FSMCalc

Leopold C. R. Haller

5. August 2006

Ziel dieser Arbeit war die Entwicklung einer Programmbibliothek für die Arbeit mit *finiten Automaten* und *Labelled Transition Systems*. Weiters wurde ein Kommandozeileninterface implementiert, dass einfachen Zugriff auf die Hauptfunktionalitäten der Bibliothek erlaubt.

Die implementierten Algorithmen sind auf die Anforderung des *Model Checkings* mit finiten Automaten zugeschnitten. Neben einigen grundlegenden Funktionen, die zur Manipulation, Konstruktion und Analyse finiter Automaten verwendet werden können, wurde ein Algorithmus zur Verifikation paralleler Systeme implementiert, der auf eine Variante der *Partial Order Reduction* zurückgreift. Dadurch lassen sich große Systeme mit mehreren asynchronen, nebenläufigen Komponenten analysieren, die ohne Reduktion des Suchraumes nicht mehr handhabbar wären.

Für die Arbeit mit dem Kommandozeileninterface wurde eine Eingabesprache für finite Automaten entwickelt, die sich am Formalismus der *Prozess-Algebra* orientiert. Des weiteren wurde mit Hilfe der externen Graphikbibliothek *JGraph* eine einfache Visualisierungsklasse entworfen.

Inhaltsverzeichnis

1 Einleitung **4**

2 Finite Automaten, Labelled Transition Systems und Prozessalgebra **6**

2.1 Prozessalgebra zur Darstellung finiter Automaten 6

3 Algorithmen zur Transformation, Konstruktion und Analyse finiter Automaten **9**

3.1 accepts() 9

3.2 product() 9

3.3 power() 11

3.4 complementLanguage() 12

3.5 parallel() 13

3.6 minimize() 16

3.7 link() 16

3.8 partialOrderReduction() 17

3.8.1 Auswahl von repräsentativen Pfaden 18

3.8.2 Berechnung von *ample(s)* 21

3.8.3 Implementierung 22

4 Aufbau der Implementierung **27**

4.1 Wahl der Programmiersprache 27

4.2 Systemarchitektur 28

4.3 Datenstruktur 29

4.4 Parsing der Eingabesprache 29

4.5 Visualisierung 30

5 Konklusion **31**

A EBNF-Grammatik der Eingabesprache **32**

1 Einleitung

Mit der wachsenden Komplexität von Soft- und Hardware und der gleichzeitig zunehmenden Integration in allen Bereichen unseres Lebens, steigen auch die Sicherheitsanforderungen, die an Softwaresysteme gestellt werden.

Gerade in der Softwarebranche gilt es als „fact of life“, dass kein größeres System ohne Fehler ausgeliefert wird. Gleichzeitig werden immer mehr sicherheitskritische Systeme entworfen, in denen die Möglichkeit eines Softwarefehlers begrenzt oder gar nicht tolerabel ist. Klassische qualitätssichernde Maßnahmen, wie der Einsatz von speziellen Softwareengineeringtechniken oder rigorosen Tests, reichen nicht immer aus, um die Auftretenswahrscheinlichkeit eines Fehlers auf ein Maß zu reduzieren, dass in kritischen Applikationen akzeptabel ist.

Die formale Verifikation von Software stellt eine Möglichkeit dar, diesem Problem zu begegnen. Dazu muss ein formales Modell des betrachteten Systems erstellt werden, das mit Verifikationsalgorithmen hinsichtlich seiner Eigenschaften analysiert werden kann. *Finite Automaten* und *Labelled Transition Systems* sind zwei Möglichkeiten, ein Soft- oder Hardwaresystem in ein solches formales Modell zu fassen.

In der vorliegenden Arbeit wird eine Softwarebibliothek präsentiert, die die Arbeit mit diesen beiden Datenstrukturen erlaubt. Zusätzlich wurde eine Benutzerschnittstelle entworfen, die einfachen Zugriff auf die Funktionen des Frameworks bietet. Es wurden Algorithmen implementiert, die die Konstruktion, Manipulation und Analyse von *finiten Automaten* und *Labelled Transition Systems* ermöglichen. Bei der Auswahl wurde dabei Wert auf die Anforderungen des *Model Checkings* gelegt.

Die Problematik der formalen Verifikation liegt in der Komplexität der involvierten Verfahren. Viele notwendige Algorithmen sind NP-vollständig, sodass sich die Analyse realer Problemstellungen enorm schwierig gestalten kann. Ein Hauptaugenmerk bei der Implementierung eines Verifikationssystems muss daher auf die Reduktion der Komplexität und die Optimierung des Quellcodes gelegt werden, damit auch große Probleme in einen handhabbaren Bereich gebracht werden können.

