addons.plugin.nameplugin.initclass
</entry>
</properties>
```

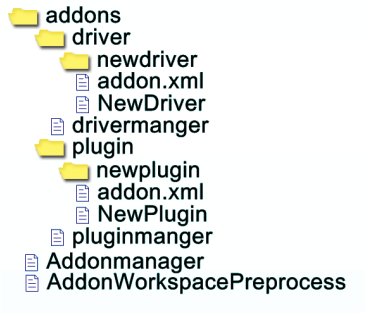


Abbildung 3.31: Die Addonverzeichnisstruktur

Wie werden Plugins dynamisch geladen: Eine gültige Pluginstruktur sieht folgendermaßen aus: Der Workspace besitzt ein Unterverzeichnis Addons. In diesem be-

finden sich die Verzeichnisse Plugin und Driver. Ein neues Addon wird in einem neuen Unterverzeichnis unter Driver oder Plugin abgelegt, ein neuer Driver unter Driver und ein neues Plugin unter Plugin. In diesem Verzeichnis werden mindestens zwei Dateien benötigt. Zum einen eine Initialklasse, die von der Klasse `AddonWorkspacePreprocess` abgeleitet ist und eine `addon.xml`, die angibt, wo diese Datei liegt.

Die Addonmanager durchsuchen nun alle direkten Unterverzeichnisse nach einer `addon.xml` Datei und laden per dynamischem Klassenladen die darin enthaltene Initialklasse.

Integration von Addons im Workspace: Zum gegenwärtigen Zeitpunkt haben wir viel über das Integrieren von Addons gesprochen. Die große Frage ist nun, welche Funktionen der Workspace unterstützen muss, damit in ihm einfach Addons integriert werden können.

Aktionen: Im Workspace gibt es eine Möglichkeit, Tranformationsverfahren neu zu starten und auf Ergebnisautomaten Operationen durchführen zu können. Diese Aktionen können über die Klasse `WorkspaceMenu` in das Menü eingefügt werden. Die Funktion wurde bereits oben erklärt.

Automatenfenster: Der WorkspaceFrame liefert Methoden, um ein leeres Automatenfenster anzulegen. Ein Plugin füllt dieses im ersten Schritt mit einem Panel, um darin eine zu transformierende Aufgabenstellung einzugeben und eine Aktion, die die Transformation startet. Wichtig beim Erzeugen eines neuen Automatenfensters ist die Einordnung des neuen Fensters in die Automatenliste. Hierzu ermittelt das Plugin den Typ des Ergebnisautomaten und erfragt beim Drivermanager den Namen des Automatentyps. Unter diesem zurückgelieferten Namen ordnet das Plugin das neue Automatenfenster in die Automatenliste ein. Für die Eingaben in Plugins steht die Klasse `TextEntryPanel` mit allen ihren bereits beschriebenen Funktionen zur Verfügung. Sollten diese nicht ausreichen, so kann in einem Plugin ein neues Eingabepanel programmiert werden. Dieses kann wie das `TextEntryPanel` einem `InnerFrame` als Tab hinzugefügt werden. Durch Starten einer Transformationsaktion wird die Eingabe umgewandelt und an den Driver übergeben.

Anzeige von Automaten: Für die Anzeige eines Ergebnisautomaten ist der Driver zuständig. Über den Drivermanager wird der Automatentyp ermittelt und mit Hilfe des zugegehörigen Automatendrivers in einem neu angelegten Tab angezeigt. Der Driver weiß also, wie er den Automaten an den Editor übergeben muss, so dass dieser korrekt dargestellt und layoutet wird.

Operationen: Werden ein oder mehrere Automaten in der Automatenleiste markiert, so müssen die auf diesen Automaten gültigen Operationen ermittelt werden. Auch darum kümmert sich der Driver. Er baut dynamisch die passenden Aktionsmenüs auf. Wird eine Automatenoperation angewendet, so führt der Driver diese durch.

Speichern/Laden: Wie ein Automat gespeichert oder geladen wird, ist ebenfalls Aufgabe des Drivers.

3.5 Der Workspace - Stand Juli 2006

Nachdem im ersten Semester die Hauptarbeit darin bestand, ein grundlegendes Framework und ein flexibles Addon-Konzept zu erstellen, so stand dieses Konzept im zweiten Semester auf dem Prüfstand und wurde mit diversen neuen Analyseverfahren getestet.

Weiterhin musste das Driver-Konzept ersetzt werden, welches am Ende des ersten Semesters noch nicht vollständig ausgereift war. Das neue Konzept macht den Workspace noch flexibler, so dass der Workspace mit einer Vielzahl neuer Features abgerundet werden konnte.

3.5.1 Planungsphase

Vor und während des zweiten Semesters fielen neue Anforderungen an, die im Weiteren kurz angesprochen werden. Inwieweit diese Anforderungen umgesetzt werden konnten, wird in den nächsten Abschnitten ebenfalls behandelt.

In der Interaktion zwischen Anwendern, den Automaten und den Transformationsverfahren sollte der Workspace folgende neue Features unterstützen:

- Alle Methoden, die von der Automatenbibliothek zur Verfügung gestellt werden, sollten per Reflection in das Workspace-Aktionenmenü eingebaut werden. Um bessere Lesbarkeit zu erreichen, könnte man Übersetzungen der Methodennamen in einer Textdatei vorhalten. Fehlt eine Übersetzung, zeigt man einfach den schlichten Methodennamen an. So ist zumindest gewährleistet, dass neue Operationen sofort unterstützt werden.
- Methoden mit dem Präfix `render` sollten so benutzt werden, wie sie gemeint sind, d. h. ruft man sie auf einem Automaten auf, verändert sich dieser selbst, und es wird kein neues Fenster geöffnet, in dem ein neuer Automat erscheint. Dies soll nur bei `generating` Operationen passieren.
- Treten bei der Anwendung von Automatenoperationen Fehler auf, so müssen diese an einen Anwender verständlich weitergeleitet werden.
- Ab einer gewissen Größe kann man mit der graphischen Darstellung des Automaten nichts mehr anfangen. Es sollte dann zumindest möglich sein, gewisse Daten abzurufen, wie die Anzahl der Zustände, Anzahl der Transitionen, Initialzustände, akzeptierende Zustände usw.
- Bei sehr großen Automaten ist die graphische Darstellung nicht nur so gut wie nutzlos, sondern auch hinderlich. Sie verbraucht so viele Ressourcen, dass der Workspace

unbedienbar wird. Darum sollten Automaten ab einer bestimmten Grösse nur auf Anforderung des Anwenders gezeichnet werden.

- Da Operationen wie das Layouten von Automaten sehr viel Rechenzeit in Anspruch nimmt, sollten solche Berechnungen nicht den Workspace blockieren, sondern auf einem eigenen Thread ausgeführt werden.
- Die Automatenliste könnte Operationen per Drag&Drop oder über ein Kontextmenü unterstützen.
- Man sollte Einträge in der Automatenliste umbenennen können.
- Für einige Automatenoperationen ist es wichtig, die Reihenfolge der gewählten Automaten zu erkennen und zu beeinflussen.
- Analyseoperationen, wie das Testen einer Eingabe auf endlichen Automaten oder das Auffinden einer akzeptierenden Belegung sollten implementiert werden.
- Der Zustand ausgewählter Automaten sollte gespeichert werden können. Hierbei ist es wichtig, dass keine Informationen verloren gehen, wie z. B. die Basiseingabe, aus der der Automat erzeugt wurde, der Automat selbst oder das Layout. Gespeicherte Daten müssen natürlich auch wieder eingeladen werden können.
- Für das Model Checking werden neue Darstellungs-Modelle benötigt, die das Erstellen, Bearbeiten und Layouten von Büchiautomaten, Kripkestrukturen und OBDDs ermöglichen.
- Der Workspace sollte Möglichkeiten zum Editieren von Automaten bieten, so dass diese auch ohne eines der Transformations-Module erstellt werden können. Darunter fällt z. B. die Erstellung von Kripkestrukturen für das Model Checking.
- Es werden diverse neue Plugins benötigt, die alle neuen Automatenmodelle und Transformationen unterstützen.

3.5.2 Das Workspacekonzept

Das Workspacekonzept und seine Hauptkomponenten haben sich im zweiten Semester sehr stark verändert und wurden durch wichtige neue Elemente erweitert. Die wohl auffälligste Neuerung fällt direkt beim Start des Workspace auf: die Dreiteilung des Workspace ist durch eine Vierteilung abgelöst worden. Unterhalb der Automatenliste befindet sich nun ein neuer Eigenschaftenbereich. In diesem Bereich können dynamisch Informationen und Eigenschaften eingeblendet werden. So können z. B. Automateninformationen, wie Anzahl der Zustände, Anzahl der Transitionen, Initialzustände, akzeptierende Zustände oder

Transitionen von Automaten dort angezeigt und sogar bearbeitet werden. So werden unnötige Dialoge minimiert, die den Arbeitsfluss stören könnten.

Als nächstes Element sind Änderungen in der Automatenliste zu erwähnen. So wird diese nicht mehr nach Automatentyp kategorisiert, sondern nach zugehörigem Plugin. Dies unterstützt die Unterschiedlichkeit der einzelnen Automaten untereinander, wie Alphabete, Sonderfunktionen oder Editiermöglichkeiten. Beispiele hierfür sind Automaten auf Basis von regulären Ausdrücken oder Presburger-Formeln. Die Ergebnisse beider Transformationen sind endliche Automaten mit abweichenden Alphabeten. Bei regulären Ausdrücken wird ein CharacterAlphabet und bei Presburger-Formeln ein AssignmentAlphabet verwendet. Einige Automatenoperationen sind aus diesem Grund nicht möglich und würden zu diversen Fehlern führen, z. B. ein Schnitt dieser beiden Automaten.

Optisch wurde die Automatenliste stark überarbeitet. So wird nun die Selektions-Reihenfolge der vom Benutzer markierten Automaten angezeigt. Das kann unter anderem sehr hilfreich sein bei Operationen, die auf mehreren Automaten angewendet werden, z. B. ist es beim Konkatenieren wichtig, die korrekte Reihenfolge der Automaten festlegen zu können.

Auch wurde stark an der Übersichtlichkeit der Automatenliste gearbeitet. Ein Anwender kann nun Automaten in der Liste umbenennen und sie so mit sprechenden Namen versehen. Doppelte Einträge werden gefiltert und durch eine angehängte Zahl eindeutig gemacht. So wird das Arbeiten mit einer Vielzahl von Automaten deutlich vereinfacht.

Auch die Konsole blieb nicht von Veränderungen verschont. So bildet diese nun nur noch einen Teil des neuen Analysebereiches. Dieser Bereich ist geschaffen worden, um in ihm spezielle Automatenanalysen anzuzeigen. Hier wären z. B. das Testen einer Eingabe auf Akzeptanz durch den Automaten oder das Auffinden einer solchen akzeptierenden Belegung zu erwähnen. Diese beiden Funktionen für endliche Automaten sind ebenfalls im zweiten Semester neu entstanden.

Wegen mangelnder Nutzung von Properties und Mehrsprachigkeit wurde deren Entwicklung im zweiten Semester nicht mehr weitergeführt. Die bereits implementierten Funktionen sind noch im Workspace vorhanden und können ggf. in einer weiteren Projektgruppe wieder aktiviert und weitergeführt werden. Auch auf eine im ersten Semester erwähnte Projektverwaltung wurde verzichtet, zumal diese durch das Speichern und Laden von mehreren Automaten in einer XML-Datei auch überflüssig geworden ist.

3.5.3 Features der Implementierung

Änderungen und Neuerungen in der Implementation des Workspace werden hier ergänzend zum gleichnamigen Abschnitt des ersten Semesters beschrieben, siehe Abschnitt 3.4.3. Einzig das Pluginkonzept erhält einen neuen Abschnitt.

3.5.3.1 Workspace

WorkspaceControl: Der Klasse `WorkspaceControl` sind neue statische Methoden hinzugefügt worden:

SaveAction Diese Aktion startet einen Prozess, der alle markierten Automaten an einer ausgewählten Stelle im XML-Format speichert. Die dazugehörige Aktion in der Klasse `WorkspaceControl` ist eine Art Starter dieser Funktion. In der `WorkspaceControl` wird über einen `FileDialog` ein Zielpfad und ein Zieldateiname ausgewählt. Diese Kombination wird an den Pluginmanager durch Aufruf der Methode `pluginManager.saveWorkspace(`*`Pfad + Dateinname`*`)` übergeben. Der Pluginmanager kümmert sich im Weiteren darum, dass die markierten Automaten dort gespeichert werden.

LoadAction Analog zur `SaveAction` kann über einen `FileDialog` eine XML-Datei gewählt werden. Diese wird an den Pluginmanager übergeben, der den Inhalt wiederherstellt. Das ganze passiert über den Aufruf der Methode `pluginManager.loadFromDisk(`*`Pfad + Dateinname`*`)`.

WorkspaceMenu: Eine neue Schnittstelle wurde in dieser Klasse implementiert. So können über die Methode `addAnalysisMenuItem(JMenuItem item)` Analyseaktionen dem Workspace dynamisch hinzugefügt werden.

WorkspaceToolBar: Die Toolbar wurde aus dem Projekt entfernt, da alle wichtigen Aktionen bereits im WorkspaceMenu untergebracht sind und eine selektive Anzeige einiger Akionen in der Toolbar nicht sinnvoll erschien.

3.5.3.2 Analysepanel

Das Analysepanel verfügt, wie ein TextEntrypanel, ebenfalls über eine geordnete Tabstruktur, abgeleitet von *`javax.swing.JTabbedPane`*. Diese macht es möglich, mehrere verschiedene Analysetabs dynamisch einzubinden. Die Konsole ist ein fester Bestandteil des Analysepanels und ist dort als erster Tab eingefügt. Über die Methode `addTab()` lassen sich neue Panels einbinden. Mit der Methode `remove()` können Tabs entfernt werden.

3.5.3.3 TextEntryPanel

Die Flexibilität der Klasse `TextEntryPanel` hat sich bewährt, sie wird nun in vielen Plugins als Eingabemöglichkeit genutzt. Aus diesem Grund wurde sie im zweiten Semester noch komfortabler gestaltet. So erscheint nun bei Drücken der rechten Maustaste ein Kontextmenü mit folgenden Einträgen:

Copy Diese Aktion fügt den aktuell markierten Inhalt des TextEntryPanels in die Zwischenablage ein. Hierbei wird auf die Methode `copy()` einer `JTextArea` zurückgegriffen.

Paste Analog zum Copy fügt diese Aktion den Inhalt der Zwischenablage an der aktuell markierten Stelle des TextEntryPanels ein, unter Verwendung der Methode `paste()` einer `JTextArea`.

Load source Mit Hilfe eines `FileReader` wird der Inhalt des TextEntryPanels durch den Inhalt einer angegebenen Textdatei ersetzt.

Save source Diese Funktion speichert den aktuellen Inhalt des TextEntryPanels in einer angegebenen Textdatei mit Hilfe eines `FileInputStreams`.

3.5.3.4 PropertyPanel

Dieses Panel besteht aus zwei Komponenten. Zum einen aus der Klasse `PropertyPanel` und zum anderen aus dem Interface `PropertypPanelUser`.

Das Interface `PropertyPanelUser` muss von allen Klassen implementiert werden, die das PropertyPanel nutzen wollen. Dieses Interface beinhaltet einen Methoderumpf für die Methode `lostPanel()`.

Die Klasse `PropertyPanel` erweitert ein `javax.swing.JPanel` und erhält seine Funktionalität durch die folgenden zwei Hauptmethoden `grabPanel()` und `releasePanel()`. Das Vorgehen wird an dieser Stelle einmal beispielhaft skizziert. Nehmen wir die folgende Situation an: Im PropertyPanel wird gegenwärtig ein Panel des aktuellen Automaten angezeigt und durch eine Editor-Aktion soll z. B. ein Transitionspanel im PropertyPanel angezeigt werden. Dies geschieht, indem vom Editor die Methode `grabPanel(PropertypPanelUser, JPanel)` mit dem Editor als PropertyPanelUser und dem Transitions-Panel als weiteren Parameter aufgerufen wird. Der vorherige Nutzer des PropertyPanels wird durch Aufruf der Methode `lostPanel()` darüber informiert, dass er nicht mehr der Besitzer des Panels im PropertyPanel ist. Er kann dann auf den Verlust des Panels reagieren und z. B. nicht mehr benötigte Ressourcen freigeben oder sich von

Listenern abzumelden. Diese einfache Struktur ermöglicht eine schnelle Implementierung. Mit der Methode `releasePanel(PropertyPanelUser)` kann der aktuelle User des Panels es abgeben, ohne dass es von einem anderen Panel erfragt wurde.

3.5.4 Plugins

Durch die Entwicklung der Klasse `GenericOperationDriver` ist eine Trennung von Addons in Plugins und Driver überflüssig geworden. Aus diesem Grund beschäftigen wir uns im Weiteren nur noch mit Plugins, da ihre Funktionalität und ihr Aufbau im zweiten Semester erhalten geblieben sind. Die Funktionen eines ehemaligen Driverplugins werden über den `GenericOperationDriver` und ein automatenspezifisches Plugin ersetzt.

Die Basiskomponenten des überarbeiteten Pluginkonzepts bilden die Klassen `GenericOperationDriver`, `Plugin` und der `PluginManager`.

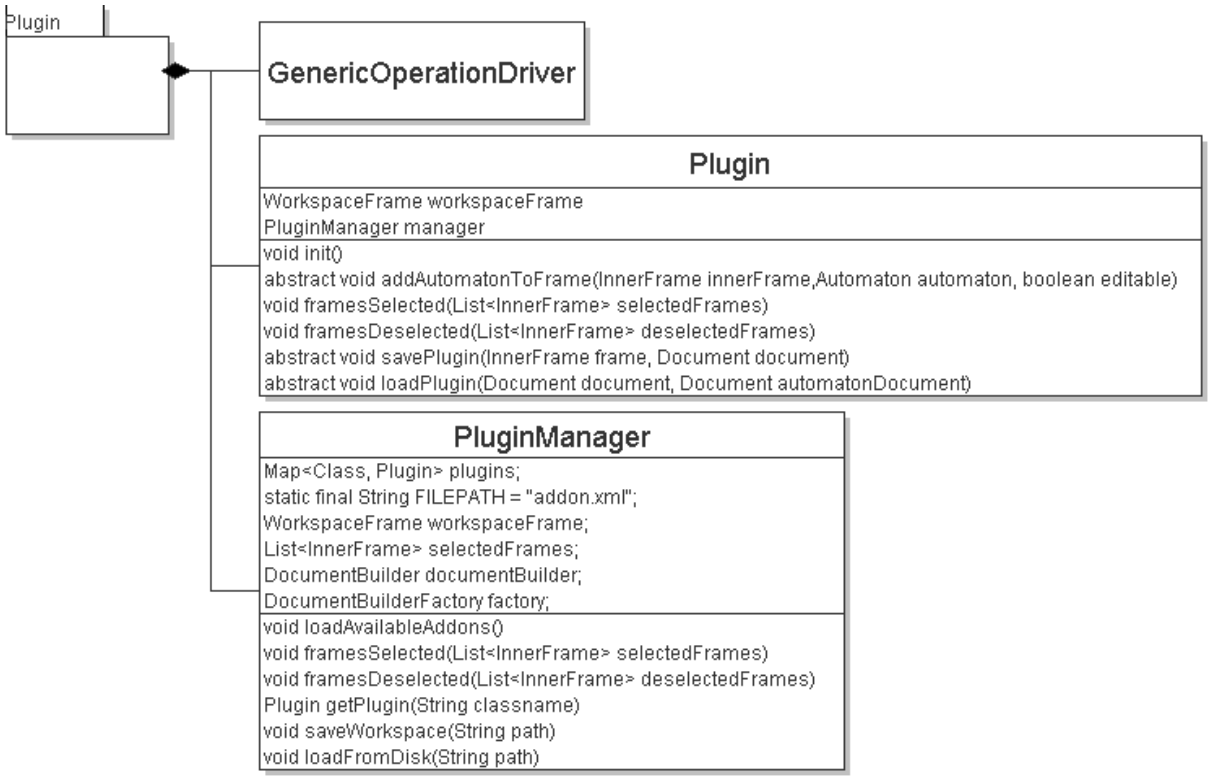


Abbildung 3.32: Das Grundkonstrukt der Plugins und die wichtigsten Methoden

3.5.5 Die Klasse `GenericOperationDriver`

Die im ersten Semester verwendete Ansteuerung der Automaton-Klassen funktionierte über das Implementieren einer eigenen Klasse pro Aktion und eine umständliche Filterung bezüglich der Anwendbarkeit einer Operation auf einer im Workspace gegebenen Menge

von ausgewählten Automaten. Dies wurde als unzureichend empfunden, da umständlich neue Automaten-Operationen jeweils als neue Aktions-Klasse im Workspace programmiert werden mussten, umfangreicher Code mehrfach im Projekt vorkam und jede Aktion einzeln auf ihre Anwendbarkeit auf die aktuelle Automatenauswahl geprüft wurde.

Daher wurde im zweiten Semester eine generische Lösung entwickelt, welche die zuvor genannten Probleme löst. Die Lösung bestand in der Anwendung der Java Reflection-Technologie und den mit Java 5 erstmals eingeführten Annotations. Damit ist es möglich, zur Laufzeit auf Methoden zuzugreifen, deren Existenz oder genaue Ausprägung zur Zeit der Programmerstellung nicht bekannt war. Über Annotations sollten fortan alle auf Automaten anwendbare Methoden in einer Automaten-Klasse markiert und mit Meta-Informationen versehen werden. Der Driver soll dann anhand dieser Markierungen die jeweils anwendbaren Methoden heraussuchen und bei Anforderung die gewünschte Operation durchführen. Die Klasse `GenericOperationDriver` leistet das genannte und wird im Folgenden erklärt.