Die Komplexität steigt noch weiter, wenn nicht ein Einzelsystem, sondern ein System mit mehreren parallelen, kommunizierenden Komponenten betrachtet wird. Da interne, asynchrone Ereignisse der einzelnen Komponenten in willkürlicher Reihenfolge geschehen können, kann sich der Suchraum für die Fehlersuche exponentiell vergrößern. Die Herausforderung solche Systeme zu verifizieren gewinnt unter anderem an Bedeutung, da durch die zunehmende Vernetzung von Software im Großen, und die parallele Ausführung von Code in Mehrkernprozessorrechnern im Kleinen, die Häufigkeit nebenläufiger Problemstellungen steigt.

Um die Verifikation paralleler Systeme handhabbar zu machen ist ein Verfahren zur Reduktion des Suchraumes nötig. Unter den implementierten Algorithmen findet sich deshalb eine Implementierung der *Partial Order Reduction*, die dem Problem der Zustandsraumexplosion bei der Parallelisierung von endlichen Automaten mit der Konstruktion eines reduzierten Zustandsgraphen begegnet.

Die Arbeit ist folgendermaßen strukturiert: In Kapitel 2 wird eine kurze Einführung in die verwendeten

formalen Strukturen gegeben und die entwickelte Eingabesprache wird präsentiert.

Kapitel 3 bildet den Hauptteil dieser Arbeit. Hier werden sowohl der formale Hintergrund als auch die Implementierung der Algorithmen der Programmbibliothek beschrieben.

Kapitel 4 beschreibt die Programmarchtitektur und die einzelnen Komponenten der Implementierung.

2 Finite Automaten, Labelled Transition Systems und Prozessalgebra

Die mathematischen Strukturen, die in dieser Arbeit behandelt werden, sind die des *Finiten Automaten* und der *Labelled Transitions Systems*. Ein finiter Automat A ist ein Quintupel

$$A = (S, I, \Sigma, T, F)$$

wo S eine endliche Menge an Zuständen ist, $I \subseteq S$ eine Menge an Initialzuständen, Σ ein Alphabet aus Eingabesymbolen, $T \subseteq S \times \Sigma \times S$ eine Transitionsrelation und F die Menge an akzeptierenden Finalzuständen. Ein Automat akzeptiert ein Wort $\omega = \alpha_1\alpha_2 \ldots \alpha_n \in \Sigma^*$, wenn eine für eine Reihe an Zuständen $s_0, s_1, \ldots, s_n \in S$ gilt, dass $s_0 \in I \wedge T(s_0, \alpha_1, s_1) \wedge T(s_1, \alpha_2, s_2) \wedge \ldots \wedge T(s_{n-1}, \alpha_n, s_n) \wedge s_n \in F$. Die Menge an akzeptierten Wörtern $L(A)$ ist die Sprache des Automaten.

Um die Algorithmen möglichst allgemein formulieren zu können, werden bei der Implementierung nichtdeterministische Automaten angenommen. Deterministische Automaten können als Spezialfall von nichtdeterministischen Automaten aufgefasst werden: Ein deterministischer Automat mit dem Anfangszustand s_0 kann durch eine triviale Konstruktion in einen nichtdeterministischen Automaten mit der Initialmenge $I = \{s_0\}$ formuliert werden.

Anstatt von $T(s_h, \mu, s_i)$ wird der einfacheren Lesbarkeit halber auch $s_h \xrightarrow{\mu} s_i$ verwendet, eine Transitionskette der Form $s_j \xrightarrow{\mu_1} s_k \wedge s_k \xrightarrow{\mu_2} s_l$ wird zu $s_j \xrightarrow{\mu_1} s_k \xrightarrow{\mu_2} s_l$ abgekürzt.

Ein *Labelled Transition System* L ist ein Quartupel

$$L = (S, I, \Sigma, T)$$

. Bei *Labelled Transition Systems* stehen nicht die akzeptierten Wörter, sondern die möglichen Transitionssequenzen im Mittelpunkt des Interesses. Ein *LTS* L kann aber als Automat $A_L = (S, I, \Sigma, T, F = S)$ betrachtet werden, in dem jeder Zustand ein Finalzustand ist, und besitzt damit die implizite Sprache $L(A_L)$.

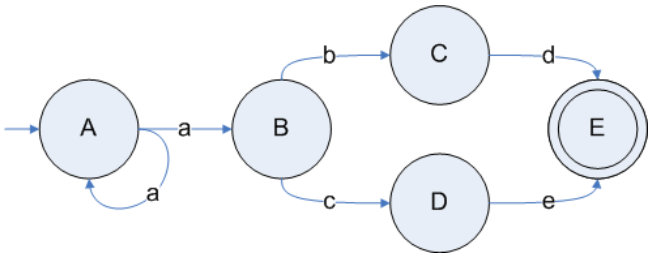
Die in der Programmbibliothek implementierten Algorithmen arbeiten ausschließlich mit finiten Automaten. Das Kommandozeilenprogramm akzeptiert *LTS* als Parameter und wandelt diese mit Hilfe der besprochenen Konstruktion automatisch in finite Automaten um.

2.1 Prozessalgebra zur Darstellung finiter Automaten

Das Kommandozeilenprogramm machte die Entwicklung einer Eingabesprache für finite Automaten notwendig. Bei der Wahl der Sprache spielten unter anderem folgende Überlegungen eine Rolle:

- Die Sprache sollte eine einfache Syntax und eine gute Lesbarkeit besitzen.

Abbildung 2.1: Endlicher Automat A_m



- Komplexe Automaten sollten sich wenig umständlich formulieren lassen.
- Die Sprache sollte sich leicht parsen lassen.

Es wurde eine an die Prozessalgebra angelehnte Darstellung für finite Automaten gewählt, die in ihrer Syntax einer Reihe von mathematischen Gleichungen entspricht. Ein finiter Automat wird durch mehrere Zeilen repräsentiert, die jeweils eine Definition eines Zustands mit seinen zugehörigen Transitionen enthalten.

Der Automat A_m (Abbildung 2.1) kann in der Eingabesprache folgendermaßen beschrieben werden:

$$\begin{aligned} iA &= a.A + a.B \\ B &= b.C + c.D \\ C &= d.E \\ D &= e.E \\ fE & \end{aligned}$$

Die bei der Zustandsdefinition vorangestellten Präfixe i und f zeigen an, ob ein Zustand ein Initial- beziehungsweise Finalzustand ist. Ein Zustand $X \in I \cap F$ würde als ifX oder auch fiX definiert werden. Zustandsnamen müssen mit Großbuchstaben beginnen.

Falls ein Zustand Transitionen besitzt, folgt nach dem Zustandsnamen ein Gleichheitszeichem, dem mehrere Transitionsdefinitionen folgen, die durch den Auswahloperator „ $+$ “ getrennt sind. Eine Transitionsdefinition besteht dabei mindestens aus einem Transitionssymbol, dem Zustandsüberführungsoperator „ $.$ “ und dem Transitionsziel. Transitionssymbole müssen mit Kleinbuchstaben beginnen.

Sequentielle Transitionsketten können mit dem Transitionsoperator kurz dargestellt werden. Ein Zustand $X0$ für den gilt $\exists X1, X2 : \; X0 \xrightarrow{j} X1 \xrightarrow{k} X2 \xrightarrow{l} X0$, und wo $X0$, $X1$ und $X2$ keine außer den angegebenen Transitionen besitzen kann folgendermaßen beschrieben werden:

$$X0 = j.k.l.X0$$

Die Zustände die das Transitionsziel von j und k sind, werden dabei automatisch erzeugt und benannt.

Genauso können Verzweigungen, die schließlich wieder in einen gemeinsamen Zustand münden, durch eine eingeklammerte Auswahl, der eine Transition auf das gemeinsame Transitionsziel folgt, zusammengefasst werden. Ein Zustand K, der mit den beiden Transitionssymbolen $k1$ und $k2$ auf sich selbst überführt kann als

$$K = (k1 + k2).K$$

dargestellt werden.

Die beiden Konzepte der sequentiellen Verkettung und Auswahl von Transitionen können beliebig miteinander kombiniert werden. So ließe sich der Automat aus 2.1 kurz folgendermaßen darstellen:

$$\begin{aligned} iA &= (a + a.(b.d + c.e)).E \\ fE & \end{aligned}$$

Transitionssymbole und Zustandsnamen dürfen dabei aus Buchstaben, Zahlen und den Sonderzeichen

$$\{\ \}\ ,\ |$$

zusammengesetzt sein. Wie bereits erwähnt, müssen Transitionssymbole mit Kleinbuchstaben und Zustandsnahmen mit Großbuchstaben beginnen.

Wird ein Zustand als Transitionsziel angegeben, aber selbst nie definiert, wird angenommen, dass er weder ein Final- noch ein Initialzustand ist, und keine Transitionen von ihm ausgehen. Mehrfachdefinitionen von Zuständen sind nicht erlaubt.