3.5.5.1 Die verwendeten Annotations

Annotations wurden in Java 5 erstmals eingeführt. Es handelt sich dabei um Markierungen im Quellcode, mit deren Hilfe Meta-Informationen eingefügt und zur Programm-Laufzeit ausgelesen werden können. Für den Einsatz des generischen Operations-Treibers haben wir eine Annotation *`@Operation`* entwickelt und in der Klasse `GenericOperationDriver` untergebracht. Die Annotation selbst wirkt bereits durch ihre alleinige Anwesenheit als Markierung einer Methode in einer Automaton-Implementation. Per Parameter können über die Annotation aber noch weitere Informationen zur Verfügung gestellt werden:

type Gibt eine Typisierung der annotierten Methode an, wobei im Augenblick zwischen „rendering“ und „generating“-Operationen unterscheiden wird. Eine „rendering“-Operation modifiziert das übergebene Automaten-Objekt, während eine „generating“-Operation ein neues Automaten-Objekt erstellt.

chainable Über diesen Booleschen Parameter kann angegeben werden, ob die mit einem Parameter aufgerufene Methode durch mehrfache Hintereinanderausführung auch für die Verknüpfung einer ganzen Liste von Parametern dienen kann. Beispielsweise kann eine Konkatenations-Methode, welche auf einem Automaten mit einem anderen Automaten als Parameter aufgerufen wird, auch mehr als zwei Automaten konkatenieren, indem das Ergebnis einer ersten Konkatenation mit dem zweiten Parameter-Automaten konkateniert wird und so weiter, bis alle Automaten verarbeitet sind.

identity Mit diesem Booleschen Parameter wird eine identitätserhaltende Methode, beispielsweise eine Minimierungs-Operation, gekennzeichnet. Der Workspace kann anhand dieses Parameters entscheiden, ob z. B. ein aus einer Formel generierter Automat nach Anwendung der Operation immer noch die Formel widerspiegelt (was zum Beispiel nach einer Minimierung ja der Fall ist) oder ob Automat und Formel nicht mehr übereinstimmen (weil z. B. das Komplement gebildet wurde).

3.5.5.2 Verwendung

Nachdem in einer Automaten-Klasse alle dazu geeigneten Methoden mit den oben genannten Annotations markiert wurden, kann ein Driver-Objekt erzeugt werden, indem der Konstruktor `public GenericOperationDriver(Class)` mit der Automaten-Klasse aufgerufen wird. Es stehen dann Methoden zur Verfügung, von denen die meisten eine Liste von Objekten als Parameter übernehmen. Diese Liste enthält alle Automaten-Objekte, auf denen eine Operation ausgeführt werden soll, bzw. für die im Vorfeld eine Liste von verfügbaren Operationen zurückgeliefert werden soll. Die beiden wichtigsten Methoden der Driver-Klasse werden nun kurz erläutert:

3.5.5.2.1 `public List<Method> availableMethods(List<? extends Object>)`
Diese Methode liefert eine Liste von `Method`-Objekten zurück, welche mit der übergebenen Liste von Parametern über diesen Driver aufgerufen werden können. Die Entscheidung darüber, ob eine Methode auf die übergebenen Parameter aufgerufen werden kann, wird anhand von Typ und Anzahl der Parameter-Objekte und anhand der Meta-Informationen der entsprechenden Operation-Annotation der Methode getroffen. Die Implementation dieser Entscheidungsfindung ist in der privaten Methode `private boolean canOperateOn(Method, List<? extends Object>)` (siehe auch 3.5.5.3) gegeben.

Die zurückgelieferte Liste enthält Objekte der Standard-API-Klasse `java.lang.reflect.Method` aus dem Java Reflection Package. Aus diesen Objekten können dann für den Workspace auf einfache Weise die jeweiligen Methodennamen und gegebenenfalls auch weitere Informationen extrahiert werden.

3.5.5.2.2 `public OperationResult operate(Method, List<? extends Object>)`
Diese Methode führt eine als Parameter übergebene Methode auf die ebenfalls übergebene Liste von Parametern aus und liefert das Ergebnis des Methoden-Aufrufes zurück. Hier sollten nur Methoden und Parameterlisten übergeben werden, die von der zuvor erwähnten Methode `availableMethods` als kompatibel identifiziert wurden,

da andernfalls eine `IllegalArgumentException` geworfen wird. Passen Parameter und Methoden zusammen, so wird ein Ergebnis, gekapselt in einem `OperationResult`-Objekt, zurückgeliefert. Dieser Ergebnis-Wrapper wird im folgenden Abschnitt kurz erläutert:

3.5.5.2.3 `class OperationResult` Die Klasse kapselt das Ergebnis eines Methodenaufrufes über die `operate`-Methode. Per einfachen getter-Methoden können dabei folgende Ergebnisse ausgelesen werden:

`boolean hasFailed()` Liefert über einen Booleschen Wert zurück, ob ein Operations-Aufruf einen Fehler (durch Werfen einer Exception) verursachte, oder erfolgreich durchlief

`Throwable getThrowable()` Liefert für den Fall, dass ein Operations-Aufruf einen Fehler verursachte (`boolean isFailed()` liefert also `true` zurück), das entsprechende `Throwable`-Objekt, also die Exception, welche von der Automaten-Klasse geworfen wurde.

`Method getMethod()` Liefert die aufgerufende Methode der Automaten-Klasse zurück.

`Object getResult` Liefert im Erfolgsfall, also wenn `boolean isFailed()` den Wert `false` hat, das Ergebnis des Methoden-Aufrufes zurück.

3.5.5.3 Details

Die private Methode `boolean canOperateOn(Method, List<? extends Object>)` ist an mehreren Stellen für die Kompatibilitäts-Überprüfung der übergebenen Methode (also einer Operation) und einer Liste von Objekten (als Parametern dieser Operation) verantwortlich. Die Methode funktioniert folgendermaßen:

1. Ist die übergebene Liste leer? Wenn ja, dann Rückgabe `false`.
2. Ist das erste Objekt in der Liste ein Exemplar der bei der Konstruktion des `GenericOperationDriver` übergebenen Klasse? Wenn nein, dann Rückgabe `false`.
3. Die Methode ist nicht als `Chainable` annotiert.
   (a) Unterscheidet sich die Anzahl von Parametern von der Anzahl der noch in der übergebenen Liste zur Verfügung stehenden Objekten (also alle bis auf das erste)? Wenn ja, dann Rückgabe `false`.
   (b) Haben alle noch verfügbaren Objekte den gleichen Typ wie der jeweils entsprechende Methoden-Parameter? Wenn nein, dann Rückgabe `false`.

4. Die Methode ist als `Chainable` annotiert.

   (a) Hat die Methode mehr als einen Parameter? Wenn ja, dann Rückgabe `false`.

   (b) Stehen noch mindestens so viele Objekte in der übergebenen Liste zur Verfügung, wie die Methode Parameter hat (also mindestens eins)? Wenn nein, dann Rückgabe `false`.

   (c) Haben alle noch zur Verfügung stehenden Objekte (also alle außer dem ersten) in der übergebenen Liste einen mit dem Methoden-Parameter kompatiblen Typ? Wenn nein, dann Rückgabe `false`.

5. Rückgabe `true`

Der Punkt 2. verdeutlicht die grundsätzliche Arbeitsweise: Das erste Objekt einer Parameterliste ist dasjenige, auf dem die entsprechende Methode aufgerufen wird. Die Punkte unter 4 erklären sich aus der Behandlung von `Chainable`-Methoden: Auf das erste Objekt der Parameterliste wird die übergebene Methode mit dem zweiten Objekt der Liste aufgerufen. Auf das Ergebnis-Objekt wird die Methode dann mit dem dritten Objekt der Liste aufgerufen usw.

3.5.5.4 Die abstrakte Klasse `Plugin`

Die Klasse `Plugin` ersetzt und übernimmt die Funktionen der Klasse `AddonWorkspacePreprocess`, da diese Klasse im zweiten Semester umbenannt und mit weiteren Funktionen versehen worden ist. Hauptaufgabe der Klasse `AddonWorkspacePreprocess` war es eine einheitliche Struktur aller Addons zu erreichen, um diese dynamisch zu laden. Die neue abstrakte Klasse `Plugin` erfüllt weiterhin diese Aufgabe und folgende neue Aufgaben kamen im zweiten Semester dazu:

- Über die Methode `addAutomatonToFrame(InnerFrame innerFrame,Automaton automaton, boolean editable)` kann jedes Plugin flexibel bestimmen, wie ein Automat in seinem zugehörigen InnerFrame dargestellt wird. Außerdem kann bestimmt werden, ob dieser Automat im Editor bearbeitet oder nicht bearbeitet werden darf. Eine konsistente Darstellung der Eingabe und der dargestellten Automaten wird so erreicht. Siehe `GenericOperationDriver` Abschnitt **??** identity.

- Jedes Plugin wird über Auswahlveränderungen von Automaten im Workspace informiert. Dies geschieht über die Methoden `framesSelected(List<InnerFrame> selectedFrames)` und `framesDeselected(List<InnerFrame> selectedFrames)`.

So kann jedes Plugin gezielt überprüfen, ob es für einen markierten oder demarkierten Automaten zuständig ist und dieses händeln, wie z. B. Hinzufügen oder Entfernen von Menüeinträgen.

- Um Automaten optimal Speichern und Laden zu können, gibt es die Methoden `savePlugin(InnerFrame frame, Document document)` und `loadPlugin(Document document,Document automatonDocument)`. Auch hier ist der Grundgedanke, dass ein Plugin am besten selber weiß, wie alle benötigten Informationen optimal gespeichert und wieder geladen werden.

3.5.5.5 Die Klasse `PluginManager`

Die abstrakte Klasse Addonmanager wird im zweiten Semester nicht mehr gebraucht, da wir nur noch Plugins und keine Driverplugins mehr verwenden. So brauchen wir auch keinen Drivermanager mehr. Der Pluginmanager hat seine alte Funktionalität beibehalten, nur erbt dieser nicht mehr von der Klasse Addonmanager. Der Pluginmanager durchsucht weiterhin alle Unterverzeichnisse nach installierten Plugins und lädt diese dynamisch ein. Neu ist, dass er eine Liste aller geladenen Plugins verwaltet.

Er übernimmt außerdem Aufgaben beim Speichern und Laden von Automaten. Diese werden vom Pluginmanager mit Hilfe des Java Document Object Model im XML-Format gespeichert. Das Document Object Model (DOM) ist eine vom World Wide Web Consortium definierte Programmierschnittstelle für XML-Dokumente. Der Pluginmanager erzeugt zuerst ein `org.w3c.dom.Document`. Dazu wird ein `DocumentBuilder` unter Zuhilfenahme der `DocumentBuilderFactory` instanziiert. Der `DocumentBuilder` erzeugt über die Methode `newDocument()` ein neues `org.w3c.dom.Document`. Das Document Object Model ist ein Objektmodell, das eine Baumstruktur des Dokuments abbildet. Als Wurzel wird ein Element *project* eingefügt. Alle weiteren Automaten werden unter dieser Wurzel in das `org.w3c.dom.Document` eingehängt. Im Anschluss geht er die Liste aller markierten InnerFrames durch, ermittelt ihr zugehöriges Plugin und fordert diese auf, den InnerFrame ins XML Format zu transformieren. Das Ergebnis wird dann vom Plugin dem übergebenen `org.w3c.dom.Document` angehängt.

Beim Laden einer XML-Datei überprüft der Pluginmanager zuerst die Gültigkeit der XML-Datei. Handelt es sich um eine gültige XML-Datei, so zerlegt der Pluginmanager diese, indem er sich alle Kinddokumente unter der Wurzel anschaut. Dort findet er die Informationen, mit welchem Plugin dieser Teilbaum gespeichert worden ist. Mit Hilfe dieser Information und des Teilbaums schaut der Pluginmanager nach, ob er dieses Plugin installiert hat. Ist dieses Plugin nicht installiert, so erhält der Anwender eine Fehlermeldung und der Pluginmanager fährt mit dem nächsten Element fort. Ist das benötigte Plugin

vorhanden, so übergibt er diesen Teilbaum direkt an das zuständige Plugin. Dieses Plugin lädt dann den Teilbaum in den Workspace ein.

3.5.5.6 Implementierung von Plugins am Beispiel endlicher Automaten

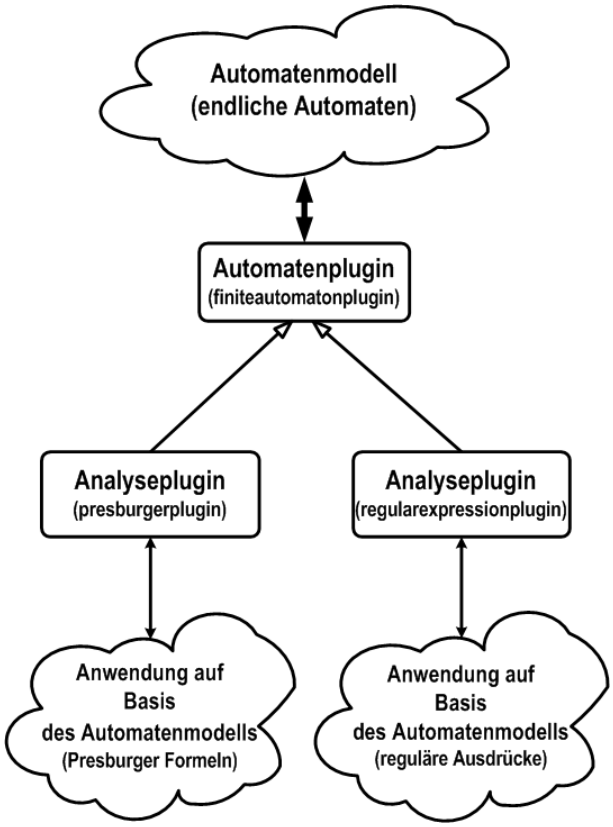


Abbildung 3.33: Zusammenhang von Automaten- und Analyseplugins

Im Minimalfall besteht ein konkretes Plugin aus den im ersten Semester beschrieben Komponenten. Es muss zum einen die Datei `addon.xml` geben und zum anderen eine von `Plugin` erbende Klasse.

Desweiteren müssen vom konkreten Plugin die neuen Methoden implementiert werden, die im vorherigen Abschnitt angesprochen worden sind.

Da es nun keine Driverplugins mehr gibt, muss sich das Plugin darum kümmern, die automatenspezifischen Operationen mit Hilfe des `GenericOperationDriver` aus der Automatenbibliothek auszulesen und diese in den Menüs anzuzeigen. Das Plugin hat dafür beste Voraussetzungen, da es über die Methoden `framesSelected(List<InnerFrame> selectedFrames)` und `framesDeselected(List<InnerFrame> selectedFrames)` immer genau informiert wird, welche und wieviele Automaten markiert sind. So kann das Plugin gezielt auf die Situation eingehen und die Menüs füllen.

Wo liegt nun der Vorteil eines Pluginkonzeptes, in dem sich jedes Plugin selber um die Au-

tomatenoperation kümmern muss, was im ersten Semester noch die Aufgabe von speziellen Driver-Plugins war? Dieser Vorteil wird erst klar, wenn man die Plugins feiner strukturiert. Sowohl das `PresburgerPlugin` als auch das `RegularExpressionPlugin` nutzen als Basis endliche Automaten. Die Implementation der Automatenoperationen unterscheiden sich in beiden nicht, und anstatt diese doppelt zu implementieren, schaffen wir eine Oberklasse `FiniteAutomatonPlugin`, von denen die anderen Plugins für endliche Automaten erben.

Welche Vorteile hat das Ganze:

- Durch das `FiniteAutomatonPlugin` bekommt man ein Plugin, mit dem man einen Automaten frei zeichnen kann, ohne eine Transformation zu verwenden. Hier stehen alle Operationen des Automatentyps zur Verfügung.
- Wenn man Automaten als Ergebnis einer Automatentransformation erhalten hat, so besitzen diese immer eine Transformationseingabe. Ruft man auf diesen Automaten z. B. `renderComplementary()` auf, so wird der aktuelle Automat verändert und passt nicht mehr zur ursprünglichen Eingabe. Aus diesem Grund ermöglicht dieses Konzept das Degradieren von Plugins. So wird aus dem Analyseplugin ein einfaches Automatenplugin, mit dem problemlos weitergearbeitet werden kann.
- Durch eine Vererbungsstruktur der Automaten spart man sich eine Menge Arbeit, da man das Einbinden der Operationen, Automatenanalysen oder die Basisfunktionen des Laden und Speicherns nur einmal implementieren muss. Dabei können diese jederzeit erweitert werden, was das gesamte System flexibler als das alte Driverpluginkonzept macht.

Wir sehen nun am Beispiel von endlichen Automaten, wie ein komplexeres Plugin implementiert werden kann.

3.5.5.7 Package `aaa.workspace.plugin.finiteautomatonplugin`

Die Klasse `FiniteAutomatonPlugin`: Dies ist die Hauptklasse des Plugins für endliche Automaten und implementiert alle im Plugin kennengelernten Methoden. Wir geben hier einen kurzen Überblick der Funktionsweise:

Menüs füllen: Bei Auswahländerungen wird das Plugin darüber informiert und ruft die Methode `updateMenus()` auf. Diese prüft über `checkSelectedFrames()`, ob das Plugin für einen markierten InnerFrame verantwortlich ist. Wenn ja, dann wird dem `GenericOperationDriver` eine Liste aller ausgewählten Automaten übergeben. Dieser liefert eine Liste aller

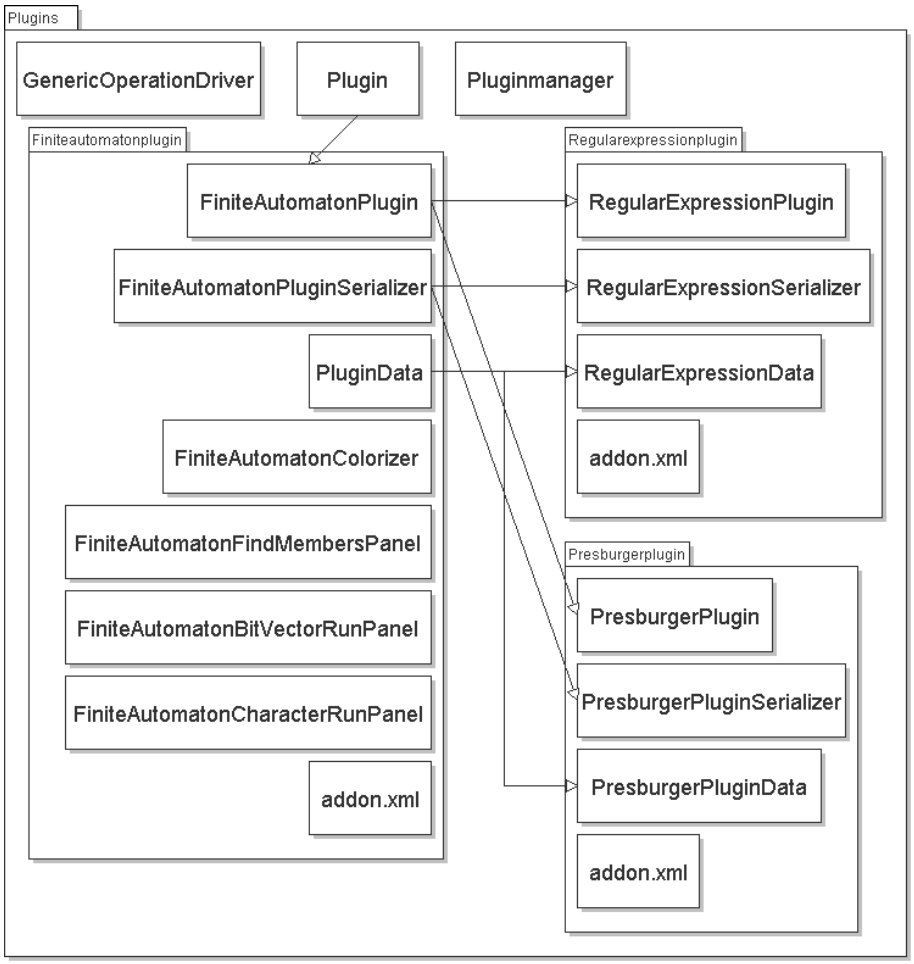


Abbildung 3.34: Klassenstruktur aller Plugins für endliche Automaten

gültigen Automatenoperationen zurück, welche über `AutomatonAction` in den Workspace eingetragen werden. Eine Besonderheit sind die zwei Analyseverfahren FindMember und FiniteRun. Diese werden in einem solchen Fall in das Analysemenü eingetragen.

Serialisieren: siehe `FiniteAutomatonPluginSerializer`

Die Klasse `FiniteAutomatonPluginSerializer`: Um die Klasse `FiniteAutomatonPlugin` etwas übersichtlicher zu gestalten, wurden alle Operationen zum Speichern und Laden in diese Klasse ausgelagert. Wird ein Plugin zum Speichern oder Laden aufgefordert, so leitet es diese Aufforderung an diese Klasse weiter.