Die Eingabesprache ist kontexfrei. Im Abschnitt A des Anhangs findet sich eine entsprechende *EBNF*-Grammatik.

3 Algorithmen zur Transformation, Konstruktion und Analyse finiter Automaten

3.1 accepts()

Der einfachste der hier vorgestellten Algorithmen ist in der *accepts()* Methode implementiert und prüft, ob eine als Parameter gegebene Sequenz an Transitionssymbolen einer Sequenz an Transitionen entspricht, die den Automaten in einen akzeptierenden Zustand bringt. Gegeben eine Liste an Transitionssymbolen $\alpha_1, \alpha_2, \ldots, \alpha_n \in \Sigma$ muss also festgestellt werden, ob

$$\exists s_0 \in I, s_1, \ldots, s_{n-1} \in S, s_n \in F : s_0 \overset{\alpha_1}{\rightarrow} s_1 \overset{\alpha_2}{\rightarrow} \ldots \overset{\alpha_{n-1}}{\rightarrow} s_{n-1} \overset{\alpha_n}{\rightarrow} s_n$$

Mithilfe eines Breitensuche-ähnlichen Algorithmus werden, ausgehend von den Anfangszuständen, nacheinander mit jedem als Parameter übergebenen Transitionssymbol alle Überführungen durchgeführt. Die Menge der möglichen Nachfolger wird in jedem Schritt gespeichert und dient im nächsten Schritt wiederum als Ausgangspunkt der Transitionen mit dem nächsten Symbol.

Wenn nach der Transition mit dem letzten Symbol die Menge an Nachfolgern nicht leer ist und zudem einen Finalzustand beinhaltet, dann akzeptiert der Automat die Sequenz. Der Algorithmus gibt optional einen Trace der durchlaufenen Zustände aus, zu diesem Zweck wird zu jedem erreichten Zustand ein Zeiger auf den vorhergehenden Zustand mitgespeichert.

3.2 product()

product() erzeugt zu zwei Automaten den zugehörigen Produktautomaten. Der Produktautomat A zweier Automaten A_1 und A_2 mit gemeinsamen Alphabet $\Sigma = \Sigma_1 = \Sigma_2$ wird folgendermaßen definiert:

$$A = (S, I, \Sigma, T, F), A_i = (S_i, I_i, \Sigma_i, T_i, F_i)$$

$$A = A_1 \times A_2$$

$$S = S_1 \times S_2$$

$$I = I_1 \times I_2$$

$$F = F_1 \times F_2$$

$$T((s_h, s_i), \alpha, (s_j, s_k)) \iff T_1((s_h, \alpha, s_j) \wedge T_2((s_i, \alpha, s_k)$$

Die Sprache des Produktautomaten $L(A)$ ist die Schnittmenge der Sprachen der beiden Ausgangsautomaten

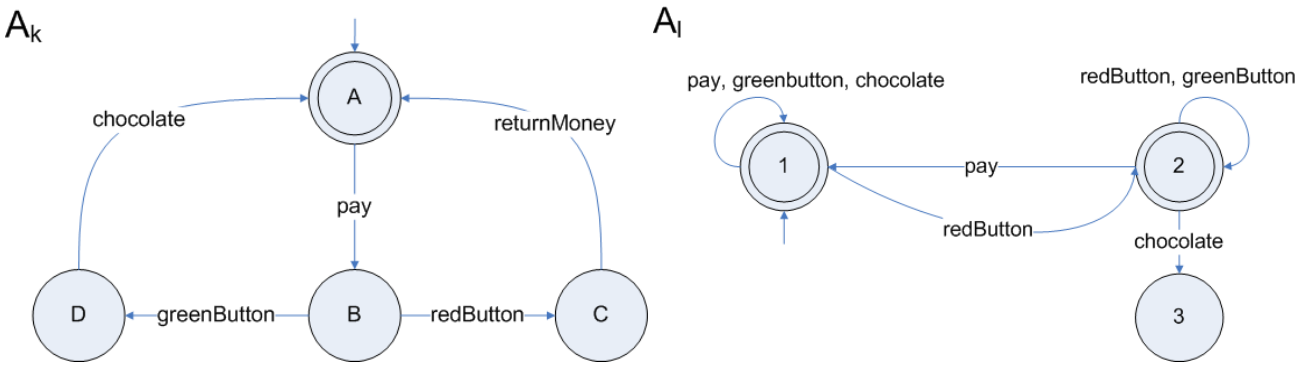


Abbildung 3.1: Schokoladeautomat A_k und Spezifikation A_l

$L(A_1) \cap L(A_2)$. Clarke et al. [2000, S. 123] zeigen wie Produktautomaten zur formalen Verifikationen von Systemen verwendet werden können. Sei A_k ein zu überprüfender endlicher Automat und A_l eine Spezifikation, die eine Menge an erlaubten Wörtern impliziert, dann erfüllt der Automat A_k die Spezifikation A_l genau dann, wenn

$$L(A_k) \subseteq L(A_l) \tag{3.1}$$

Diese Bedingung kann umformuliert werden zu

$$L(A_k) \cap \overline{L(A_l)} = \emptyset \tag{3.2}$$

wobei $\overline{L(A_l)} = \Sigma^* \setminus L(A_l)$. Sei $\overline{A_l}$ der Komplementautomat von A_l für den gilt $L(\overline{A_l}) = \overline{L(A_l)}$, dann sind die Bedingungen (3.1) beziehungsweise (3.2) genau dann erfüllt, wenn der Produktautomat $A_k \times \overline{A_l}$ keinen erreichbaren Finalzustand hat.

Der Automat A_k in Abbildung 3.1 stellt das Verhalten eines Schokoladeautomaten dar: Nach Einwurf einer gewissen Menge Geld, kann der Benutzer entweder eine grüne Taste drücken, um den Kauf der Tafel Schokolade zu bestätigen, oder eine rote Taste, die die Rückgabe des Geldes zur Folge hat. Die Spezifikation A_l geht genau dann in einen nicht-akzeptierenden Zustand, wenn der Automat nach Betätigen der roten Geldrückgabetaste Schokolade ausgibt, ohne dass zwischen diesen beiden Aktionen ein weiterer Bezahlvorgang stattgefunden hat. Eine solche Abfolge würde für den Benutzer wahrscheinlich kostenlose Schokolade und für den Entwickler einen Systemfehler bedeuten. Die Menge der unerlaubten Wörter ist $\Sigma^* \setminus L(A_l)$, kurz $L(\overline{A_l})$. Bei de-

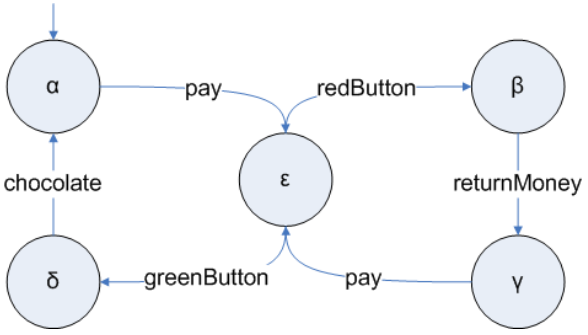


Abbildung 3.2: Produktautomat $A_k \times \overline{A_l}$

terministischen und vollständigen Automaten kann ein solcher komplementärer Automat $\overline{A_l}$ durch Invertieren der Finalzustände konstruiert werden. Die Konstruktion wird später im Abschnitt 3.4 genauer beschrieben.

Der Produktautomat $A_k \times \overline{A_l}$ (Abbildung 3.2) hat keinen erreichbaren Finalzustand, daher ist $L(A_k) \cap L(\overline{A_l}) = \emptyset$. A_k erfüllt die Spezifikation A_l.

Die Implementierung des Algorithmus unterscheidet sich in einigen Punkten von der eingangs gegebenen formalen Definition. Der resultierende Automat enthält nicht das volle Kreuzprodukt der beiden Zustandsmengen, sondern nur erreichbare Zustände. Diese Beschränkung hat in den meisten Anwendungsfällen keine negativen Konsequenzen und ist durch die erhöhte Übersichtlichkeit von Vorteil.

Die zweite Abweichung vom Formalismus liegt im Eingabealphabet der Automaten A_1 und A_2. Diese müssen nicht ident sein, allerdings müssen für eine erfolgreiche Überführung im resultierenden Automaten weiterhin beide Einzelautomaten überführen. Für das Alphabet des resultierenden Automaten wird daher immer $\Sigma \subseteq \Sigma_1 \cap \Sigma_2$ gelten. Wenn für das Alphabet der beiden Automaten etwa $\Sigma_1 \cap \Sigma_2 = \emptyset$, dann wird für den resultierenden Automaten gelten, dass $S = I_1 \times I_2$, da ausgehend von den Initialzuständen keine weiteren Transitionen möglich sind.