Speichert man einen einfachen endlichen Automaten, so erhält man die folgende Struktur:

```
<?xml version="1.0" encoding="windows-1252" ?>
  */XML-Header /*

<project>
  */Wurzel des XML Dokuments*/

  <plugin class="aaa.workspace.plugin
     .finiteautomatonplugin.FiniteAutomatonPlugin">
    /*Plugin des ersten gespeicherten Automaten*/

      <title>Finite Automaton</title>
        /*Titel des Automaten*/

      <automaton editable="true">
        /*Informationen des Backendautomaten.
        Das Attribut editable gibt an ob der
        Automat bearbeitet werden darf*/
       ...
      </automaton>

      <statelocations>
        /*Posititon aller Frontendknoten*/
       ...
      </statelocations>
  </plugin>
    /*weitere Automaten*/
```

```
    ...
</project>
```

Diese Struktur entsteht folgendermaßen: der Pluginmanager erzeugt ein `org.w3c.dom.Document` mit Wurzelknoten *project*. Danach ermittelt der Pluginmanager zu jedem Innerframe bzw. Automatenfenster das gültige Plugin und fordert dieses zum Speichern des Innerframes auf. Dies geschieht, indem das Plugin die Methode `public void savePlugin(InnerFrame innerFrame, Document document)` im `FiniteAutomatonPluginSerializer` aufruft.

Der `FiniteAutomatonPluginSerializer` fügt zuerst einen neuen Knoten *plugin* hinzu. Der Knoten erhält ein Attribut *class*, dass mit dem Klassennamen des Plugins belegt wird. So weiß der Pluginmanager beim Laden, welches Plugin den Teilbaum fehlerfrei laden kann. Unter diesen Knoten werden nun alle Informationen des Innerframes eingetragen. Der Knoten *title* wird mit dem Titel des Innerframes gefüllt. Die Daten des Automaton werden über die Methode `AutomatonSerializer.createDocumentFromAutomaton(automaton)` und die aktuellen Koordinaten der Knoten des PaintableAutomaton werden über die Methode `StateLocationSerializer.serialize(pAutomaton, document, element)` ermittelt und dem Dokument angehängt.

Beim Laden zerteilt die Klasse `FiniteAutomatonPluginSerializer` das Dokument in die oben beschriebenen Teile und lädt diese über geeignete Methoden ein.

Die Klasse `PluginData`: Diese Klasse sammelt alle wichtigen Informationen eines Plugins. So werden hier der `FiniteAutomaton` und der `PaintableAutomaton` gespeichert. Die Klasse ermöglicht es, diese Daten an einer Stelle zentral zu verwalten.

Die Klasse `FiniteAutomatonColorizer` Diese Klasse ist eine Hilfsklasse für die Analyseverfahren, wie das Testen einer Eingabe auf endlichen Automaten oder das Auffinden eines akzeptierten Wortes. Die Klasse besitzt Methoden, um einen Frontendautomaten einzufärben und somit bestimmte Knoten oder Kanten farbig hervorzuheben.

Die Klasse `FiniteAutomatonFindMemberPanel`: Diese Klasse implementiert ein Find-Member Panel, das für jeden akzeptierenden Zustand ein akzeptiertes Wort anzeigt. Dazu wird im Analysebereich ein neues Tab erzeugt und das Ergbnis in einer `JTable` angezeigt.

Dazu wird zuerst ermittelt, welches Basisalphabet der gegebene Automat besitzt. Je nach Alphabet wird zur Darstellung der Tabelle ein unterschiedliches `TableModel` verwendet. Im Augenblick sind `TableModels` für `CharacterAlphabet` und das

`AssignmentAlphabet` implementiert. Danach wird für jeden akzeptierenden Zustand ein akzeptiertes Wort über die Automatenmethode `findMembers()` ermittelt, die eine `List<List<Symbol>>` als Ergebnis liefert. Dieses Ergebnis wird dem `TableModel` übergeben und in der Tabelle angezeigt. Als kleines optisches Feature kann ein Anwender die Einträge in der Tabelle auswählen. Diese akzeptierenden Pfade werden dann im `PaintableAutomaton` mittels des`FiniteAutomatonColorizer` farbig hervorgehoben.

Die Klassen `FiniteAutomatonBitVectorRun` & `FiniteAutomatonCharacterRun`: Diese beiden Klassen ermöglichen es, Eingaben im Automaten testen zu lassen und diese farblich im `PaintableAutomaton` hervorzuheben. Wegen Unterschieden in den Alphabeten gibt es hier eine Klasse für das `CharacterAlphabet` und eine für das `AssignmentAlphabet`.

Beide Klassen ermöglichen eine komfortable Eingabemöglichkeit. Beim `CharacterAlphabet` reicht ein einfaches `JTextField`, bei `AssignmentAlphabet` steht eine Tabelle mit den einzelnen Variableneinträgen zur Verfügung. Der Run wird von der Automatenklasse `FiniteRun` unterstützt, so kann man einen schrittweisen Lauf durch die Eingabe durchführen. Das Einfärben des Automaten wird auch hier über Schnittstellen der Klasse `FiniteAutomatonColorizer` durchgeführt.

3.5.5.7.1 Package `aaa.workspace.plugin.presburgerplugin` und Package `aaa.workspace.plugin.regularexpressionplugin`: Durch die Vererbungsstruktur unterscheiden sich nur noch die Transformation und die dabei entstehenden Daten vom `FiniteAutomatonPlugin`. Unterschiede treten an drei Stellen auf.

- Selbstverständlich unterscheiden sich die Aktionen `CreateAutomatonAction` und `NewAutomatonAction`, weil hier jeweils ein Eingabetab benötigt wird und beim Erzeugen eines neuen Automaten unterschiedliche Transformationen angestoßen werden müssen.
- Durch das neue Sourcetab muss der Pluginserializer angepasst werden, so dass dieser auch das Sourcetab mitspeichert
- Die `Plugindata` wird um einen Source-Eintrag erweitert, weil wir dort zentral alle Daten sammeln.

3.5.5.8 Pluginübersicht

Der Workspace wurde im zweiten Semester mit zwei Automaten und sechs Analyseplugins ausgestattet.

Automantenplugins:

- Plugin für endliche Automaten
- Plugin für Büchiautomaten

Analysen mit endlichen Automaten:

- Plugin zur Erzeugung eines endlichen Automaten aus regulären Ausdrücken
- Plugin zur Lösung von Formeln der Presburger Arithmetik

- Model Checking auf Basis eines Kripkemodells
- Model Checking über eine LTL-Formel
- Erzeugung von Büchiautomaten auf Basis eines ω-regulären Ausdrucks
- Erzeugung von Kripkemodellen aus While-Programmen

3.6 Editor

Zu Beginn der PG wurden einige Visualisierungsmöglichkeiten für Automaten durch Fremdbibliotheken evaluiert. Zwar wurde mit *JUNG - Java Universal Network/Graph Framework* [**?**] eine anfangs brauchbare Lösung gefunden, doch die Notwendigkeit, in diesem Paket zahlreiche Änderungen vornehmen zu müssen, sowie die Motivation, auf Fremdbibliotheken weitgehend zu verzichten, haben zu dem Entschluss geführt, eine eigene Implementation eines Visualisierers und Editors für Automaten, ergo Graphen, zu erstellen.
Im Folgenden werden Ideen, Funktionen und Implementationsdetails für den entwickelten Editor aufgeführt.

3.6.1 Ziele

Der Editor hat natürlich als primäres Ziel die *Visualisierung von Automaten*. Neben der Erstellung eines dafür geeigneten Frameworks zur graphischen Anzeige gab es aber als grundlegende Problemstellung noch das *automatisierte Layouten* der Automaten-Graphen.
Zusätzlich zu diesen nur die Anzeige betreffenden Zielen sollte es auch noch ermöglicht werden, *Automaten visuell zu editieren*, im Übrigen eine Anforderung, die mitverantwortlich für die Abkehr vom JUNG-Framework war.

Abgesehen von den gerade genannten Primärzielen war es auch ein Anliegen, den zu erstellenden Editor flexibel bezüglich der Erweiterung um neue Automaten-Typen und Layout-Algorithmen zu gestalten, aber auch das Design möglichst eigenständig zu halten: der Editor, zumindest die reine Visualisierungskomponente, sollte auch eigenständig ohne Workspace lauffähig sein. Trotz dieser Eigenständigkeit wurde aber natürlich Wert auf eine gute Integration des Editors in den parallel zu erstellenden Workspace gelegt.
Während im ersten Semester vor allem die Entwicklung des allgemeinen Frameworks zur Anzeige von Automaten im Vordergrund stand, wurde das zweite Semester dem Ausbau der Editiermöglichkeiten, neuen Automaten-Typen und der noch tiefer gehenden Integration in den Workspace gewidmet.

Zusammengefasst stellten sich also folgende Anforderungen:

- Visualisierung von verschiedenen Automaten-Modellen
- automatisiertes Layout von Automaten-Graphen
- Bearbeitung / Erstellung von Automaten

- leichte Erweiterbarkeit auf neue Automaten- bzw. Graphen-Typen
- Implementation als eigenständiges Framework innerhalb des Gesamt-Projekts
- Integration in den Triple-A-Workspace

3.6.2 Das Paintable-Framework

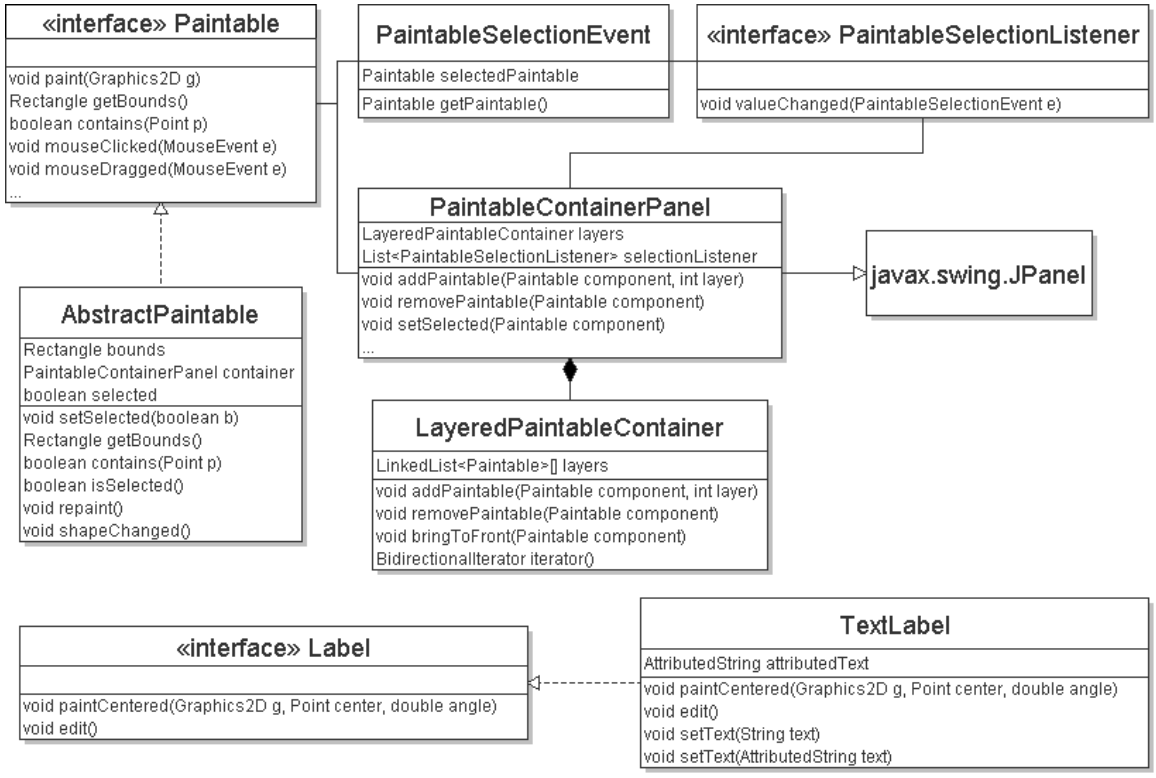


Abbildung 3.35: Die wichtigsten Klassen und Methoden des Paintable-Frameworks

Zur Visualisierung von Komponenten im Editor wurde auf keine bestehende Implementation eines graphischen Systems für Java zurückgegriffen. Stattdessen wurde ein eigenes System, das *Paintable-Framework* geschaffen. Dieses nutzt das Standard-Java-Package `java.awt` und verwendet die ebenfalls bekannten Klassen des `javax.swing`-Packages nur als Schnittstelle in Form von Panels, welche in eine Zielanwendung eingebaut werden können. Die wichtigsten Klassen und Methoden sieht man in Abbildung 3.35. Im Sinne der Lesbarkeit werden nicht alle Methoden und Attribute dargestellt. Die API-Dokumentation (die nicht Teil dieses Dokumentes ist) ist jedoch vollständiger.

3.6.2.1 Interface `Paintable`

Ein Objekt, dass im Paintable-Framework gezeichnet werden soll, muss von einer Klasse erzeugt worden sein, die das Interface `Paintable` implementiert. Dieses Interface verlangt

vom Objekt, sich selbst über die Methode `paint` zeichnen zu können. Darüber hinaus bietet es Schnittstellen zur Mausansteuerung. Im folgenden werden die wichtigsten Methoden kurz erklärt. Für eine umfassende Erläuterung sei auf die API-Dokumentation verwiesen.

3.6.2.1.1 `public void paint(Graphics2D g)` Diese Methode wird vom Editor aufgerufen, wenn sich das Objekt selbst zeichnen soll. Das übergebene Objekt `g` des Typs `Graphics2D` stammt aus dem *Java-AWT-Package* und ist somit durch jeden erfahrenen Java-Entwickler leicht nutzbar. Über das Graphics-Framework von `java.awt` lassen sich alle erdenklichen Formen leicht zeichnen und zahlreiche weitere graphische Operationen, z. B. das Einbinden von Grafik-Dateien, durchführen.

3.6.2.1.2 `public Rectangle getBounds()` Diese Methode liefert ein Rechteck-Objekt zurück, welches das zu zeichnende Objekt vollständig umschließt.
Die Methode wird vom Paintable-Framework zur Performance-Optimierung genutzt. Einerseits kann der neu zu zeichnende Bereich bei Veränderungen eingegrenzt werden (und damit auch, welche weiteren Objekte neu gezeichnet werden müssen, weil sie jetzt zum Beispiel nicht mehr durch ein weiteres verdeckt werden), andererseits dient das zurückgelieferte Rechteck auch dem schnelleren Auffinden von mit der Maus zu manipulierenden Objekten.
Da die Methode der Performance-Steigerung dient, braucht sie kein exakt passendes Rechteck zurückzuliefern, sollte dessen Berechnung komplexer sein. Das Rechteck kann also durchaus größer sein als die Komponente, darf aber niemals kleiner sein als diese.

3.6.2.1.3 `public boolean contains(Point point)` Die Methode `contains` liefert per Wahrheitswert die Information, ob ein übergebener Punkt innerhalb des gezeichneten Objektes liegt oder nicht. Diese Information dient dem Editor zur Auffindung desjenigen Objektes, welches gerade mit der Maus bearbeitet wird. Eine solche Maus-Interaktion wird dann mit einer der Methoden `public void mouse...(MouseEvent e)` an dasjenige Objekt, welches über `contains` den Wert `true` liefert, weitergereicht.
Zur Optimierung wird die `contains`-Methode nur bei denjenigen Objekten aufgerufen, bei denen der Ort der Maus-Interaktion in dem per `getBounds` zurückgelieferten Rechteck liegt. Die Berechnung dieses Rechtecks ist häufig schneller möglich, als die Positionsbestimmung eines Punktes innerhalb einer komplexeren Form, so dass diese Maßnahme der Geschwindigkeitsoptimierung dient, die umso sinnvoller ist, je mehr Objekte innerhalb des Editors dargestellt werden.

3.6.2.1.4 `public void mouse...(MouseEvent e)` Die Punkte im Methodennamen stehen hier stellvertretend für eine Menge von Methoden, welche bei Maus-Interaktionen automatisch vom Editor aus aufgerufen werden. Es gibt Methoden für das Anklicken, Betreten, Verlassen, Ziehen usw. Das dabei übergebene `MouseEvent`-Objekt stammt aus dem Package `java.awt.event` und ist somit für Java-Programmierer kein Unbekanntes.

3.6.2.2 Klasse `AbstractPaintable`

Diese Klasse enthält einige der vom `Paintable`-Interface verlangten Methoden mit einer Standard-Implementation und dient nur dem Komfort des Programmierers, der für seine eigenen Komponenten dann nur noch von dieser Klasse ableiten muss.

3.6.2.3 Klasse `PaintableContainerPanel`

Die Klasse `PaintableContainerPanel` stellt die Schnittstelle zu Suns `javax.swing`-Package dar. Es ist ein `JPanel`, das die Zeichenfläche des Editors enthält und in jede Java-Swing-Applikation eingebaut werden kann. Auch hier folgt ein Überblick über die wichtigsten Methoden:

3.6.2.3.1 `public void addPaintable(Paintable component, int layer)` Fügt der Zeichenfläche ein `Paintable`-Objekt hinzu. Der Parameter `layer` bestimmt dabei, auf welcher Ebene das Objekt einzufügen ist. Ebenen mit kleineren Integer-Werten liegen dabei unter solchen mit höheren Werten.

3.6.2.3.2 `public void removePaintable(Paintable component)` Diese Methode stellt den Gegenspieler zu `addPaintable` dar und entfernt somit ein `Paintable`-Objekt wieder von der Zeichenfläche.

3.6.2.3.3 `public void justify(int padding)` Bei Aufruf dieser Methode werden alle Komponenten der Zeichenfläche so verschoben, dass am linken und oberen Rand der Abstand zur nahegelegensten Komponente dem mit `padding` übergebenen Wert (in Pixeln) entspricht. Die Methode nutzt zur Bestimmung der Verschiebung die`getBounds`-Methode der Komponenten, so dass der per `padding` übergebene Wert nur so genau eingehalten werden kann, wie es der Genauigkeit der `getBounds`-Methode in den verschiedenen `Paintable`-Komponenten entspricht.

3.6.2.3.4 `public void addSelectionListener(PaintableSelectionListener listener)` Über diese Methode können sich interessierte Klassen, welche das `PaintableSelectionListener`-Interface implementieren, über das Selektieren von Komponenten im Editor informieren. Bei einer Selektion wird jedem Listener ein Objekt vom Typ `PaintableSelectionEvent` übermittelt, das die Auswahlinformation enthält. Als Anwendungszweck wäre denkbar, dass die den Editor einbindende Applikation zu einem ausgewählten Objekt zusätzliche Informationen anzeigt. Genutzt wird diese Funktion beispielsweise bei der Anzeige im Property-Panel des Workspace (siehe 3.5.3.4).

3.6.2.4 Klasse `LayeredPaintableContainer`

Die Klasse `LayeredPaintableContainer` stellt einen Container mit mehreren Ebenen für Paintable-Objekte zur Verfügung. Die Klasse wird intern vom Editor benutzt und stellt Methoden zum Hinzufügen, Entfernen und Positionieren von Objekten innerhalb ihrer Ebene zur Verfügung. Zusätzlich ist ein Iterator verfügbar, der die im Container enthaltenen Objekte in beide Richtungen zurückliefern kann, was einerseits für die Reihenfolge beim Zeichnen (unterste zuerst zeichnen), als auch für das Feststellen von Maus-Zielen (oberste zuerst prüfen) von Bedeutung ist.

3.6.2.5 Interface `Label` und Klasse `TextLabel`

Das Interface `Label` und die implementierende Klasse `TextLabel` stellen einen innerhalb des `Paintable`-Frameworks abgesetzten Teil dar. Sie bieten einen Standard für die Implementation von Beschriftungen, die von innerhalb eines `Paintable`-Objekts genutzt werden kann.
Natürlich kann jede **Paintable**-Komponente ihre Beschriftung, sofern vorhanden, auf eigene Art und Weise vornehmen. Die Verwendung der Klasse **TextLabel** kann aber eine große Vereinfachung dieser Arbeit bedeuten.

3.6.2.5.1 `public void paintCentered(Graphics2D g, Point center, double angle)` Eine Implementation dieser Methode muss das Label-Objekt zentriert zum Punkt `center`, im angegebenen Winkel `angle` in das übergebene `Graphics2D`-Objekt `g` zeichnen.

Die Klasse `TextLabel` dient dem Darstellen von textuellen Labels. Diese können attributiert, also mit unterschiedlichen Schriftarten und -größen, sowie mit Effekten wie Unterstreichung, Kursivdarstellung, Fettdruck usw. versehen werden.

3.6.2.5.2 `public void setText(AttributedString text)` dient in der Klasse `TextLabel` dem Festlegen des darzustellenden Textes. Der Objekttyp `AttributedString` stammt aus dem Package `java.text` und ist somit ein Java-Standard-Element.
Sollte keine Attributierung gewünscht sein, kann der Text auch über die Methode `public void setText(String text)` als einfacher String gesetzt werden.

3.6.3 Das „PaintableAutomaton“-Addon

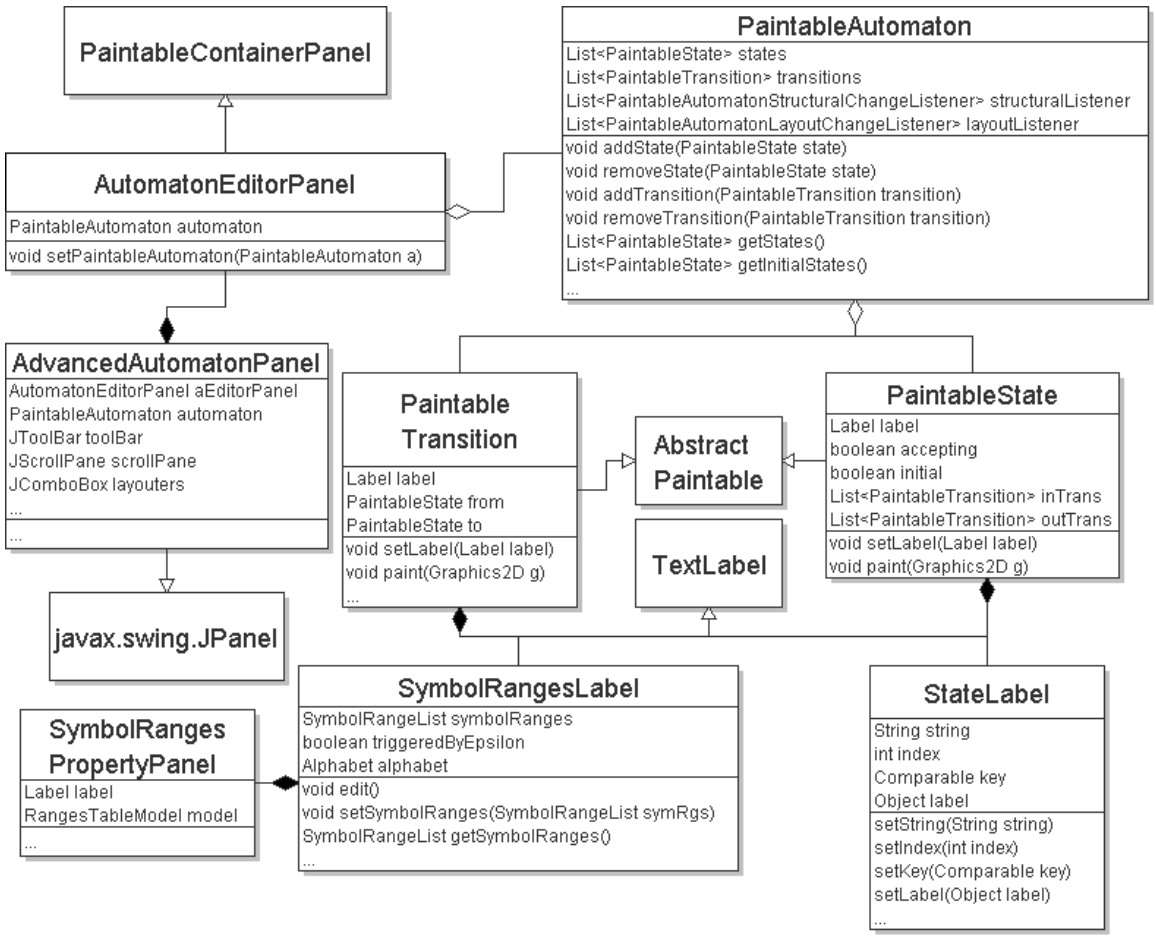


Abbildung 3.36: Die wichtigsten Klassen und Methoden des PaintableAutomaton-Addons, welches auf dem Paintable-Framework aufsetzt

Das PaintableAutomaton-Addon benutzt das oben beschriebende Paintable-Framework zur Darstellung von Automaten. Im ersten Semester diente es zur Visualisierung von Automaten des Typs `aaa.automaton.FiniteAutomaton`. Im zweiten Semester kamen weitere Modelle hinzu. Die Erweiterungen und Anpassungen auf diese Modelle werden in den Abschnitten 3.6.4 (Kripke-Modelle) und 3.6.5 (OBDDs) beschrieben.

Die Klassen `PaintableAutomaton`, `PaintableState` und `PaintableTransition` stellen eine eigene Datenstruktur für Automaten dar. Allerdings enthält sie keine automatenspe-

zifische Funktionalitäten, kann sich dafür aber über das Paintable-Framework darstellen und bearbeiten lassen. Zur Anzeige eines Automaten aus dem `aaa.automaton`-Package ist eine Übersetzung in einen `PaintableAutomaton`-Automaten erforderlich, welche aber im Prinzip nur eine 1:1-Abbildung darstellt und somit ohne größeren Aufwand möglich ist.

Abbildung 3.36 zeigt die wichtigsten zur Umsetzung implementierten Klassen. Die Paintable-Basis findet man in den Klassen `PaintableContainer`, `AbstractPaintable` und `TextLabel` wieder, die zur Verwendung für die Automatendarstellung passend erweitert werden.

3.6.3.1 Klasse `PaintableAutomaton`

Die Klasse `PaintableAutomaton` stellt das visualisierbare Gegenstück zur Klasse `aaa.automaton.Automaton` dar. Sie ist im Prinzip nur ein Container für `PaintableState`- und `PaintableTransition`-Objekte. `PaintableAutomaton` implementiert selbst nicht das `Paintable`-Interface, sondern ist im Prinzip nur eine Verwaltungsklasse.
Da die Methoden zur Verwaltung von Zuständen und Transitionen selbstbeschreibend sind, seien hier nur noch kurz die Listener erwähnt, die sich in `PaintableAutomaton` registrieren lassen.

3.6.3.1.1 `PaintableAutomatonLayoutChangeListener` Diese Listener können über Veränderungen im Layout eines PaintableAutomaton informiert werden. Das ist im Wesentlichen das Verschieben von Zuständen.
Ein möglicher Einsatzzweck dieses Listener-Typs ist die Realisierung einer Mini-Map, welche den (evtl. nur teilweise) dargestellten Automaten in einer kleinen Version (aber vollständig) im Workspace anzeigt und als Navigator dienen kann.

3.6.3.1.2 `PaintableAutomatonStructuralChangeListener` Listener dieses Typs erhalten Benachrichtigungen über strukturelle Änderungen am Automaten. Strukturelle Änderungen sind das Hinzufügen/Entfernen von Zuständen und Transitionen sowie Veränderungen an den Labels, also Zustandsbeschreibungen und Kantenübergangs-Symbole.
Dieser Listener kann ebenfalls für die oben erwähnte Mini-Map genutzt werden und ermöglicht außerdem das Aktualisieren des `aaa.automaton.Automaton`-Objektes, welches durch den dargestellten `PaintableAutomaton` repräsentiert wird. Letzterer Anwendungszweck wurde im zweiten Semester zur Bearbeitung und Erstellung von Automaten realisiert. Diesem ist ein eigener Abschnitt (3.6.3.6) gewidmet.

3.6.3.2 Klasse `PaintableState`

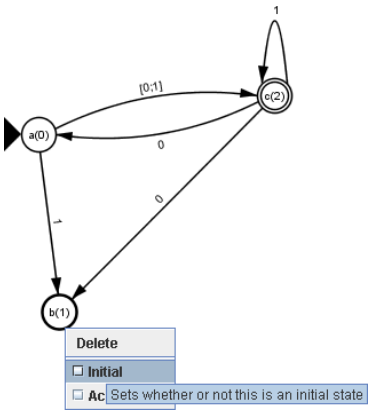


Abbildung 3.37: Beispielbild der drei möglichen „Typen“ von Zuständen und ein Popup-Menü zur Bearbeitung

Die in Abbildung 3.37 dargestellten Zustände sind Instanzen der Klasse `PaintableState`. Diese ist der zeichenbare Gegenspieler zur Klasse `aaa.automaton.State`. Die Zustände können im Editor frei verschoben werden, so dass das Layout des Automaten leicht nachbearbeitet werden kann.
Über die Methode `setColor(Color color)` kann die Farbe eines Zustandes gesetzt werden. Dies wird zum Beispiel im Workspace genutzt, um einen akzeptierenden Pfad innerhalb eines Automaten zu kennzeichnen.

Über ein Popup-Menü kann die Akzeptanz-Eigenschaft des Zustandes und der Initialknoten eines Automaten editiert werden.

3.6.3.2.1 Klasse `StateLabel` Dieses Label dient der Darstellung von Zustands-Beschriftungen, wie sie in `aaa.automaton.State` definiert sind.

3.6.3.3 Klasse `PaintableTransition`

Abbildung 3.38 zeigt die drei verschiedenen, durch die Klasse `PaintableTransition` gezeichneten Kantendarstellungen. Durch die Verwendung von Kurven (`java.awt.geom.QuadCurve2D`) konnte eine ansprechende Darstellung erreicht werden.
Wie bei den `PaintableStates`, kann auch bei den Kanten über die Methode `setColor(Color color)` die Farbe gesetzt werden. Die Methode `setLabel(Label label)` dient zum Setzen der Kantenbeschriftung, welche je nach Orientierungssinn der Kante ebenfalls mitgedreht wird. Dabei wurde darauf geachtet, dass die Beschriftung niemals auf dem Kopf steht.

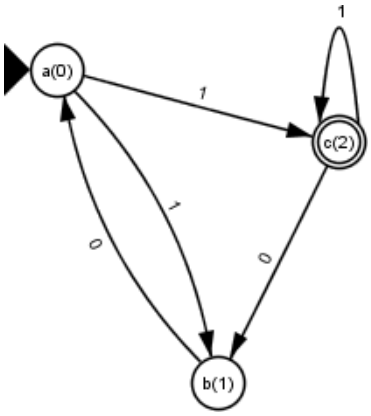


Abbildung 3.38: Die drei möglichen Darstellungsformen von Transitionen: einfache, symmetrische und selbstbezogene Kanten

3.6.3.3.1 Klasse `SymbolRangesLabel` Die Klasse `SymbolRangesLabel` leitet von `TextLabel` ab und ist eine generische Klasse zum Anzeigen von `SymbolRange`-Listen, welche im `aaa.automaton.FiniteAutomaton` für die Übergangsfunktion benutzt werden. Zur Anzeige wird die `toString()`-Methode der jeweils verwendeten Symbole benutzt. Es ist vorstellbar, dass für speziellere Darstellungswünsche an den Kanten neue Label erstellt werden. So könnten zum Beispiel Bitvektoren vertikal dargestellt werden.

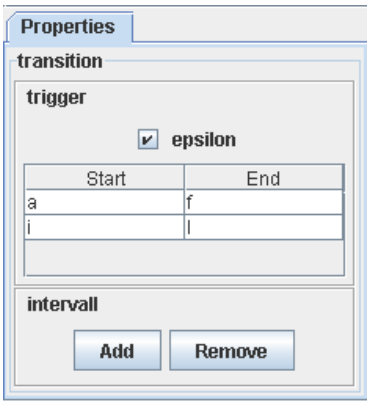


Abbildung 3.39: Ein im Property-Panel eingeblendeter Dialog zur Bearbeitung der `SymbolRange`s an Transitionen

3.6.3.3.2 Klasse `SymbolRangesPropertyPanel` Mit dem in Abbildung 3.39 dargestellten `SymbolRangesPropertyPanel` wird eine ebenfalls generische, vom `SymbolRanges-Label` aus benutzte Klasse angeboten. Sie stellt ein Editier-Panel dar, welches zum Bearbeiten der `SymbolRange`-Listen benutzt werden kann. Die Darstellung und auch die Eingabe der Symbole erfolgt per Strings. Für die Eingabe wird die `createSymbol(String sym)`-Methode des vom Automaten benutzten Alphabets benutzt. Auch hier ist es vorstellbar, die generische Programmierung dieses Panels für speziellere Symbol-Typen durch

eine komfortablere Version zu ersetzen. Bitvektoren könnten beispielsweise durch An- und Ausklicken von Bits editiert werden, und nicht – wie im generischen Dialog der Fall – durch Eingabe von 0-1-Strings.

3.6.3.4 Klasse `AutomatonEditorPanel`

Diese Klasse erbt von `PaintableContainerPanel` (siehe auch 3.35) und stellt somit die Hauptverbindung zum Paintable-Framework dar. Anstelle von Paintable-Objekten wird hier einfach ein `PaintableAutomaton`-Objekt dem Panel hinzugefügt. Die einzelnen Komponenten, Zustände und Transitionen werden dann „ausgepackt“ und angezeigt.

Weitere wesentliche Funktionen des AutomatonEditorPanel sind die Editier-Funktionen für Automaten. Per Rechtsklick auf die Editorfläche lassen sich neue Zustände hinzufügen. Das Erzeugen von Kanten geschieht durch Drag&Drop mit gedrückter STRG-Taste. Man kann auf diese Art dann ganz einfach eine Kante von einem Zustand zu einem weiteren ziehen. Details sind im Abschnitt 3.6.3.6 aufgeführt.

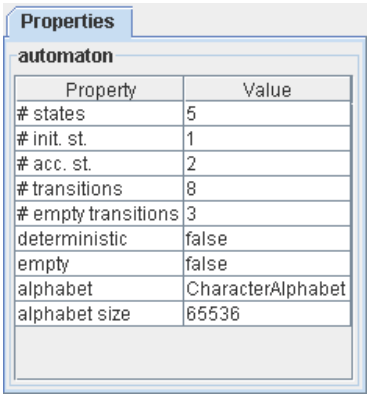


Abbildung 3.40: Anzeige von allgemeinen Automaten-Informationen im Workspace

Im Zuge der Integration in den Workspace übernahm das AutomatonEditorPanel auch die Anzeige von allgemeinen Automaten-Information im Property-Panel des Workspace. Dies ist insbesondere bei solchen Automaten sinnvoll, die zu groß sind, um sie als Graph zu überschauen. Das Panel benutzt zur Anzeige der Eigenschaften die eigens dafür geschriebene Methode `public Map<String, Object> getProperties()` aus der Klasse `aaa.automaton.Automaton`.

3.6.3.5 Klasse `AdvancedAutomatonPanel`

Die Klasse `AdvancedAutomatonPanel` bettet die Editierfläche in eine scrollbare Fläche ein, so dass auch große Automaten ausschnittsweise angezeigt werden können. Über ei-

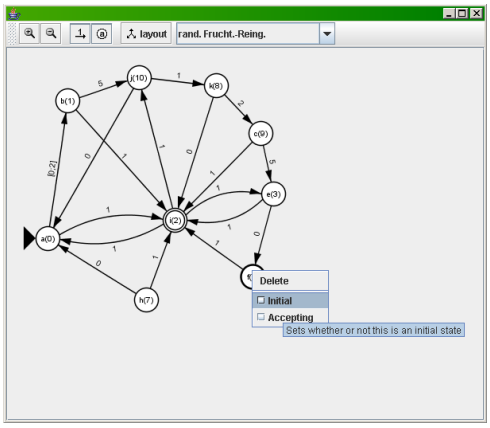


Abbildung 3.41: Das `AdvancedAutomatonPanel` im Einsatz.

ne Toolbar erhält der Anwender weitere Funktionen, zum Beispiel die Möglichkeit die Darstellung zu skalieren. Weitere Informationen befinden sich im Handbuch unter 3.7.8

Eine der wichtigsten Funktionen des AdvancedAutomatonPanel ist die Steuerung der Automaten-Layouter, welche im Abschnitt 3.6.6 noch genauer erläutert werden. Das Panel stellt eine Auswahl von Layoutern zur Verfügung und steuert den Layout-Vorgang.

Das Panel erbt von `javax.swing.JPanel` und ist somit problemlos in Swing-Applikationen (oder Applets) einzubinden.

3.6.3.6 Erstellen und Bearbeiten von Automaten

War es im ersten Semester bereits möglich, das Layout von dargestellten Automaten zu manipulieren, so waren weitergehende Modifikationen wie das Entfernen, Anlegen und Bearbeiten von neuen Zuständen und Transitionen nur bruchstückhaft und provisorisch möglich. Dies entsprach dem damaligen Planungsstand, die über die Anzeige hinausgehenden Editier-Funktionen im zweiten Semester zu erstellen.

Zahlreiche Detailveränderungen mussten zunächst vorgenommen werden, darunter z. B. das Erkennen von Tastaturereignissen und eine komfortable Möglichkeit, neue Transitionen aus Zuständen „herauszuziehen“. Auf die Implementation dieser Details soll hier nicht weiter eingangen werden, für eine Beschreibung der Anwendung sei hiermit auf das Benutzerhandbuch des Workspace verwiesen.

Das im ersten Semester programmierte Listener-System, welches beliebige, interessierte Objekte über Veränderungen des angezeigten Automaten durch den Benutzer signalisierte, wurde ausgebaut. Die im Abschnitt 3.6.3.1 beschriebenen Klassen und Methoden wurden dazu gründlich getestet und in Details optimiert. Damit war nun eine Grundlage geschaffen, um Veränderungen in der Darstellung in „Echtzeit“ an das

darunterliegende Automatenmodell weiterzureichen. Diese Aufgabe übernimmt die Klasse `AutomatonToPaintableConverter`, welche nun das Interface `PaintableAutomaton-StructuralChangeListener` implementiert und damit nun nicht mehr nur `Automaton`-Objekte in eine `PaintableAutomaton`-Darstellung konvertieren, sondern auch in Gegenrichtung arbeiten kann.

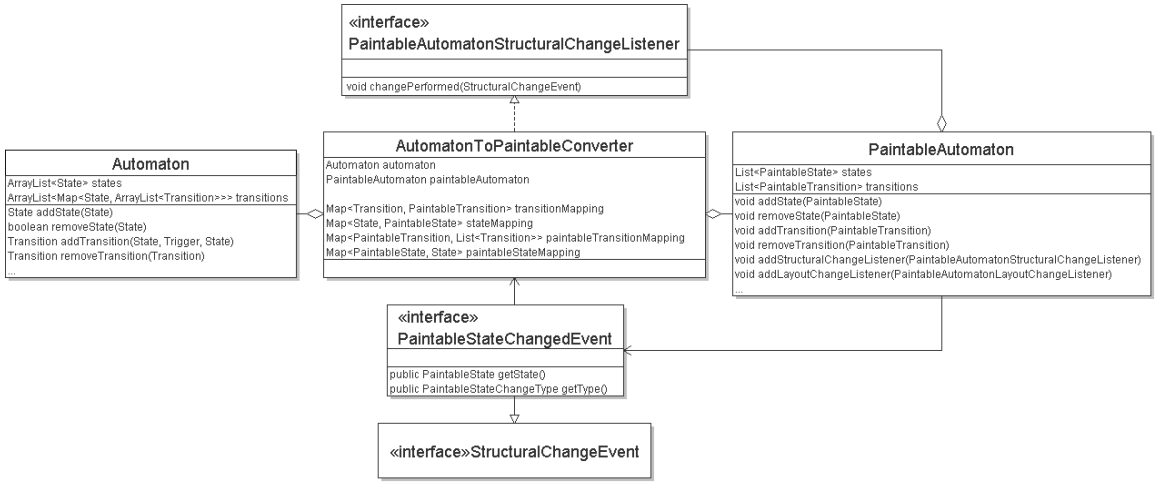


Abbildung 3.42: Klassendiagramm zur Rückkonvertierung von Änderungen an einem `PaintableAutomaton`

Abbildung 3.42 zeigt einen Ausschnitt dieses per Listenern realisierten Konvertierens. Im Zentrum steht der `AutomatonToPaintableConverter`, welcher ein `Automaton`-Objekt mit einem `PaintableAutomaton`-Objekt verbindet. Da der Konverter das Interface `PaintableAutomatonStructuralChangeListener` implementiert, kann er sich dem `PaintableAutomaton` als Interessent für Änderungen am im Editor dargestellten Automaten anmelden. Eine solche Änderung, zum Beispiel ein Hinzufügen eines neuen Zustands, erzeugt ein entsprechendes Event, hier ein `PaintableStateChangedEvent`. Mit diesem Event wird sowohl der neu erzeugte (oder gelöschte/veränderte) `PaintableState` übergeben als auch die Information, was mit diesem Zustand geschehen ist. Der Konverter kann diese Informationen auswerten und mittels seiner Verknüpfungs-Tabellen auf den entsprechenden `Automaton`-Äquivalenten die Änderung auch dort durchführen.

Als Besonderheit seien hier lediglich die ε-Transitionen genannt. Während im dargestellten (`Paintable-`)Automaten eine ε-Transition einfach durch ein weiteres Symbol ε im Transitions-Label dargestellt wird, so als wäre sie einfach ein Element einer Menge von Triggern, sind ε-Transitionen im (`aaa.automaton.Automaton`)-Automaten als eigenes, explizites Transitions-Objekt modelliert und liegen potentiell parallel zu einer weiteren, gelabelten Transition zwischen den gleichen Zuständen.

3.6.3.7 Das Speichern und Wiedereinladen von Automaten-Layouts

Eine wesentliche Neuerung im zweiten Semester stellte das Sichern und Wiedereinladen von Layout-Informationen dar. Zuvor wurde ein vom Benutzer mühevoll durchgeführtes Zurechtrücken von Zuständen nicht dauerhaft gespeichert, so dass nach dem Einladen eines gespeicherten Automaten der Standard-Layouter des Editors „zuschlug“ und mitunter ein nicht unbedingt ansprechendes Ergebnis lieferte.

Zur Sicherung eines Layouts reicht es aus, die Positionen und Dimensionen der Zustände zu speichern. Das Aussehen der Transitionen ist durch die angrenzenden Zustände eindeutig bestimmt.