Das Alphabet wird in der Datenstruktur eines *finiten Automaten* nicht explizit modelliert, es können nur die *verwendeten* Symbole durch Traversieren der Zustände ermittelt werden. Aus diesem Grund ist es in der Datenstruktur nicht möglich, dass ein Transitionssymbol im Alphabet enthalten ist, mit dem niemals überführt wird. Eine Einschränkung des Algorithmus auf Automaten mit gleichem Alphabet würde daher eine Änderung der Datenstruktur nötig machen oder eine Einschränkung der Anwendbarkeit bedeuten. In den meisten Fällen würde ein Anwender eine so strenge Interpretation des Formalismus wahrscheinlich als störend empfinden.

Bei der Implementierung der Produktkonstruktion wurde ein modifizierter Breitensuchealgorithmus verwendet. Der Algorithmus wird mit dem kompletten Kreuzprodukt $I_1 \times I_2$ auf der Expansionsqueue initialisiert und erweitert alle möglichen synchronen Transitionen $l \in \Sigma_1 \cap \Sigma_2$. Wenn ein Zustandspaar auf der Expansionsqueue abgelegt wird, wird im resultierenden Automaten ein entsprechender Zustand erzeugt.

3.3 power()

Der Power-Automat $A_p = (S_p, I_p, \Sigma_p, T_p, F_p) = P(A)$ eines Automaten A ist wie folgt definiert:

$$\begin{aligned} S_p &= P(S) \\ I_p &= I \\ \Sigma_p &= \Sigma \\ F_p &= \{S' \in S_p \mid \exists s \in S' : s \in F\} \\ T_p(S, \alpha, S') &\iff S' = \bigcup_{s \in S} s \overset{\alpha}{\rightarrowtail} \end{aligned}$$

$\overset{\alpha}{\rightarrowtail}$ ist die Menge aller Nachfolger für ein Transitionssymbol α:

$$\begin{aligned} \overset{\alpha}{\rightarrowtail}\colon S &\mapsto P(S) \\ s &\mapsto \{s' \mid s \xrightarrow{\alpha} s')\} \end{aligned}$$

Der Power-Automat $P(A)$ eines endlichen Automaten A ist deterministisch und vollständig und akzeptiert die Sprache $L(A)$.

In der Implementierung werden natürlich ausschließlich erreichbare Zustände des Power-Automaten behandelt. Es wird ein Breitensuche-ähnlicher Algorithmus verwendet, der ausgehend von der Initialzustandsmenge $I \in I_p$ für alle Symbole des Alphabets die Nachfolgermengen bestimmt.

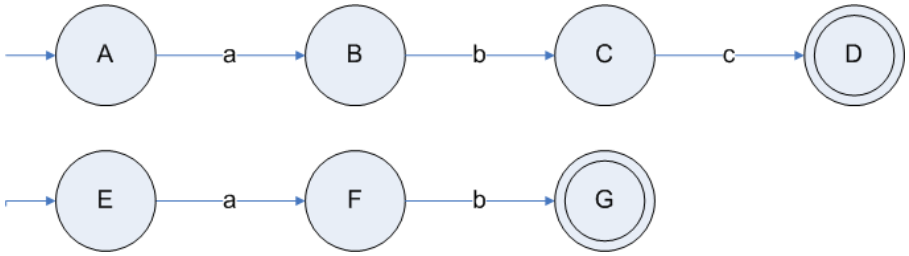


Abbildung 3.3: Nichtdeterministischer, unvollständiger Automat A_m

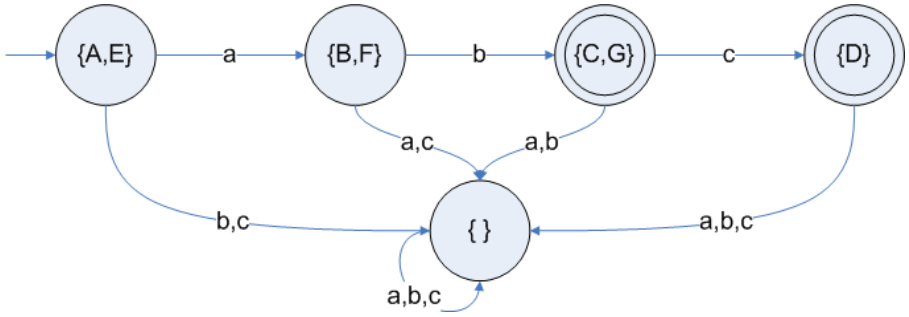


Abbildung 3.4: Power-Automat $P(A_m)$

3.4 complementLanguage()

Sei A ein beliebiger Automat und $L(A)$ die zugehörige Sprache, dann ist der Komplementär-Automat $\overline{A}$ definiert als

$$\overline{A} = (S, I, \Sigma, T, S \setminus F)$$

Ist der Automat A darüber hinaus deterministisch und vollständig, dann akzeptiert $\overline{A}$ die komplementäre Sprache zu A. (Biere [SS2005])

$$L(\overline{A}) = \overline{L(A)} = \Sigma^* \setminus L(A) \tag{3.3}$$

Vollständigkeit und Determinismus von A sind also hinreichende Bedingungen für (3.3). Im Folgenden wird der Grund erläutert.