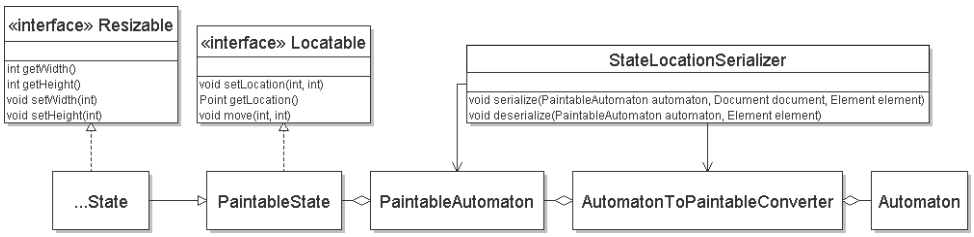


Abbildung 3.43: Überblick über die Funktionen um die Klasse `StateLocationSerializer`

3.6.3.7.1 Implementation Um die Zustandspositionen zu sichern, wird die Klasse `StateLocationSerializer` verwendet. Diese hat nur zwei öffentliche Methoden, einerseits `void serialize(PaintableAutomaton, Document, Element)`, welche die Zustandsinformationen eines `PaintableAutomaton` unterhalb eines übergebenen XML-`Elements` aus einem XML-`Document` sichert und die Methode `void deserialize(PaintableAutomaton, Element)`, welche sich um die Rückübertragung gespeicherter Positionsinformationen in einen `PaintableAutomaton` kümmert.

Zwischen einem dargestellten `PaintableAutomaton` und dem zugrundeliegendem `Automaton` gibt es eine 1:1-Abbildung der jeweiligen `PaintableState` und `State`-Objekte. Die von den `State`-Objekten zur Verfügung gestellte, eindeutige Zustands-Nummer wird dabei über den `AutomatonToPaintableConverter` für den Serialisierungs- und Deserialisierungs-Prozess der Positionsinformationen genutzt. Für jeden dargestellten Zustand wird seine Position (und ggf. Dimension) gesichert und mit der Zustandsnummer des zugehörigen `Automaton` `State`-Objekts versehen.

3.6.3.7.2 Die Interfaces `Locatable` und `Resizable` Um verschiedene `Paintable`-Modelle zu unterstützen, werden zwei Interfaces eingesetzt, welche jeweils von einer Zustands-Klasse implementiert werden können. Abhängig davon, welche Interfaces implementiert sind, wird zur Laufzeit entschieden, ob nur die Positionsdaten, oder auch die

Dimensions-Informationen (Höhe & Breite) gespeichert werden.

3.6.4 Kripke-Modelle

Die Projektgruppe entschied sich im zweiten Semester, das Projekt mehr in Richtung Model Checking zu bewegen. Neben den dazu notwendigen Automaten und Transformations-Komponenten war es auch wichtig, überhaupt zu überprüfende Modelle eingeben zu können. Natürlich bot es sich an, dies einen Benutzer komfortabel über den bereits existierenden Editor erledigen zu lassen. Da Kripkemodelle, wie Automaten, relativ simple Graphen sind, war eine Erweiterung des Editors glücklicherweise schnell möglich. Das Ergebnis sieht man in Abbildung 3.44.

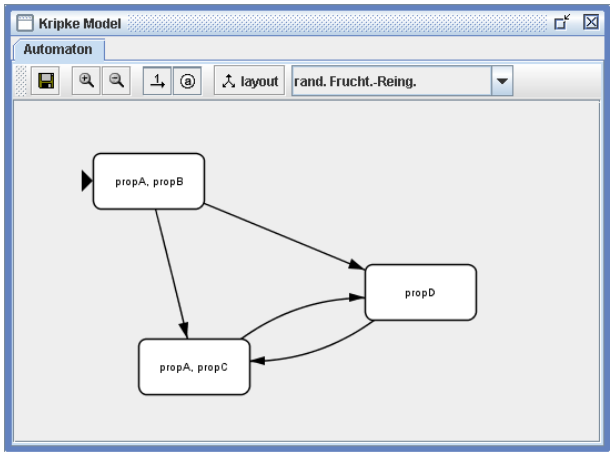


Abbildung 3.44: Beispiel eines Kripke-Modells im Editor

3.6.4.1 Implementation

Wie bereits erwähnt, sind Kripkemodelle strukturell nicht sehr unterschiedlich von Automaten: es sind ebenfalls Graphen mit Knoten und Kanten. Daher bot es sich an, das bestehende `PaintableAutomaton`-Modell passend per Vererbung zu erweitern. Abbildung 3.45 zeigt das vereinfachte Klassendiagramm des neu entstandenen `PaintableKripkeModel`-Packages.

Wie man sieht, erben alle neuen Klassen von bestehenden alten Klassen. Die durch die Vererbung eingebrachten Unterschiede sind im Diagramm durch Erwähnung der wichtigsten veränderten Methoden und Attribute angedeutet und werden nun kurz erklärt:

3.6.4.1.1 `KripkeEditorPanel` Die bestehende Klasse `AutomatonEditorPanel` wurde dahingehend verändert, dass die für das Erzeugen neuer Zustände verantwortliche Aktion

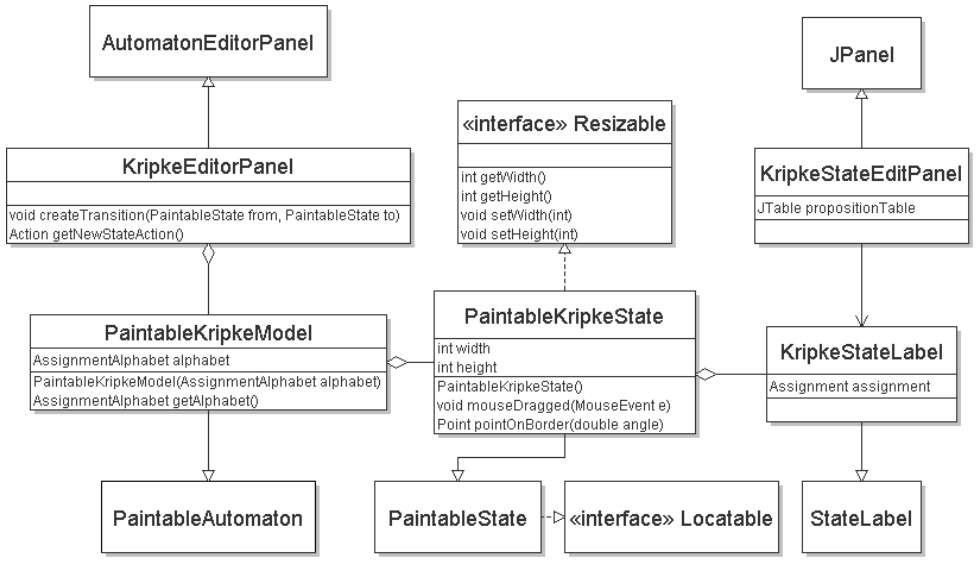


Abbildung 3.45: Das Klassendiagramm des `PaintableKripleModel`-Packages

nun Objekte des Typs `PaintableKripkeState` erzeugt. Noch einfacher war die Anpassung der für die Transitions-Erzeugung verantwortlichen Methode `createTransition`: Die neu erzeugte Transition wird einfach ohne ein Label ausgeliefert, da Transitionen in Kripkemodellen üblicherweise nicht beschriftet sind.

3.6.4.1.2 `PaintableKripkeModel` Die neue Klasse zur Aufnahme eines darstellbaren Kripkemodells unterscheidet sich von der Basisklasse `PaintableAutomaton` nur dadurch, dass nun ein `AssignmentAlphabet` verwendet wird.

3.6.4.1.3 `PaintableKripkeState` Die Klasse für Kripkezustände ist die am meisten modifizierte Klasse bei den Kripkemodellen. Neue Attribute für die Verwaltung von Höhe und Breite, welche nun durch den Benutzer veränderbar sind, kamen hinzu. Dies wird auch durch das Interface `Resizable` kenntlich gemacht, welches beim Persistieren von Layout-Informationen Bedeutung hat. Die Form eines Zustandes ist hier rechteckig und nicht mehr rund wie bei der Basisklasse `PaintableState`. Daher musste auch die Methode `Point pointOnBorder(double angle)`, welche für die korrekte Positionierung von Transitions-Anfang/Ende notwendig ist, erweitert werden. Die umfangreichsten Änderungen betrafen die Maus-Interaktion zur Größenveränderung. `PaintableKripkeState` eignet sich als Ausgangsklasse für mögliche zukünftige Editor-Komponenten, welche nicht nur in ihrer Position, sondern auch in ihrer Größe bearbeitet werden sollen.

3.6.4.1.4 `KripkeStateLabel` Zur Darstellung von für einen Zustand aktiven Propositionen wurde die Klasse `StateLabel` passend erweitert. Das `KripkeStateLabel` verwendet

direkt die `Assignment`-Objekte, wie sie auch im `Automaton`-Package verwendet werden.

3.6.4.1.5 `KripkeStateEditPanel` Um die Zustands-Propositionen bearbeiten zu können wurde das `KripkeStateEditPanel` entwickelt. Dieses wird im Triple-A-Workspace verwendet, indem es bei Selektion eines Kripkezustandes im Property-Panel des Workspaces angezeigt wird. Abbildung 3.46 zeigt ein solches Panel im Einsatz: per Checkbox können vom Benutzer die Propositionen eines Zustandes aktiviert und deaktiviert werden.

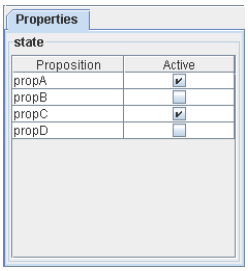


Abbildung 3.46: Bearbeitung der aktiven Propositionen bei einem Zustand in einem Kripkemodell

3.6.5 OBDDs

OBBDs wurden im zweiten Semester als Hilfsmittel für das Model Checking mit LTL-Formeln eingeführt. Da die Klasse `OBDD` von `Automaton` erbt, war es fast ohne Änderungen möglich, OBDDs im Editor anzuzeigen. Natürlich entsprach die Darstellung in ihrer Form der von Automaten und entsprach nicht den gewohnten OBDD-Konventionen. Es musste also ein weiteres Package für den Editor erstellt werden, um eine Darstellung zu erreichen, wie sie zum Beispiel in Abbildung 3.12 zu sehen ist.

3.6.5.1 Implementation

Wie bei den Kripkemodellen, so war es auch bei den OBDDs möglich, die neuen Klassen als Ableitungen von bereits bestehenden Klassen zu modellieren. Abbildung 3.47 zeigt die neuen Klassen.

3.6.5.1.1 Die Klasse `PaintableOBDDSink` Während die inneren Knoten eines OBDDs in ihrer Form mit der normalen Darstellung von Automaten-Zuständen übereinstimmen (Kreise), werden OBDD-Senken üblicherweise als Rechtecke dargestellt. Die Klasse `PaintableOBDDSink` erweitert daher `PaintableState` so, dass die Kreisform

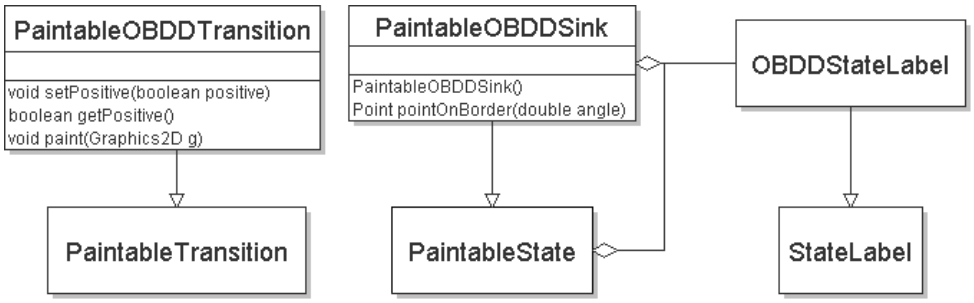


Abbildung 3.47: Klassendiagramm des (Paintable)`OBDDs`-Packages

durch ein Rechteck abgelöst wird. Für innere OBDD-Verzweigungsknoten wird weiterhin die normale `PaintableState`-Klasse benutzt.

3.6.5.1.2 Die Klasse `OBDDStateLabel` Die Darstellung von OBDD-Knoten wurde nur dahingehend verändert, dass die von den Automaten kommende Zustands-Nummerierung bei OBDDs nicht mehr sichtbar ist.

3.6.5.1.3 Die Klasse `PaintableOBDDTransition` Bei OBDDs werden die Verzweigungen durch durchgezogene oder gestrichelte Kanten dargestellt. Die neue Klasse `PaintableOBDDTransition` übernimmt diese Darstellungsform und ersetzt damit ebenfalls die nun redundante Anzeige einer Beschriftung der OBDD-Kanten.

3.6.6 Layouting von Automaten

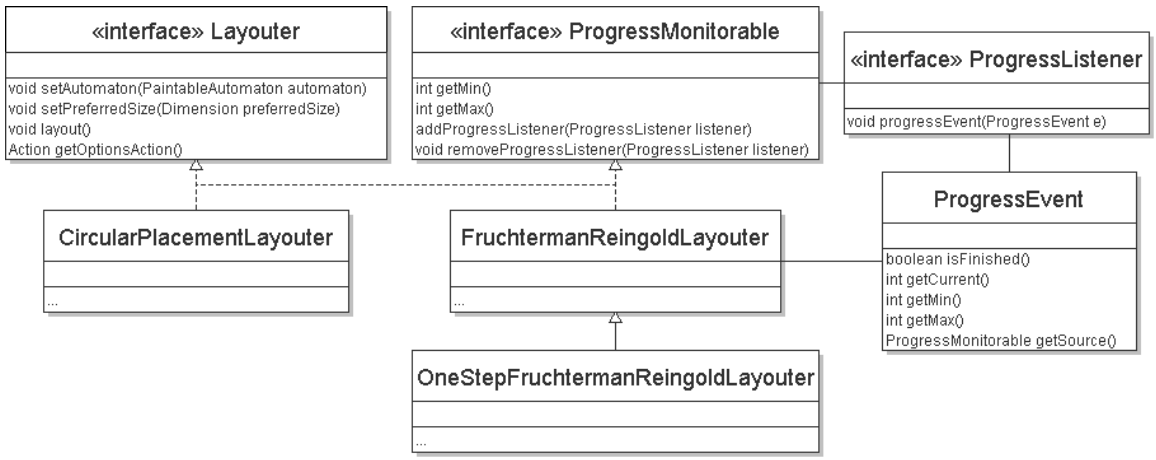


Abbildung 3.48: Die wichtigsten Klassen und Methoden der Layouting-Funktionalität des PaintableAutomaton-Addons

Ein interessantes Problem beim Anzeigen von Automaten, oder allgemeiner von Graphen, war das Realisieren eines automatischen Layoutings. Diese Funktionalität wurde im

PaintableAutomaton-Framework geschaffen, dessen wichtigsten Klassen im 3.48 dargestellt sind.

3.6.6.1 Interface `Layouter`

Um als Layouter für das Framework benutzbar zu sein, muss eine Klasse das Interface `Layouter` implementieren. Dazu sind vier Methoden zu implementieren, die fast alle selbsterklärend sind:

`public void setAutomaton(PaintableAutomaton automaton)` übergibt dem Layouter den Automaten, der zu layouten ist.

`public void setPreferredSize(Dimension preferredSize)` übermittelt dem Layouter eine bevorzugte Endgröße für den aus dem Layoutvorgang hervorgehenden Automaten.

`public void layout()` führt den Layout-Vorgang aus.

`public Action getOptionsAction()` liefert, sofern vom Layouter unterstützt, ein `Action`-Objekt, welches zur Anzeige eines Optionsmenüs oder Ähnlichem benutzt werden kann.

`public boolean canLayout(PaintableAutomaton automaton)` liefert zurück, ob der Layouter den übergebenen `PaintableAutomaton` layouten kann. Diese Methode wurde im zweiten Semester hinzugefügt, da mit den neu hinzugekommenen spezialisierten Layoutern erstmals Inkompatibilitäten entstanden. So kann z. B. der OBDD-Layouter nur OBBDs layouten und kann mit anderen Automaten nichts anfangen.

3.6.6.2 Interface `ProgressMonitorable`

Da das Layouten unter Umständen längere Zeit in Anspruch nehmen kann (man bedenke, dass bereits zahlreiche Entscheidungsprobleme auf Graphen NP-vollständig sind), ist es vorteilhaft, wenn der Layoutvorgang nicht die komplette Anwendung blockiert und der Anwender in der Zwischenzeit an anderen Automaten weiterarbeiten kann.
Dies kann erreicht werden, indem ein Layouter zusätzlich das Interface `ProgressMonitorable` implementiert. Die Implementierung erfordert, dass der Layouter Listener zur Verfügung stellt, mit denen die benutzende Anwendung, also bei uns der Workspace, über den aktuellen Fortschritt informiert wird.
Sobald ein Layouter das Interface implementiert, wird er automatisch in einem eigenen Thread gestartet und läuft in diesem nebenläufig bis zur Vollendung des Layoutvorgangs.

3.6.6.3 Fruchterman-Reingold Algorithmus

Nach den durchaus positiven Erfahrungen des Fruchterman-Reingold-Layouters im JUNG-Framework sollte dieser auch im PG-Editor implementiert werden. Das Verfahren von Thomas M. J. Fruchterman und Edward M. Reingold wurde 1991 in deren Paper *Graph Drawing by Force-directed Placement* [**?**] vorgestellt und anhand des Papers für unseren Editor implementiert.

Das Verfahren beruht auf der Simulation eines physikalischen Modells, bei dem man die Knoten eines Graphen als sich abstoßende Körper betrachtet, und sich die Kanten als Federn vorstellt, die ihre inzidenten Knoten zueinander zu ziehen versuchen.
Verteilt man zunächst die Knoten zufällig und simuliert dann den oben skizzierten Prozess, so stellt sich nach einer gewissen Anzahl von Iterationen ein Zustand (nicht notwendigerweise globaler) Gesamtkräfte ein. Der so erzeugte Endzustand erfüllt zumindest annähernd folgende Eigenschaften:

- gleichverteilte Knoten
- gleiche Kantenlängen
- Wiederspiegelung von Symmetrien im Graphen

Der Algorithmus enthält keinerlei Mechanismen zur möglichst planaren Darstellung von Graphen, so dass das Ziel der weitestgehend kreuzungsfreien Darstellung von Kanten nicht immer erreicht wird. Die Algorithmenerzeugnisse sind jedoch, sofern eine gewisse, ohnehin als unübersichtlich zu betrachtende Anzahl von Knoten und Kanten nicht überschritten wird, durchaus als ästhetisch, zumindest aber als lesbar zu bezeichnen.

3.6.6.3.1 Kräftegleichgewicht Sei k die als optimal angesehene Kantenlänge und d die Distanz zweier Knoten, dann ist $f_a(d) = d^2/k$ die Anziehungskraft, die eine Kante auf ihre inzidenten Knoten ausübt und $f_r(d) = k^2/d$ die Abstoßungskraft, welche zwei beliebige Knoten jeweils zueinander ausüben.
Setzt man beide Kräfte gleich, so erhält man ein Gleichgewicht bei der Distanz $d = k$.

Fruchterman und Reingold empfehlen als ein sinnvolles k den Wert $k = C\sqrt{\frac{hoehe \cdot breite}{|V|}}$, wobei *hoehe* und *breite* die gewünschten Dimensionen der Zeichenfläche und C eine „experimentell zu bestimmende Konstante“ darstellen.

Bei eigenen Experimenten erwies sich dieses Vorgehen bei kleinen Graphen als sinnvoll. Bei Graphen mit vielen Knoten und Kanten führte die oben genannte Wahl des k zu

einer Verklumpung des resultierenden Graphen, ohne dabei die bereitgestellte Fläche auszufüllen. Dies ist auf das bei vielen Knoten kleine k und die durch die vielen Kanten starken Anziehungskräfte zurückzuführen.
Eine sinnvolle Lösungsoption war es, das k einfach konstant zu wählen.

3.6.6.3.2 Algorithmus Der Algorithmus des Layouters wird hier kurz als stark verkürzter Java-Pseudo-Code vorgestellt. Die Methoden `calculateRepulsion` und `calculateAttraction` aktualisieren jeweils einen für jeden Knoten existierenden Kraft-Vektor, welcher in der Methode `moveVertex` dann zu einer Positionsänderung des Knotens führt.

```
for(int i = 0; i < maxIterations; i++) {
  for(Vertex v : vertices) {
    for(Vertex u : vertices) {
      if (v != u) {
        calculateRepulsion(u, v);
      }
    }
  }

  for(Edge e : edges) {
    calculateAttraction(e);
  }

  for(Vertex v : vertices) {
    moveVertex(v);
  }
}
```

Die Bewegung der Knoten in `moveVertex` wird, dem *Simulated Annealing*-Schema folgend, durch eine „Temperatur“ begrenzt. Durch eine Begrenzung der Temperatur wird die Maximalbewegung der Knoten pro Iteration eingeschränkt, mit dem Ziel, dass Minima im Kräfteverhältnis nicht einfach „übersprungen“ werden. Um gleichzeitig nicht der Gefahr ausgeliefert zu sein, in (schlechten) lokalen Minima festhängen zu bleiben, wird zu Beginn eine hohe Temperatur gewählt, die dann mit steigender Iterationsanzahl abnimmt.

Beim Experimentieren mit dem Algorithmus wurde beobachtet, dass Graphen die aus mehreren Zusammenhangskomponenten bestehen, unvorteilhaft layoutet wurden: bedingt durch die wechselseitigen Abstoßungskräfte, trieben die einzelnen Zusammenhangskomponenten sehr weit auseinander. Als Lösung für dieses Problem (das zugegebenermaßen

bei Automaten nicht wirklich gegeben ist) wurde die Reichweite der Abstößungskräfte begrenzt. Als gute Heuristik hat sich eine Maximalreichweite der Abstoßungskraft von $2 \cdot k$ Einheiten herausgestellt.

Unsere Implementation des Fruchterman-Reingold-Algorithmus benutzt 500 Iterationen bei einer linear absteigenden Temperatur, die jedoch über einen konfigurierbaren Mindestwert gehalten wird. Der Algorithmus ist in der Klasse `FruchtermanReingoldLayouter` im Package `aaa.workspace.editor.layout` implementiert.

Die durchgeführten Modifikationen sowie die Parameter des Algorithmus, wie Iterationszahl, Temperaturfunktion etc. sind im Übrigen gute Beispiele für eine mögliche Implementation eines Optionen-Dialogs, wie sie im Interface `Layouter` durch die Methode `getOptionsAction()` ermöglicht wird.

3.6.6.4 Kreisförmige Anordnung

Neben dem Fruchterman-Reingold-Layouter steht noch ein wesentlich einfacherer Layouter zur Verfügung, welcher einfach alle Zustände auf einem gedachten Kreis positioniert. Dieser Layouter, welcher als Klasse `CircularPlacementLayouter` implementiert ist, eignet sich vor allem für größere Graphen, die ihrer Komplexität wegen nicht sinnvoll vom Fruchterman-Reingold-Algorithmus layoutet werden können. Zwar ist die Lesbarkeit des Graphen in diesem Fall bei kreisförmiger Anordnung nicht unbedingt besser, aber der Graph wird wesentlich schneller aufbereitet.

Im zweiten Semester wurden neue Automaten-Modelle, bzw. auch eine OBDD-Datenstruktur implementiert. Für letztere ist beispielsweise ein Layouting, wie es der Fruchterman-Reingold-Algorithmus leistet, nicht unbeding optimal. Das hoch motivierte Automaton-Team nutzte die Layouter-Schnittstelle und erstellte eigene Layouter, welche nun folgend kurz beschrieben werden.

3.6.6.5 OBBD-Layouter

Um OBDDs in gewohnter Weise als einen von oben wachsenden Baum darzustellen, hat das Automaton-Team mit der Klasse `OBDDLayouter` einen Layouter geschaffen, der das Gewünschte erledigt. Der Layouter nutzt die über die Klasse `OBDDState` ermittelbaren Informationen zur Einteilung der Knoten in Ebenen, wobei die Variablenordnung des OBDDs natürlich berücksichtigt wird. Anschließend werden die Knoten einer Ebene möglichst ansprechend verteilt.

3.6.6.6 Level-Layouter

Ebenfalls aus dem Automaton-Team stammt die Klasse `LevelLayouter`. Dieser ordnet Zustände entsprechend ihres Abstands vom Startzustand aus an. Dabei wird der Startzustand an den linken Rand geschoben und anschließend alle Zustände gleicher Ebene in Spalten nach rechts gehend angeordnet.

3.6.6.7 Partitions-Layouter

Die Klasse `PartitionedLayouter` wurde vom Automaten-Team entwickelt und ordnet Zustände horizontal nach ihrer Partitionszahl an. Die Partitionszahl wird dabei dem Label der Zustands-Objekte entnommen und muss durch ein `Integer`-Objekt repräsentiert werden. Anwendung kann dieser Layouter zum Beispiel bei der Darstellung von degeneralisierten Büchiautomaten finden.

3.7 Handbuch

Ein wichtiger Punkt dieser Projektgruppe war es, eine Benutzeroberfläche zu schaffen, die dazu dient, die Bedienung verständlicher, einfacher und effektiver zu machen. Dieses Handbuch behandelt diese graphische Benutzeroberfläche der Projektgruppe. Es gibt eine umfassende Übersicht über die Funktionen und eine Einführung in alle Verfahren, die im Umgang mit dem Workspace erforderlich sind. An dieser Stelle sei darauf hingewiesen, dass hier keine theoretischen Grundlagen der Automaten oder Analyseverfahren erklärt werden. Diese befinden sich an anderer Stelle im Endbericht.

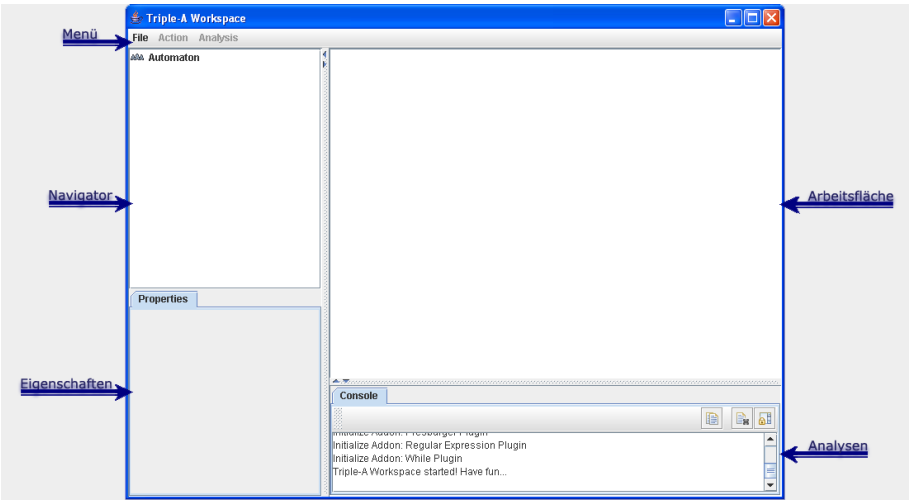


Abbildung 3.49: Triple-A Workspace

3.7.1 Der Workspace-Überblick

In der Abbildung 3.49 sieht man die Grundelemente des Workspace.

3.7.1.1 Menü

Wie in jeder gängigen Computeranwendung organisiert das Menü alle Funktionen dieses Programms. Zum Beispiel lassen sich im *File-Menu* neue Objekte erzeugen, ausgewählte Objekte speichern oder laden, sowie das Programm beenden.

3.7.1.2 Die Arbeitsfläche

Alle neuen Analysen oder Automatenkonstruktionen werden in separaten Fenstern auf der Arbeitsfläche abgelegt.

3.7.1.3 Navigator

Der Navigator verwaltet alle erzeugten Automatenanalysen nach ihren zugehörigen Plugins. So behält man alle Fenster auf der Arbeitsfläche im Blick. Sie dient außerdem der Auswahl von Automaten, auf denen Aktionen ausgeführt werden sollen.

3.7.1.4 Eigenschaften

In diesem Bereich werden Zusatzinformationen zur Automatenkonstruktion angezeigt. Einige Dialoge können zum Bearbeiten genutzt werden.

3.7.1.5 Analyse

Der Analysebereich informiert über alle wichtigen Informationen, die während einer Analyse oder der Bearbeitung auftreten. Einen festen Bestandteil des Analysebereiches stellt die Konsole dar. In ihr werden Sonderinformationen ausgegeben, wie z. B. geladene Plugins oder Fehlermeldungen. Weitere Analyseinhalte werden später im Kapitel Plugins des Handbuchs beschrieben.

3.7.2 Starten von neuen Automatenanalysen

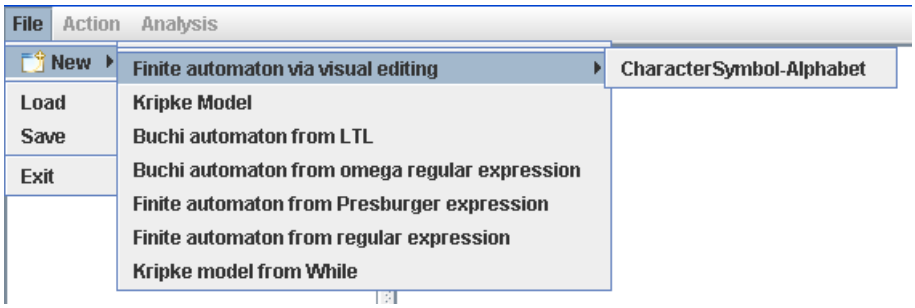


Abbildung 3.50: Triple-A Workspacemenü

Über den Menüeintrag `File - New` erreicht man alle installierten Automatenanalysen. Die Einträge in diesem Menü können je nach installiertem Plugins abweichen. Abbildung 3.50 zeigt alle Basisplugins. Diese sind:

Finite automaton via visual editing Über diese Plugin lässt sich ein leerer Automat im Editor manuell zusammenbauen.

Kripke Model Manuell wird hier ein Kripkemodell im Editor erstellt. Dieses kann dann zu einem Büchiautomaten konvertiert werden. Dieses Plugin ist für Model Checking geeignet.

Büchi Automaton from LTL Erstellt einen Büchiautomaten auf Basis einer LTL-Spezifikation.

Büchi Automaton from omega regular expression Ein Büchiautomat aus einem ω-regulären Ausdruck kann hier erzeugt werden.

Finite Automaton from Presburger expression Erzeugt einen endlichen Automaten zu einer Formel der Presburger Arithmetik.

Finite Automaton from regular expression Reguläre Ausdrücke können mit diesem Plugin in endliche Automaten transformiert werden.

Kripke Model from While Eine Eingabe aus einer einfachen While-Sprache wird hier in ein Kripkemodell transformiert.

3.7.3 Umgang mit dem Navigator

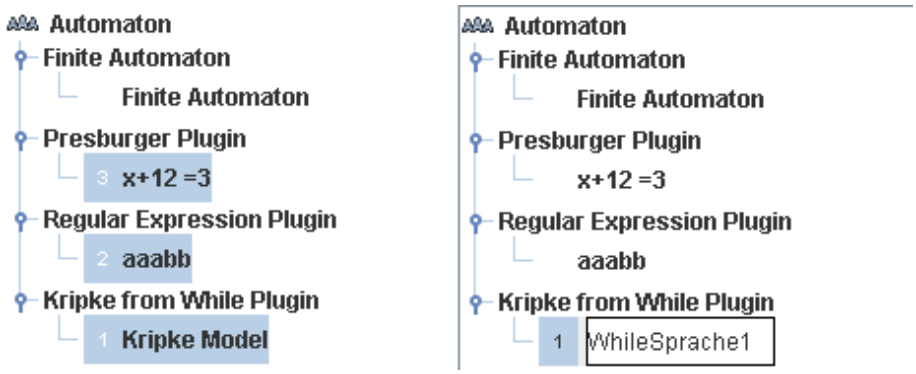


Abbildung 3.51: Auswahl von Automaten im Navigator / Umbennen von Automaten im Navigator

Der Navigator erfüllt mehrere Funktionen. Klickt man in ihm einen Eintrag an, so wird das zugehörige Automatenfenster auf der Arbeitsfläche in den Vordergrund geholt. Klickt man zweimal auf diesen Eintrag, so wird das Fenster zusätzlich noch maximiert oder wieder reduziert.

Einträge können umbenannt werden, indem man einen Eintrag anklickt und die Maus auf dem Eintrag belässt, bis der Eintrag bearbeitbar ist (zu sehen in Abbildung 3.51 auf der rechten Seite). Dann kann man an dieser Stelle einen neuen Namen eingeben, dieser muss mit Enter bestätigt werden, ansonsten wird der alte Eintrag wiederhergestellt. Um die

Einträge eindeutig zu halten, wird der Eintrag um eine Zahl ergänzt, sofern ein Eintrag mit dem selben Namen bereits existiert.

Die linke Seite der oben erwähnten Abbildung zeigt das Markieren von Einträgen im Navigator. Hierbei werden die gängigen Methoden `UMSCHALT - Mausklick` und `STRG - Mausklick` unterstützt. Die Nummern vor der Auswahl geben die Reihenfolge der ausgewählten Automatenanalysen an. Automatenoperationen sind abhängig von der Auswahl in der Automatenliste.

3.7.4 Die Konsole

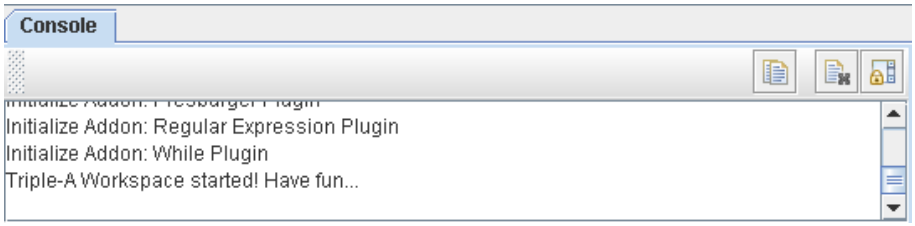


Abbildung 3.52: Die Konsole

Die Konsole zeigt einige ausgeführte Aktionen, Fehlermeldungen und Informationen vom Workspace an. Hier wird z. B. beim Start des Workspace angezeigt, welche Plugins installiert worden sind.

Die Konsole ist mit drei Aktionen ausgestattet. Siehe Abbildung 3.52, die von links nach rechts folgende Funktion erfüllen:

- Mit der ersten Schaltfläche kann der aktuelle Inhalt der Konsole in die Zwischenablage kopiert werden.
- Über diese Schältfläche wird der gesamte Inhalt der Konsole gelöscht.
- Normalerweise scrollt die Konsole bei jedem neuen Eintrag in ihr automatisch mit. Diese Funktion kann über den dritten Button ausgeschaltet oder eingeschaltet werden.

3.7.5 Laden

Liegen gespeicherte Automatenanalysen vor, so können diese über den Menüpunkt `File - Load` wieder eingeladen werden. Hierzu wählt man in einem Dateidialog die XML-Datei aus, die man laden möchte. Sollte die angegebene XML-Datei ein fehlerhaftes Format haben oder beschädigt sein, so wird eine Fehlermeldung ausgegeben.

Ist die XML-Datei gültig, so werden alle gespeicherten Automatenanalysen eingeladen. Ist in der Datei eine Automatenanalyse mit einem unbekannten Plugin gespeichert worden, so wird auch hier der Anwender darüber informiert. Die entsprechende Automatenanalyse kann leider nicht geladen werden, bevor dieses Plugin nicht installiert worden ist. Alle anderen Automatenanalysen, für die ein Plugin vorhanden ist, werden geladen.

Beim Laden von Automatenanalysen kann es zu Umbenennungen kommen, sofern eine Automatenanalyse mit dem angegebenen Namen bereits im Workspace existiert. Hierbei wird dem doppelten Eintrag wieder eine Zahl angehängt. Es ist also empfehlenswert, vor dem Laden von Dateien alle anderen Automatenfenster zu schließen.

3.7.6 Speichern

Möchte man eine oder mehrere Automatenanalysen speichern, so geht dieses sehr einfach. Zuerst markiert man im Navigator alle Automatenanalysen, die man speichern möchte. Danach ruft man den Menüpunkt `File - Save` auf. Hier kann man den Speicherort und den Dateinamen festlegen. Bestätigt man mit *OK*, so werden alle gewählten Automaten in ein XML-Format konvertiert und am angegebenen Ort gespeichert.

3.7.7 Operationen und Analysen

Um Operationen und Analysen auf einem oder mehreren Automaten ausführen zu können, müssen diese zuerst ausgewählt werden. Dies geschieht, indem man das Automatenfenster direkt oder die Einträge im Navigator auswählt. Je nach Auswahl wird dynamisch das Aktionen- und Analysemenü mit Operationen gefüllt. Sind keine gültigen Aktionen verfügbar, so sind die entsprechenden Menüs ausgegraut.

3.7.8 Automateneditor

Der Automateneditor dient innerhalb des Workspaces sowohl zur Anzeige als auch zur Bearbeitung von Automaten. Es stehen zahlreiche Funktionen zur Verfügung, die im Folgenden beschrieben werden.

3.7.8.1 Die Toolbar

Über die in Abbildung 3.53 dargestellte Toolbar lassen sich einige Funktionen, welche die Darstellung von Automaten betreffen, beeinflussen. Die in der Abbildung dargestellten

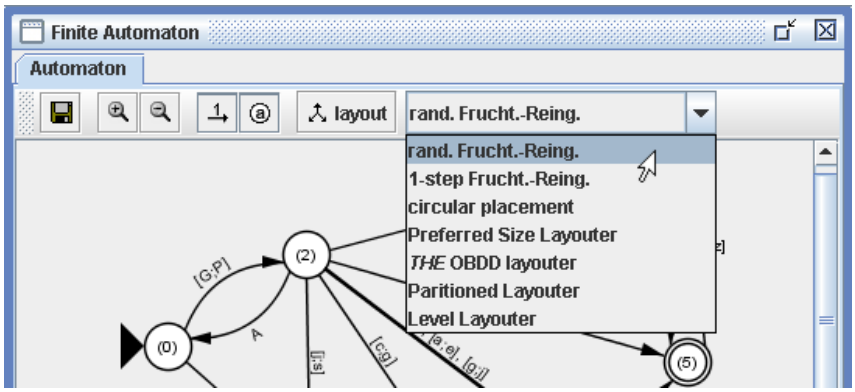


Abbildung 3.53: Die Editor-Toolbar

sieben Schaltflächen haben folgende Funktionen:

Bild speichern Mit dieser Funktion kann ein Bild des dargestellten Automaten im Dateisystem abgelegt werden. Das Bildformat ist `.png`.

Vergrößern Ein Druck auf diese Schaltfläche vergrößert proportional die Abstände zwischen allen Zuständen, so dass der Automat größer erscheint.

Verkleinern Bewirkt das Gegenteil der vorherigen Schaltfläche und „zoomt“ durch Verkleinerung der Abstände zwischen den Zuständen aus der Automatendarstellung heraus.

Transitionsbeschriftungen ein-/ausblenden Aktiviert oder deaktiviert die Darstellung von Labels an Transitionen. Dies kann insbesondere bei mit OBDDs gelabelten Transitionen sinnvoll sein, da deren Darstellung üblicherweise wesentlich länger als die Transition selbst ist. Auch bei ausgeblendeten Beschriftungen lassen sich Informationen über Transitionen selbstverständlich durch einfaches Anklicken einer Transition abrufen, wobei dann im PropertyPanel des Workspaces ein Informations-Panel eingeblendet wird.

Zustandsbeschriftungen ein-/ausblenden Aktiviert oder deaktiviert, analog zur vorherigen Aktion, Labels an den Zuständen.

Layouting durchführen Startet den unmittelbar rechts von dieser Schaltfläche ausgewählten Layouter.

Layouter auswählen In dieser Liste kann ein Layouter ausgewählt werden und dann anschließend mit der links daneben befindlichen Schaltfläche gestartet werden. Der Fruchterman-Reingold-Layouter ist standardmäßig vorselektiert.

3.7.8.2 Das Bearbeiten von Automaten

3.7.8.2.1 Erzeugen neuer Zustände Neue Zustände können über das Kontextmenü, welches bei einem Rechtsklick auf der Editierfläche angezeigt wird, erzeugt werden. Es ist der Menüeintrag „New state“ zu wählen.

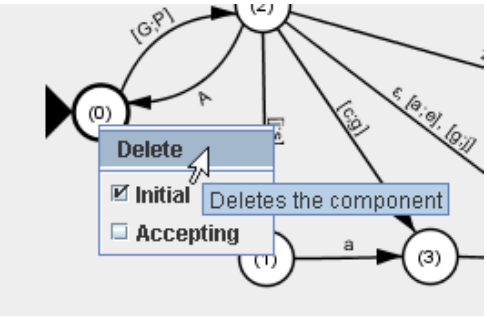


Abbildung 3.54: Optionen für Zustände

3.7.8.2.2 Löschen von Zuständen und Zustandsoptionen Durch Rechtsklick auf einen Zustand lässt sich ein Zustand über den Eintrag „Delete“ entfernen. Alternativ kann ein Zustand durch Drücken der Taste ENTF gelöscht werden. Zusätzlich werden, abhängig vom aktuellen Automaten- und Zustandstyp, weitere Optionen angezeigt. Abbildung 3.54 zeigt zum Beispiel das Menü bei einem Zustand eines endlichen Automaten. Hier lässt sich ein Zustand auch zu einem Start- oder akzeptierenden Endzustand erklären.

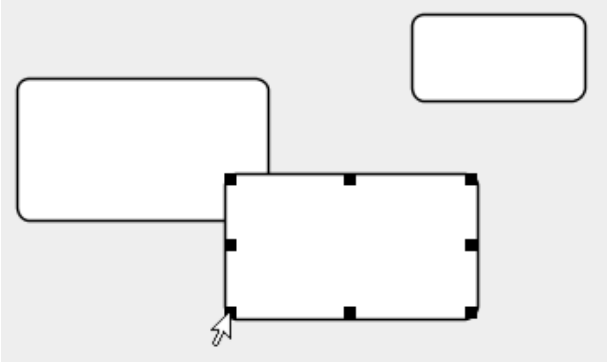


Abbildung 3.55: Größenveränderung an Zuständen

3.7.8.2.3 Position und Größe von Zuständen Per Drag&Drop können Zustände an beliebige Positionen verschoben werden. Für Transitionen ist dies nicht möglich, aber auch nicht notwendig, da diese immer fest zwischen zwei Zuständen hängen und sich bei deren Bewegungen mitbewegen.
Bei normalen Automaten ist die Größe der dargestellten Zustände fest und kann nicht verändert werden. Je nach Plugin sind aber auch in der Größe bearbeitbare Zustände

möglich, so etwa bei den Kripkemodellen. In diesem Fall lässt sich die Größe wie in gängigen Anwendungen durch Anklicken eines Zustandes und anschließendem Drag&Drop auf einem der 8 dargestellten Punkte verändern. Abbildung 3.55 zeigt einen solchen Vorgang.