Um die folgende Argumentation zu vereinfachen ist die Erweiterung des Transitionsoperators auf Wörter

sinnvoll. Für ein beliebiges $\omega \in \Sigma^*$, wo $\omega = \mu_0\mu_1 \dots \mu_n$, ist $\overset{\omega}{\rightarrowtail}^*$ folgendermaßen definiert:

$$\begin{aligned} \overset{\omega}{\rightarrowtail}^* : S &\mapsto P(S) \\ s &\mapsto \{s' \mid \exists s_0 \in I, s_1 \dots, s_n \in S : s_0 \overset{\mu_0}{\rightarrow} s_1 \overset{\mu_1}{\rightarrow} s_2 \overset{\mu_2}{\rightarrow} \dots \overset{\mu_{n-1}}{\rightarrow} s_n \overset{\mu_n}{\rightarrow} s'\} \end{aligned}$$

Sei A_u ein unvollständiger Automat, dann $\exists s \in S_u, \alpha \in \Sigma_u : s \overset{\alpha}{\rightarrowtail} = \emptyset$. Sei s_k so ein s und α_k so ein α, und gehen wir weiter davon aus, dass s_k erreichbar ist. Dies würde bedeuten, dass $\exists \omega \in \Sigma^*, S' \subseteq S, s_0 \in I : s_0 \overset{\omega}{\rightarrowtail}^* = S' \wedge s_k \in S'$. Sei $\mu_1\mu_2 \dots \mu_n$ ein solches Wort ω und $\overline{A_u}$ der Komplementär-Automat. Nun gilt $\mu_0\mu_1 \dots \mu_n\alpha \notin L(A_u)$ und $\mu_0\mu_1 \dots \mu_n\alpha \notin L(\overline{A_u})$, es gibt also ein Wort $\omega \in \Sigma^*$, das weder in $L(A_u)$ noch in $L(\overline{A_u})$ enthalten ist, daher ist die Komplementbedingung (3.3) nicht erfüllt und $L(\overline{A_u}) \neq \Sigma^* \setminus L(A_u)$.

Es ist jedoch durchaus denkbar einen unvollständigen Automaten zu konstruieren, für dessen Komplementär-Automat Bedingung (3.3) gilt, etwa indem man zu einem vollständigen Automaten einen unerreichbaren Zustand hinzufügt, der die Bedingung der Vollständigkeit verletzt.

Seien nun A_n ein nicht-deterministischer Automat und $\overline{A_n}$ sein Komplement, sodass das Wort $\omega = \mu_0\mu_1, \dots \mu_n$ in A_n folgende Eigenschaften besitzt:

$$\exists S' \subseteq S_n, s_0 \in I_n, s_f \in S' \cap F_n, s_{nf} \in S' \setminus F_n : s_0 \overset{\omega}{\rightarrowtail}^* = S'$$

Das Wort ω wird von A_n akzeptiert, da es in einen Finalzustand s_f überführt, gleichzeitig überführt ω aber auch in einen nicht-finalen Zustand s_{nf}. Innerhalb von $\overline{A_n}$ wird nun s_{nf} ein finaler und s_f ein nicht-finaler Zustand. Das Wort ω wird also auch in $\overline{A_n}$ akzeptiert. Damit ist $\omega \in L(A_n)$ und $\omega \in L(\overline{A})$, die Gleichung (3.3) gilt also nicht.

Determinismus ist keine notwendige Bedingung für (3.3). Vereinigt man etwa zwei vollständige, deterministische Automaten, die strukturell ident sind, zu einem einzigen nicht-deterministischem Automaten, dann gilt für die Vereinigung weiterhin (3.3).

Die Implementierung nimmt als Parameter den Automaten A und gibt einen Automaten zurück, der, wie bereits der Name der Funktion andeutet, die Bedingung (3.3) für A erfüllt. Wenn A deterministisch und vollständig ist, wird eine Kopie des Automaten zurückgegeben, deren Finalzustände invertiert wurden. Andernfalls wird zuerst ein Powerautomat $P(A)$ konstruiert um einen vollständigen und deterministischen Automaten mit der Sprache $L(A)$ als Basis für die Inversion der Finalzustände zu erhalten.