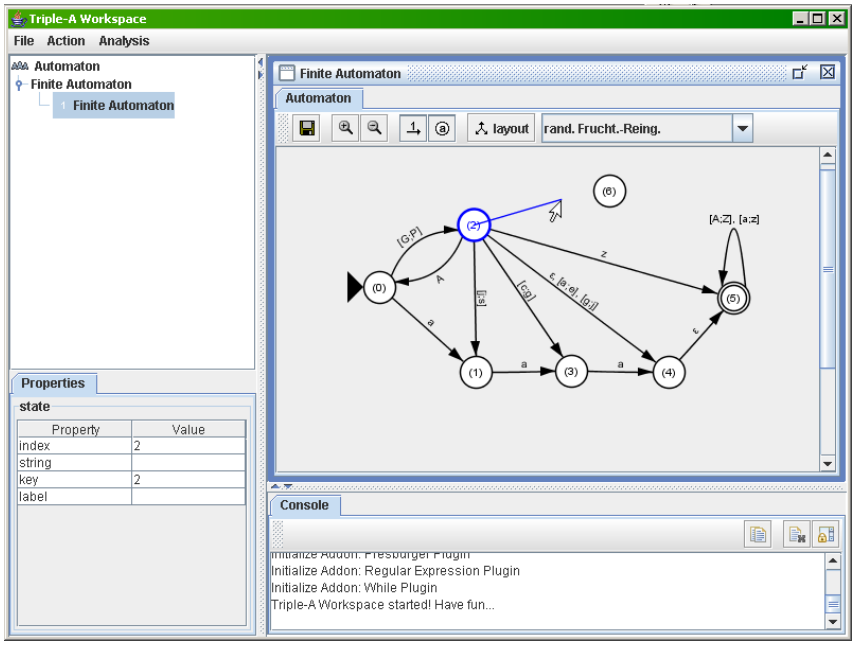


Abbildung 3.56: Eine neue Transition wird durch Drücken der STRG-Taste und das Ziehen der Maus zwischen zwei Zuständen erstellt

3.7.8.2.4 Erzeugen von neuen Transitionen Transitionen werden erstellt, indem sie aus einem Quellzustand „herausgezogen“ und an einem Zielzustand „losgelassen“ werden. Dazu drückt man die STRG-Taste und hält diese gedrückt. Mit der Maus klickt man nun auf den Quellzustand und hält dabei die Maustaste gedrückt. Nun kann man per Drag&Drop eine Transition aus dem Zustand herausziehen und über einem beliebigen Zielzustand loslassen. Bei Loslassen wird dann, sofern noch keine Transition vorhanden war, eine neue erstellt. Während des Drag&Drop-Vorgangs werden der Quell- und der jeweils „überflogene“ Zielzustand blau eingefärbt. Abbildung 3.56 zeigt das Erzeugen einer neuen Transition auf dem halben Weg zwischen dem Quellzustand (2) und dem Zielzustand (6).

3.7.8.2.5 Löschen von Transitionen Das Löschen von Transitionen funktioniert analog zum Löschen von Zuständen über das Kontextmenü der rechten Maustaste oder durch Drücken der Taste ENTF.

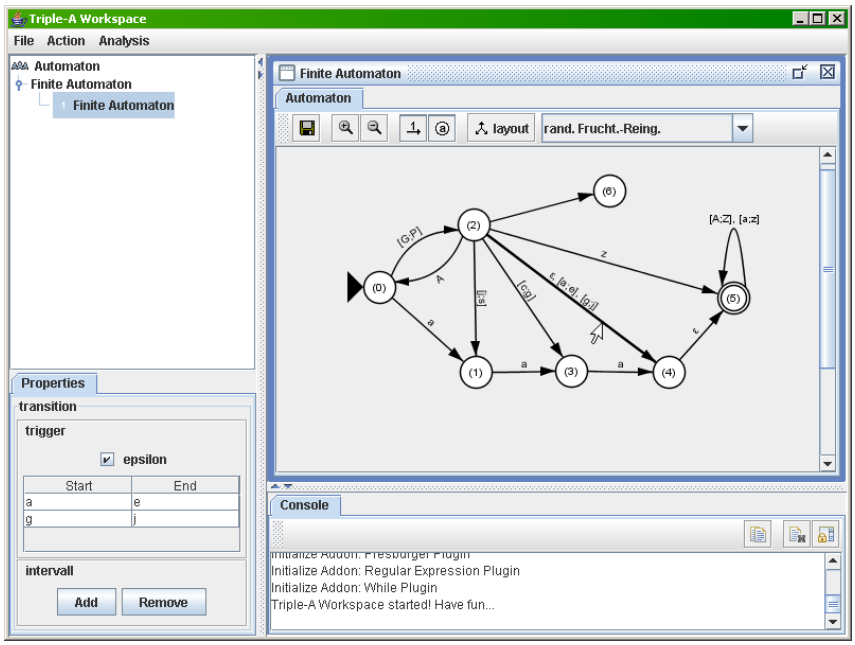


Abbildung 3.57: Bearbeiten einer Transition im PropertyPanel des Workspaces (unten links)

3.7.8.2.6 Beabeiten von Transitionen Transitionen werden über das PropertyPanel des Workspaces bearbeitet. Abbildung 3.57 zeigt einen solchen Vorgang. Das angezeigt Panel ist abhängig vom Automaten- und Transitionstyp. Das in der genannten Abbildung gezeigte Panel ist jedoch die Standardversion zum Bearbeiten von Transitionen mit Symbol-Intervallen. Über die Checkbox „epsilon“ kann eingestellt werden, ob die markierte Transition auch einen Epsilon-Übergang haben soll. In der Tabelle darunter können Intervalle von Symbolen erstellt und bearbeitet werden. Das Erstellen und Entfernen geschieht über die beiden Schaltflächen „Add“ und „Remove“. Das Start- und das Endsymbol eines Intervalles kann direkt in der Tabelle bearbeitet werden. Dazu muss einfach die entsprechende Tabellenzelle angeklickt werden. Anschließend kann man über die Tastatur Veränderungen durchführen, die durch Drücken der ENTER-Taste umgesetzt werden. Die entsprechende Darstellung im Editor wird dabei sofort aktualisiert.

3.7.9 Plugins

Der Triple-A Workspace erhält seine Funktionalität durch installierte Plugins. Ein gültiges Plugin befindet sich in einem Unterverzeichnis des Pluginverzeichnisses. Folgende Plugins sind bereits vorinstalliert.

3.7.9.1 Finite Automaton Plugin

Dieses Plugin ist ein einfaches Plugin für endliche Automaten. Mit diesem können endliche Automaten mit einem Character Alphabet gezeichnet werden. Des weiteren ermöglicht es alle gängigen Automatenoperationen. Zur Basis der Analysetools gehören zwei Funktionen, die einen Lauf im Automaten ermöglichen, sowie eine Funktion, die akzeptierte Wörter für jeden akzeptierenden Endzustand anzeigt.

3.7.9.1.1 Starten des Plugins Über den Menüeintrag *File - New - Finite automaton via visual editing - CharacterSymbol-Alphabet* kann eine neue Automatenanalyse gestartet werden. Man erhält ein neues Automatenfenster, in dem ein neuer endlicher Automat gezeichnet werden kann.

3.7.9.1.2 Operationen auf Automaten Das Plugin erlaubt die gängigen Automatenoperationen auf endlichen Automaten. Hierbei unterscheidet man Operationen auf genau einem oder auf mehreren Automaten. Auf die einzelnen Operationen wollen wir hier im Einzelnen nicht eingehen.

3.7.9.1.3 Analysen

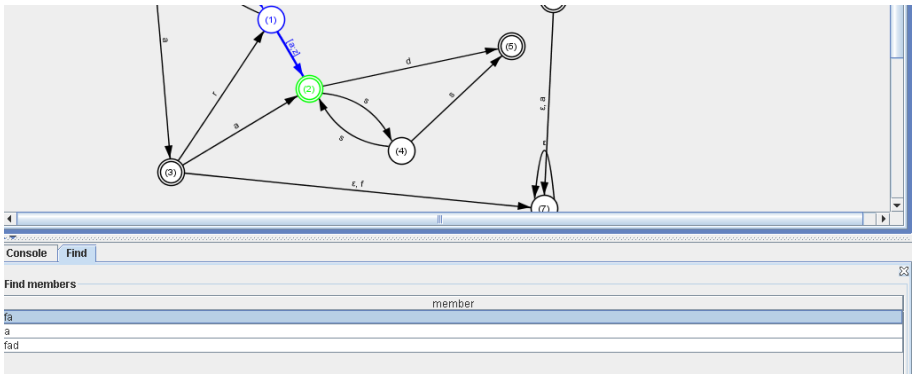


Abbildung 3.58: Auffinden einer akzeptierenden Belegung

Findmembers: Diese Funktion startet man, indem man einen Automaten markiert und diese Funktion im Analysemenü aufruft. Das Ergebnis ist eine Liste mit je einem akzeptierten Wort pro akzeptierendem Endzustand. Klickt man einen dieser Einträge an, so wird dieser farblich hervorgehoben, sofern der Automat nicht verborgen ist.

Run: Bei dieser Funktion kann ein Lauf im Automaten simuliert werden. Auch hierzu muss ein Automat markiert werden und im Analysemenü die Aktion *run* gestartet werden. Im Analysepanel erscheint das entsprechende Tab, siehe Abbildung 3.59. In

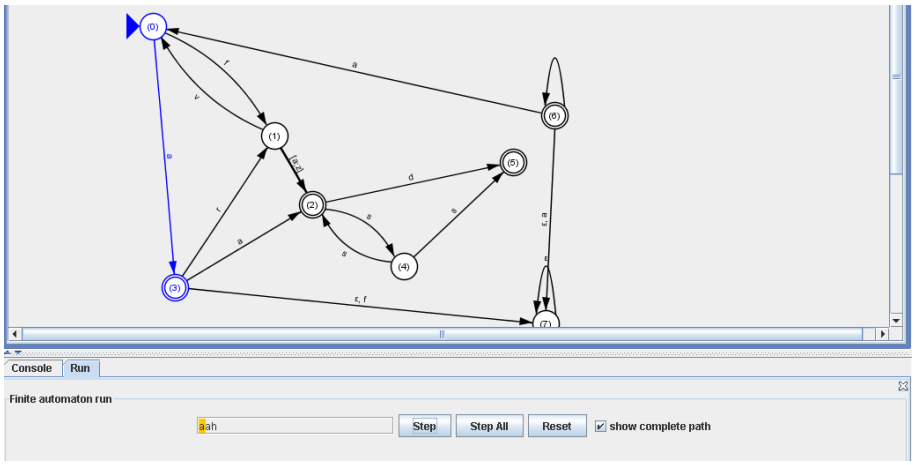


Abbildung 3.59: Automatenlauf

diesem Run-Tab gibt man eine zu testende Eingabe in das Eingabefeld ein. Mit der Schaltfläche *Step* kann man die Eingabe Schritt für Schritt durchtesten. Die Schaltfläche *StepAll* versucht das ganze Wort auf einmal zu durchlaufen. Im Eingabefeld wird das bereits gelesene Wort farblich markiert. Ist eine Eingabe akzeptierend, so wird der zuletzt markierte Zustand im Automaten und das Eingabefeld grün, ansonsten rot gefärbt dargestellt. Wird ein Zeichen gelesen, das im Automaten nicht vorhanden ist, so stoppt der Automat an dieser Stelle. Das Wort ist dann nicht akzeptierend. Es gibt noch zwei weitere Funktionen. Über die Schaltfläche *Reset* wird die Analyse zurückgesetzt. Über die Option *show complete path* kann man bestimmen, ob der ganze Weg des Laufes oder nur der aktuell erreichte Zustand im Automaten angezeigt werden soll.

3.7.9.2 Presburger Plugin

Das Presburger Plugin ist eine Erweiterung des Finite Automaton Plugins. Aus diesem Grund besitzt es dieselben Funktionen wie dieses. Der einzige Unterschied besteht darin, dass ein Automat auf Basis einer Transformation entsteht und deshalb nicht mehr bearbeitet werden darf.

Eine Formel der Presburger-Logik kann in einen endlichen Automaten transformiert werden, und die Erfüllbarkeit dieser Formel kann mit der Analysefunktion *findMembers* entschieden werden.

3.7.9.2.1 Eingabe Als Eingabe wird eine Formel der Presburger Arithmetik erwartet.

3.7.9.2.1.1 Atome Im einfachsten Fall ist die Formel ein Atom, wobei ein Atom die Verknüpfung zweier Terme durch $=$, $<$, $\leq$, $>$ oder $\geq$ ist und ein Term entweder ein einzelnes Glied oder die Verknüpfung mehrerer Glieder durch binäre $+$- oder $-$-Operatoren ist. Ein Glied wiederum ist entweder die dezimale Repräsentation g einer ganzen Zahl, eine einzelne Variable x oder aber das Produkt $g * x$ einer ganzen Dezimalzahl g und einer Variablen x. Als Variablen sind alle Zeichenfolgen erlaubt, die mit einem Kleinbuchstaben beginnen und nur aus Kleinbuchstaben oder Dezimalziffern bestehen. Alle Zahlen müssen im Intervall $\{-2147483648, \ldots, 2147483647\}$ liegen. Innerhalb von Termen und Atomen dürfen sich Variablen beliebig oft wiederholen und sich auch gegenseitig aufheben. Es ist nur darauf zu achten, dass sich das Atom nicht zu $true$ oder $false$ reduzieren lässt. An keiner Stelle muss unbedingt ein Leerzeichen stehen, aber es dürfen zwischen den Elementen beliebig viele Leerzeichen eingefügt werden. Ein Atom darf in eine beliebige Anzahl von Klammerpaaren eingebettet werden. Alle Variablen werden als ganzzahlig interpretiert. Ein Beispiel für ein Atom ist $3 * x1 + 7 * x2 - 8 * x3 + z + 4 = y + z + 8$.

3.7.9.2.1.2 Boolesche Verknüpfungen Sind F_1 und F_2 Presburger-Formeln, so ist $\neg(F_1)$ eine Presburger-Formel. $(F_1 \wedge F_2)$ und $(F_1 \vee F_2)$ sind ebenfalls Presburger-Formeln, falls keine Variable in F_1 quantifiziert und in F_2 frei auftritt oder umgekehrt. Die Klammern dürfen nicht weggelassen werden, aber es können beliebig viele zusätzliche Klammerungen vorgenommen werden. Auch hier sind Leerzeichen lediglich optional. Bei der Negation ist zu beachten, dass auch dann ein zusätzliches Klammerpaar hinzugefügt werden muss, wenn F_1 bereits geklammert ist, aber das äußere Klammerpaar von F_1 nicht überflüssig ist. Alternativ zu $\wedge$, $\vee$ und $\neg$ können auch $/and$, $/or$ und $/not$ verwendet werden. Ein Beispiel für eine korrekte Formel ist $\neg((x - y \leq 2 \wedge x \leq 5))$.

3.7.9.2.1.3 Quantoren Für eine Presburger-Formel F sind $\exists x : (F)$ sowie $\forall x : (F)$ ebenfalls Presburger-Formeln, wenn F nicht bereits einen die Variable x bindenden Quantor enthält. Leerzeichen sind optional, zusätzliche äußere Klammern möglich, und wie bei $\neg$ kann sich die Notwendigkeit doppelter Klammerpaare ergeben. x repräsentiert eine beliebige Variable, die nicht unbedingt in F vorkommen muss. Alternativ zu $\exists$ und $\forall$ können auch $/ex$ und $/all$ benutzt werden.

3.7.9.2.2 Transformation Eine Tranformation wird gestartet, indem man auf die Schaltfläche Create Automaton drückt. Bei erfolgreicher Transformation erscheint ein neues Tab im Fenster mit der Überschrift Automaton.

3.7.9.2.3 Operationen und Analysen Alle Funktionen verlaufen analog zum Finite Automaton Plugin. [htpb] Die Analysefunktionen weichen optisch leicht ab, weil ein

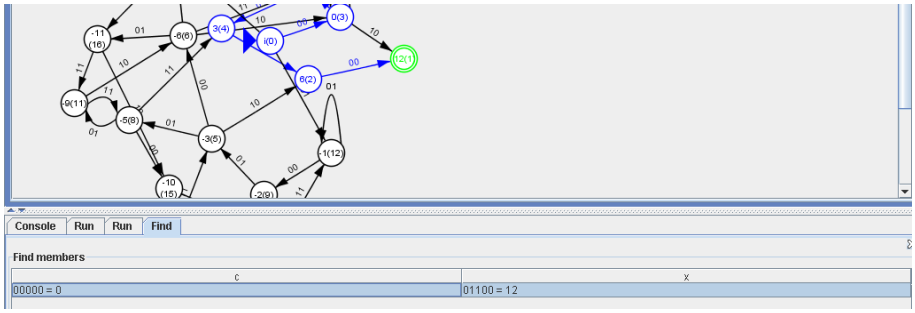


Abbildung 3.60: Auffinden einer aktzepierenden Belegung bei Bitvektoren

Presburger Plugin Automaten mit Bitvektoren verwendet. So werden nun in beiden Analysefunktionen Tabellen verwendet, die die jeweiligen Variablenbelegungen anzeigen, siehe 3.61 und 3.60

3.7.9.3 Regular Expression Plugin

Auch das Plugin für Regular Expressions ist eine Erweiterung des Finite Automaton Plugin. Mit diesem Plugin lässt sich ein regulärer Ausdruck in einen Automaten transformieren.

3.7.9.3.1 Eingabe Als Eingabe wird ein regulärer Ausdruck erwartet.

Folgende Operationen von regulären Ausdrücken werden unterstützt:

- Vereinigung von S und U: $S \mid U = \{s \mid s$ ist in S oder s ist in $U\}$.
- Konkatenation von S und U: $SU = \{st \mid s$ ist in S und u ist in $U\}$.
- Kleene-Abschluss von S: $S\star =$"null oder mehr Konkatenationen von S".

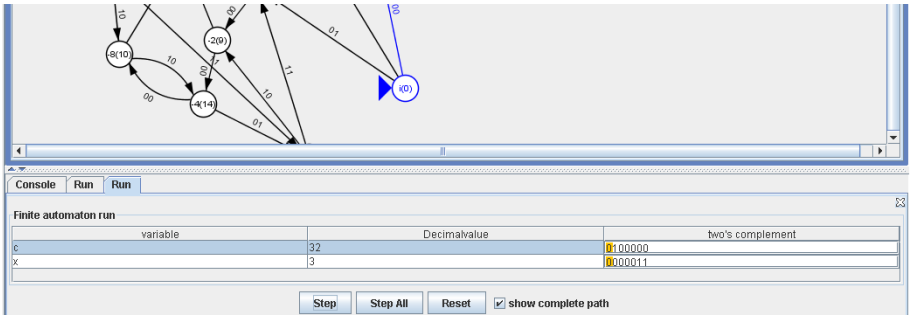


Abbildung 3.61: Automatenlauf bei Bitvektoren

- Positiver Abschluss von S: $S+$ = “ein oder mehr Konkatenationen von S”.
- Epsilon Abschluss von S: $S?$ = “null oder ein S”.

Beispiel : $(a \mid b) \star abb$

3.7.9.3.2 Transformation Eine Tranformation wird gestartet, indem man auf die Schaltfläche **Create Automaton** drückt. Bei erfolgreicher Transformation erscheint ein neues Tab im Fenster mit der Überschrift **Automaton**.

3.7.9.3.3 Operationen und Analysen Alle Funktionen verlaufen analog zum Finite Automaton Plugin.

3.7.10 LTL Plugin

Dieses Plugin setzt auf dem Plugin der Büchiautomaten auf und erzeugt einen solchen anhand einer eingegebenen LTL-Formel (Linear Temporal Logic).

3.7.10.1 Eingabe

Als Eingabe werden beliebige LTL-Formeln der akzeptiert.

3.7.10.1.1 Klammerung Formeln und Teilformeln dürfen beliebig oft geklammert werden, obwohl keine explizite Klammerung erzwungen wird. Von Klammerungsmöglichkeiten sollte aber großzügig Gebrauch gemacht werden, da es zwischen den binären Operatoren keine Operatorpräzedenzen gibt. Jedoch binden unäre Operatoren grundsätzlich stärker als binäre. Unter binären Operatoren gilt eine implizite Rechtsklammerung.

3.7.10.1.2 Leerzeichen Zwischen binären Operatoren und den Operanden muss jeweils ein Leerzeichen stehen, wenn die Operanden keine äußere Klammer besitzen. Ansonsten sind Leerzeichen optional. Es dürfen beliebig viele optionale Leerzeichen zwischen Operatoren, Operanden und Klammern eingefügt werden.

3.7.10.1.3 Atomare Propositionen Jede atomare Proposition ist eine LTL-Formel. Eine atomare Proposition ist dabei eine aus nicht für Operatoren reservierten Zeichen

bestehende Zeichenkette. Beispiele für atomare Propositionen sind p, aBc, $n = 3$, $3 = n$ und 123.

3.7.10.1.4 Boolesche Konstanten Die Booleschen Konstanten $\top$ und $\bot$ bzw. alternativ $true$ und $false$ sind LTL-Formeln.

3.7.10.1.5 Boolesche Operatoren Sind F_1 und F_2 LTL-Formeln, so sind auch $F_1 \wedge F_2$, $F_1 \vee F_2$, $\neg F_1$, $F_1 \Rightarrow F_2$ und $F_1 \Leftrightarrow F_2$ LTL-Formeln. Alternativ kann auch `&&`, `|||`, `!`, `->` und `<->` geschrieben werden.

3.7.10.1.6 Temporallogische Operatoren Für LTL-Formeln F_1 und F_2 sind XF_1, FF_1, GF_1, F_1UF_2 und F_1RF_2 ebenfalls LTL-Formeln. Die Semantik jedes dieser Operatoren soll im Folgenden kurz beschrieben werden, da die temporallogischen Operatoren für gewöhnlich weniger häufig in der Welt angetroffen werden als die anderen Operatoren. Man stelle sich vor, dass man sich in einem bestimmten Zustand q eines Kripkemodells befindet.

3.7.10.1.7 Next-Operator X Die Formel XF_1 gilt in q genau dann, wenn auf allen in q startenden Pfaden der jeweils zweite Zustand des Pfades die Formel F_1 erfüllt.

3.7.10.1.8 Finally-Operator F FF_1 sagt aus, dass auf allen in q startenden Pfaden irgendwann einmal F_1 gilt.

3.7.10.1.9 Globally-Operator G GF_1 bedeutet, dass F_1 auf allen in q startenden Pfaden gilt, also in jedem Zustand jedes solchen Pfades.

3.7.10.1.10 Until-Operator U Eine Formel F_1UF_2 sagt aus, dass auf allen in q startenden Pfaden F_1 gilt, bis F_2 gilt. Für die ersten Zustände eines jeden Pfades gilt also F_1. Der erste Zustand, der F_1 nicht mehr erfüllt, muss F_2 erfüllen. Es ist nicht spezifiziert, was anschließend passiert. Es ist erlaubt, dass F_1 auf einem Pfad immer gilt, aber F_2 muss auf jedem Pfad irgendwann einmal erfüllt sein. Die Formel ist in q auch dann erfüllt, wenn F_2 bereits in q wahr ist. Die Erfülltheit von F_1 ist in diesem Fall unerheblich.

3.7.10.1.11 Release-Operator R Eine Formel F_1RF_2 ist genau dann erfüllt, wenn auf allen in q startenden Pfaden F_2 stets erfüllt ist bis zu dem Punkt an dem sowohl

F_1 als auch F_2 gelten. Was danach geschieht, ist nicht spezifiziert. Der Zeitpunkt der gemeinsamen Erfülltheit von F_1 und F_2 muss nicht eintreten.

3.7.10.2 Operationen

Es stehen die für Büchiautomaten üblichen Operationen im „Action“-Menü zur Verfügung. Dies sind für einzelne Automaten insbesondere das Generalisieren und Degeneralisieren und bei mehreren ausgewählten Automaten das Schneiden und Vereinigen.

3.7.11 Kripke Plugin

3.7.11.1 Eingabe

Die Eingabe von Kripkemodellen erfolgt im Gegensatz zu den meisten anderen Plugins nicht über die Eingabe einer Formel, sondern durch direktes Zeichnen im Editor.
Direkt nach Starten des Plugins erscheint ein Dialog zur Eingabe der vom Kripkemodell abgedeckten Propositionen. Über das Textfeld und die beiden Aktionen „Add“ und „Delete selected“ kann die Liste der Propositionen bearbeitet werden. Nach Abschluss dieser Eingabe öffnet sich ein Editor-Fenster zum Erstellen der Kripkemodells. Die Bedienung erfolgt wie im Abschnitt 3.7.8 beschrieben. Nachdem man einen Zustand erzeugt hat, kann man im Propertypanel des Workspace dessen Propositionen bearbeiten. Über das Kontextmenü der rechten Maustaste kann man einen Zustand des Kripkemodells zum Startzustand machen. Für Transitionen gibt es, wie bei Kripkemodellen üblich, keine weiteren Editiermöglichkeiten außer dem Erstellen und dem Entfernen.

3.7.11.2 Analysen

3.7.11.2.1 convert to Büchi automaton Diese Aktion erzeugt ein neues Automaten-Fenster, welches das in einen Büchiautomaten konvertierte Kripkemodell enthält. Mit dem Büchiautomaten können anschließend weitere Operationen, wie im entsprechenden Abschnitt dargestellt, durchgeführt werden.

3.7.11.2.2 add LTL formula tab Dieser Befehle fügt dem Kripke-Fenster ein neues Tab hinzu, über das eine LTL-Formel eingegeben und anschließend ein Model Checking des Kripkemodells anhand dieser Formel durchgeführt werden kann. Nach Betätigen der entsprechenden Aktion „Run model check“ wird über eine Dialog-Box angezeigt, ob das Kripkemodell der durch die LTL-Formel dargestellten Spezifikation entspricht, oder nicht.

3.7.12 While Plugin

Das While-Plugin ist eine Erweiterung des Kripke-Plugins und wird zur Erstellung von Kripkemodellen aus Programmen der While-Sprache verwendet. Damit ist es möglich, ein Model Checking von in While geschriebenen Algorithmen durchzuführen und diese somit auf Korrektheit zu untersuchen.

3.7.12.1 Eingabe

Die Eingabe erfolgt in Form eines While-Programmes in das Source-Tab des neu geöffneten Fensters. Eine genau Erläuterung der Eingabe befindet sich in 3.1.5.1.4 unter While Semantik.

3.7.12.2 Analysen

Die angebotenen Optionen im Analysis-Menü, das Konvertieren in einen Büchiautomaten und das Hinzufügen einer LTL-Formel, sind die gleichen wie im vorhergehenden Abschnitt zum *Kripke Plugin* erläutert.

3.7.12.3 Omega Reguläres Plugin

Diese Plugin transformiert eine als ω-regulärer Ausdruck dargestellte Sprache (auch als ω-reguläre Sprache bezeichnet) in einen Büchiautomaten.

3.7.12.3.1 Eingabe Als Eingabe wird ein $S\omega$-regulärer Ausdruck erwartet.

Es werden alle Operationen von regulären Ausdrücken (siehe 3.7.9.3.1) unterstützt. Hinzu kommt folgende Operation:

- $S\omega$-Abschluss von S: $S\omega$ = "unendlich viele Konkatenationen von S".

Beispiel : $abb(a \mid b)\omega$

3.7.12.3.2 Transformation Eine Tranformation wird gestartet, indem man auf die Schaltfläche Create Automaton drückt. Bei erfolgreicher Transformation erscheint ein neues Tab im Fenster mit der Überschrift Automaton.

Kapitel 4

Epilog

Am Ende der Projektarbeit kommt der Zeitpunkt, an dem man zurückblickt, die geleistete Arbeit betrachtet und nach dem Reflektieren begutachtet. Sicher ist es nicht möglich alle Wünsche und Vorstellungen jedes Teilnehmers in der knapp bemessenen Zeit von nur zwei Semestern umzusetzen. Auch und gerade bei dem Thema der automatischen Analyse mit Automaten konnten nicht alle Prinzipien in dem Umfang gelöst werden, wie es mit den in der Seminarphase ausgearbeiteten Themen geschehen ist.

4.1 Ergebnisse des ersten Semesters

Am Anfang der Projektgruppe war noch nicht klar, in welche Richtung die Umsetzung des abstrakten Themas der „automatischen Analyse mit Automaten“ gehen würde. Als Minimalziel war lediglich festgelegt, dass exemplarisch die Analyse der Presburger Arithmetik auf Grundlage von endlichen Automaten in irgendeiner Form gelöst werden sollte. Das erste Semester sollte der Orientierung dienen. Es wurde das Gerüst einer eigenen Automatenbibliothek begonnen, das im Hinblick auf weiterführende Ziele mehr bieten sollte als nur endliche Automaten, denn für diese Automatenklasse existierten bereits Lösungen. Die Projektstruktur nahm eine Vierteilung an: Die Automatenbibliothek als Kern, die sich darauf stützenden Analyse, einen dafür benötigten Parser und eine graphische Oberfläche für die Ein- und Ausgabe.

Die Analyse der Presburger Arithmetik wurde auf der Grundlage der Interface-Spezifikation der Automatenbibliothek und des Parsers begonnen und durch Rücksprache ggf. angepasst. Als Ergebnis sollte sich herausstellen, dass dieses Element nach Abschluss der anderen beiden reibungslos funktionierte.

Für den Parser war ursprünglich vorgesehen, dass zunächst ein einfacher statischer Parser im späteren Projektverlauf durch einen tabellengesteuerten ersetzt werden sollte. Damit sollten alle möglichen Analysevorhaben unterstützt werden, wobei zu diesem Zeitpunkt noch nicht klar war, welche dies sein werden. Dieses Projekt wurde gegen Ende des ersten Semesters eingestellt, da es zu aufwändig geworden wäre und eine derart flexible Lösung auch nicht mehr benötigt wurde, so dass der Nutzen die Kosten nicht mehr rechtfertigte.

Die entstandene Automatenbibliothek war eine sehr schnelle Implementation, deren Erweiterbarkeit aber noch auf die Probe gestellt werden sollte, da bisher nur ein Automatenmodell, nämlich die endlichen Automaten, benötigt und implementiert wurde.

Als Ergebnis für den geplanten Workspace kam ein sehr ansehliches Produkt zu Tage. Insbesondere der automatische Layouter, der einen sehr komfortablen Editor begleitete, war ein besonderes Ergebnis dieser Projektgruppe. Der Workspace war bereits sehr modular gestaltet, bot allerdings ebenfalls nur das eine bisher umgesetzte Analyseverfahren. Einige kleinere, grundlegende Operationen standen noch aus, wurden aber gleich zu Beginn des zweiten Semesters umgesetzt.

4.2 Ergebnisse des zweiten Semesters

Nachdem das Minimalziel bereits schon im ersten Semester erreicht war, war ein neuer Kurs zu Beginn des zweiten Semesters einzuschlagen. Im besonderen Maße ist die Alternative zwischen Baumautomaten und Büchiautomaten abgewogen worden. Man hat sich für Büchiautomaten und das damit verbundene Model Checking mit LTL-Formeln entschieden. Dieses Automatenmodell ist dem der endlichen Automaten noch sehr nahe und somit sollte mehr Wert auf die Analyse gelegt werden und die Abrundung aller Feinheiten. Das ist auch ein Grund, warum man sich entschlossen hatte, den Schritt zu gehen, dass die interne Darstellung von transitionsauslösenden Symbolmengen nicht mehr nur über Intervalle von Symbolen, sondern auch andere denkbare Datenstrukturen gelöst werden sollte.

Es wurde beschlossen, dass sich als Datenstruktur dafür OBDDs gut eignen, und dies wurde zu einem neuen Arbeitsteilgebiet. Die Automatenbibliothek musste u. A. daraufhin erweitert werden und somit wurde der Kern der Automatenbibliothek als getrennter Arbeitsbereich zu den konkreten Operationen bzw. Algorithmen für Büchiautomaten eingestuft. Zusätzlich musste noch eine simple Datenstruktur für die zu prüfenden Modelle bereitgestellt werden.

Um für das LTL-Model Checking einfach an sinnvolle Modelle zu kommen und Analysen somit automatischer zu gestalten, sollten aus simplen Programmen geeignete Modelle generiert werden. Diese Anforderung fand ihre Umsetzung in einem Konverter, welcher While-Programme in Kripkemodelle und diese schließlich in Büchiautomaten umwandelt.

Für die GUI wurde ebenfalls der Kern restrukturiert, denn wie bei der Automatenbibliothek wurden auch hier einige Verallgemeinerungen vorgenommen. Diese zeigten dann auch gleich ihre gute Seite, als es daran ging, die neu entwickelten Datenstrukturen zu visualisieren. Das graphische Framework konnte praktisch nur durch Erweiterungen und ohne Veränderung des Bestehenden auf die neuen darzustellenden Strukturen wie OBBDs und Kripkemodelle erweitert werden.
Der Workspace und der graphische Editor verschmolzen zu einem Gesamtprodukt, das durch zahlreiche Verbesserungen den im ersten Semester gewünschten und den im zweiten Semester neu notwendig gewordenen Anforderungen größtenteils gerecht wurde.

Das LTL-Model Checking arbeitete am Ende mit allen Komponenten, wie schon im ersten Semester die Presburger Arithmetik, problemlos zusammen. Alle Ergebnisse flossen zusammen und konnten mit Hilfe des Workspace genutzt werden.

4.3 Ausblick

Das Konzept des Projektes war es, einen flexiblen, universell einsetzbaren, aber auch erweiterbaren Baukasten für Automaten zu schaffen, auf dessen Grundlage Analysekonzepte umgesetzt werden können. Mit dem Abschluss dieser Projektgruppe liegt nun dieser Baukasten vor. Die Welt der automatischen Analyse mit Automaten ist aber noch nicht ausgeschöpft und bietet genug Material, um sich auch weiterhin damit zu beschäftigen.

Zunächst stellt man fest, dass von den theoretischen Automatenmodellen lediglich zwei konkrete umgesetzt wurden. Es gibt aber noch mindestens folgende Automatenmodelle, die bereits aus der Planungsphase des Projektes bekannt waren:

- Pushdown Automaten
- Bottom-up Baumautomaten
- Top-down Baumautomaten
- Alternierende (Büchi) Automaten
- Alternierende (Büchi) Baumautomaten

Die Umsetzung dieser Automatenmodelle würde gleichzeitig neue Analyseverfahren zugänglich machen, die auf diesen Modellen beruhen. Als Beispiel sei das CTL-Model Checking genannt, was eine natürliche Erweiterung zu dem LTL-Model Checking ist.

Gleichwohl sei festgestellt, dass der Workspace zur Visualisierung und Umgang mit Automaten und Analysen ebenfalls modular aufgebaut ist, um das modulare Konzept der Automatenbibliothek und deren Analysemethoden angemessen begleiten zu können. Auf Grundlage dieser Elemente ist es vorstellbar, dass sich die hier beispielhaft genannten Modelle (und evtl. auch noch nicht beachtete Modelle) auf natürliche Weise umsetzen lassen. Auch abseits des Themas „Automaten“ sind im Bereich der Visualisierung noch Erweiterungen denkbar. So wäre es möglich einen weiteren Layouter, wie zum Beispiel den sehr ausgeklügelten Layouter des bekannten Graphviz-Paketes, zu programmieren.

4.4 Schlusswort

Sind alle Ziele erreicht worden? *„Ja“* wird man sagen, wenn man die Mächtigkeit dieser Ausarbeitung betrachtet und die Ergebnisse des ersten und zweitens Semesters gelesen hat. Doch neben diesen Zielen hat die „Lehrveranstaltung Projektgruppe“ auch weitere. Auf den Webseiten des Projektgruppenbeauftragten heisst es: *„Die Teilnehmer sollen Erfahrungen in Teamarbeit und Aufgabenorganisation im Team erwerben. Sie sollen Ziele selbst definieren und ihre Durchsetzung selbständig verfolgen lernen.“*
Dies taten die Teilnehmer dann auch. Die Anfangsphase der Projektgruppe war geprägt von der Suche nach einem sinnvollen Einsatz der in der Seminarphase erlangten Kenntnisse. Es folgte eine Aufteilung in Gruppen, welche rasch ihre Arbeit aufnahmen und schon früh Ergebnisse präsentieren konnten. All dies geschah in ständiger Kommunikation miteinander, sei es in den wöchentlichen PG-Sitzungen, gruppeninternen Sitzungen, aber auch über fast 500 Forums-Einträgen und über 1000 Wiki-Seiten-Editierungen (eine nun bewährte Kombination, welche zukünftigen Projektgruppen nur empfohlen werden kann!).
Auch andere Erfahrungen wurden gemacht. Dazu gehörte die Feststellung, dass wann immer eine Gruppe solcher Größe zusammenkommt, Inhomogentitäten zwischen den Kenntnissen und Fähigkeiten, Wünschen und Vorstellungen eines jeden Einzelnen auftreten. Aber auch solche Momente wurden ausdiskutiert, gelöst und trugen somit auch zum Erfahrungsgewinn der Teilnehmer bei.

Literaturverzeichnis

[1] *Mona-Manual*. `http://www.brics.dk/mona/manual.html`.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers. principles, techniques, and tools*. Addison-Wesley Publishing Company, 1986.

[3] Jürgen Bill. *Ausarbeitung zum Vortrag über Model Checking*. 2005.

[4] Beate Bollig. `http://ls2-www.cs.uni-dortmund.de/~bollig/tdl-dortmund.html`.

[5] Ahmed Bouajjani, Javier Esparza, and Oded Maler. Reachability analysis of pushdown automata: Application to model-checking. In Antoni W. Mazurkiewicz and Józef Winkowski, editors, *CONCUR*, volume 1243 of *Lecture Notes in Computer Science*, pages 135–150. Springer, 1997.

[6] BSD. `http://www.opensource.org/licenses/bsd-license.php`.

[7] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[8] Hubert Comon, Max Dauchet, Rémi Gilleron, Florent Jacquemard, Denis Lugiez, Sophie Tison, and Marc Tommasi. *Tree Automata Techniques and Applications*. 2005. `http://www.grappa.univ-lille3.fr/tata/`.

[9] Jim des Rivières. *Eclipse project briefing materials*. IBM Corporation and others, Mar 2003. `http://www.eclipse.org/eclipse/presentation/eclipse-slides.ppt`.

[10] dk.brics. `http://www.brics.dk/automaton/`.

[11] Eclipse Foundation. *Graphical Editing Framework*. `http://www.eclipse.org/gef/`.

[12] Wikipedia: Parsen einer Grammatik. `http://en.wikipedia.org/wiki/Parsing_expression_grammar`.

[13] Javier Esparza and Jens Knoop. An automata-theoretic approach to interprocedural data-flow analysis. In Wolfgang Thomas, editor, *FoSSaCS*, volume 1578 of *Lecture Notes in Computer Science*, pages 14–30. Springer, 1999.

[14] FMS-Folien. `http://ls5-www.cs.uni-dortmund.de/imperia/md/content/fms05/fms05_folien1.pdf`.

[15] Thomas M. J. Fruchterman and Edward M. Reingold. *Graph Drawing by Force-directed Placement. Softw., Pract. Exper.*, 21(11):1129–1164, 1991.

[16] Erich Gamma and Kent Beck. *Contributing to Eclipse*. Addison Wesley, 2003.

[17] Rob Gerth, Doron Peled, Moshe Y. Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *PSTV*, pages 3–18, 1995.

[18] GPL. `http://www.gnu.org/copyleft/gpl.html`.

[19] GraphViz. `http://www.research.att.com/sw/tools/graphviz/`.

[20] Grappa. `http://www.research.att.com/%7Ejohn/Grappa/grappa.html`.

[21] jABC. `http://jabc.cs.uni-dortmund.de/`.

[22] JFlap. `http://www.jflap.org/`.

[23] JRexx. `http://www.karneim.com/jrexx/`.

[24] JUNG. `http://jung.sourceforge.net/`.

[25] P. Kelb, T. Margaria, M. Mendler, and C. Gsottberger. Mosel: A flexible toolset for monadic second-order logic, March 1997.

[26] Aufsteigende Kettenbedingung. `http://en.wikipedia.org/wiki/Ascending_chain_condition`.

[27] Kupferman and Vardi. Weak alternating automata are not that weak. In *ISTCS: 5th Israeli Symposium on the Theory of Computing and Systems*, 1997.

[28] Orna Kupferman and Moshe Y. Vardi. Weak alternating automata and tree automata emptiness. In *STOC*, pages 224–233, 1998. `http://doi.acm.org/10.1145/276698.276748`.

[29] LGPL. `http://www.gnu.org/licenses/licenses.html#LGPL`.

[30] Bill Moore, David Dean, Anna Gerber, Gunnar Wagenknecht, and Philippe Vanderheyden. *Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework*. IBM Corporation, Feb 2004.

[31] F. Nielson, H. R. Nielson, and C. L. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[32] org.eclipse.draw2d.graph. `http://help.eclipse.org/help31/topic/org.eclipse.draw2d.doc.isv/reference/api/overview-summary.html`.

[33] J.P. Pécuchet. On the complementation of Büchi automata. In *Theoretical Computer Science 47*, pages 95–98. 1986.

[34] Spin-Homepage. `http://spinroot.com/spin/whatispin.html`.

[35] Bernhard Steffen and Tiziana Margaria. Metaframe in practice: Design of intelligent network services. In *Correct System Design*, pages 390–415, 1999.

[36] Das Java Automata Toolkit. `http://humboldt.sunyit.edu/JCT/JAT.htm`.

[37] Wikipedia: Verfahren unter Packrat. `http://en.wikipedia.org/wiki/Packrat_parser`.

[38] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *lics86*, pages 332–344, 1986.

[39] Moshe Y. Vardi. Alternating automata and program verification. In *Computer Science Today*, pages 471–485. 1995.

[40] Moshe Y. Vardi. Alternating automata: Unifying truth and validity checking for temporal logics. In *CADE*, pages 191–206, 1997.

[41] Vollständiger Verband. `http://de.wikipedia.org/wiki/Verband_%28Mathematik%29`.

[42] Wikipedia. `http://de.wikipedia.org/wiki/Omega-Automat`.

[43] Wikipedia. `http://de.wikipedia.org/wiki/Julius_Richard_B%fcchi`.

[44] Wikipedia. `http://de.wikipedia.org/wiki/B%fcchi-Automat`.

[45] Wikipedia. `http://www.informatik.hu-berlin.de/~kschmidt/Modelchecking/Node25.Html`.

[46] Pierre Wolper and Bernard Boigelot. On the construction of automata from linear arithmetic constraints. In Susanne Graf and Michael I. Schwartzbach, editors, *TACAS*, volume 1785 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2000.