3.5 parallel()

Das parallele Produkt $A = A_1 \| A_2$ zweier Automaten A_1 und A_2 kann als parallele Ausführung zweier kommunizierender Systeme aufgefasst werden. Es wird folgendermaßen definiert:

$$\begin{aligned} S &= S_1 \times S_2 \\ I &= I_1 \times I_2 \\ \Sigma &= \Sigma_1 \cup \Sigma_2 \\ F &= F_1 \times F_2 \\ T &= T_s \cup T_a \end{aligned}$$

Das Synchronisationsalphabet ist die Schnittmenge der Alphabete beider Komponenten:

$$\Theta = \Sigma_1 \cap \Sigma_2$$

T_s und T_a entsprechen den synchronen beziehungsweise asynchronen Transitionen des Systems. Synchrone Transitionen sind solche Transitionen, die mit einem gemeinsamen, globalen Symbol $\alpha_s \in \Theta$ überführen. Asynchrone Transitionen führen mit lokalen Symbolen $\alpha_a \in \Sigma \setminus \Theta$ über.

T_s entspricht dabei der Transitionsfunktion des Produktautomaten, also einer gemeinsamen Transition beider Komponenten. Synchrone Transitionen werden auch als *Rendevouz* bezeichnet.

$$T_s \subseteq (S_1 \times S_2) \times \Theta \times (S_1 \times S_2)$$

$$T_s((s_h, s_i), \alpha, (s_j, s_k)) \iff T_1(s_h, \alpha, s_j) \wedge T_2(s_i, \alpha, s_k)$$

T_a repräsentiert asynchrone Überführungen einer der beiden Komponenten. Diese werden auch *Interleaving* genannt. Das *Interleaving-Modell* für nebenläufige Systeme modelliert asynchrone Überführungen, die gleichzeitig passieren können, indem alle möglichen Ausführungsreihenfolgen im Zustandsgraphen repräsentiert werden. Dieser Umstand ist eine der Hauptursachen für die *Zustandsraumexplosion*, die in Abschnitt 3.8 näher besprochen wird.

$$T_a \subseteq (S_1 \times S_2) \times ((\Sigma_1 \cup \Sigma_2) \setminus \Theta) \times (S_1 \times S_2)$$

$$\begin{aligned} T_a((s_h, s_i), \alpha, (s_j, s_k)) \iff\ & (\alpha \in \Sigma_1 \wedge T_1(s_h, \alpha, s_j) \wedge s_i = s_k) \\ & \vee(\alpha \in \Sigma_2 \wedge T_2(s_i, \alpha, s_k) \wedge s_h = s_j) \end{aligned}$$

Durch die Kombination von synchronen und asynchronen Überführungen stellt das parallele Produkt eine Nebeneinanderausführung kommunizierender Systeme dar. Lokale bzw. asynchrone Transitionen eines Automaten können unabhängig vom anderen Automaten ausgeführt werden und repräsentieren damit die unabhängige nebenläufige Ausführung beider Automaten, während gemeinsame Transitionen als Kommunikationsereignisse interpretiert werden können, die die beiden Komponenten des parallelen Produkts synchronisieren.

Auf diese Art können nebenläufige Systeme mit beliebig vielen parallelen Komponenten als *finite Automaten* modelliert werden. Das parallele Produkt kann dazu verwendet werden mehrere parallele Komponenten als ein Gesamtsystem zu betrachten. Dieses kann wiederum mit Techniken des *Model Checking* hinsichtlich gewisser Eigenschaften geprüft werden.

Die Systeme L_1 und L_2 in Abbildung 3.5 stellen zwei parallele Prozesse dar, die mit Hilfe eines einfachen Synchronisationsprotokolls gegenseitigen Ausschluss beim Lesen und Schreiben von Daten sicherstellt. Die Systeme sind als *Labelled Transition Systems* modelliert, da wir ausschließlich an möglichen Transitionssequenzen interessiert sind und nicht an der akzeptierten Sprache der Systeme. Für die Kompatibilität zu den bereits besprochenen Algorithmen kann ein *LTS* $L = (S, I, \Sigma, T)$ als endlicher Automat $L_A = (S, I, \Sigma, F = S, T)$ interpretiert werden.

Die beiden Automaten L_1 und L_2 sind sehr stark synchronisierte Systeme. Der Großteil der verwendeten Transitionssymbole ist Teil des Synchronisationsalphabets Θ. Entsprechend kompakt fällt das parallele Produkt $(L_1|L_2)$ (Abbildung 3.6) aus.

Analog zur Vorgehensweise in 3.2 kann das Gesamtsystem nun auf die Konformität zu einer gegebenen

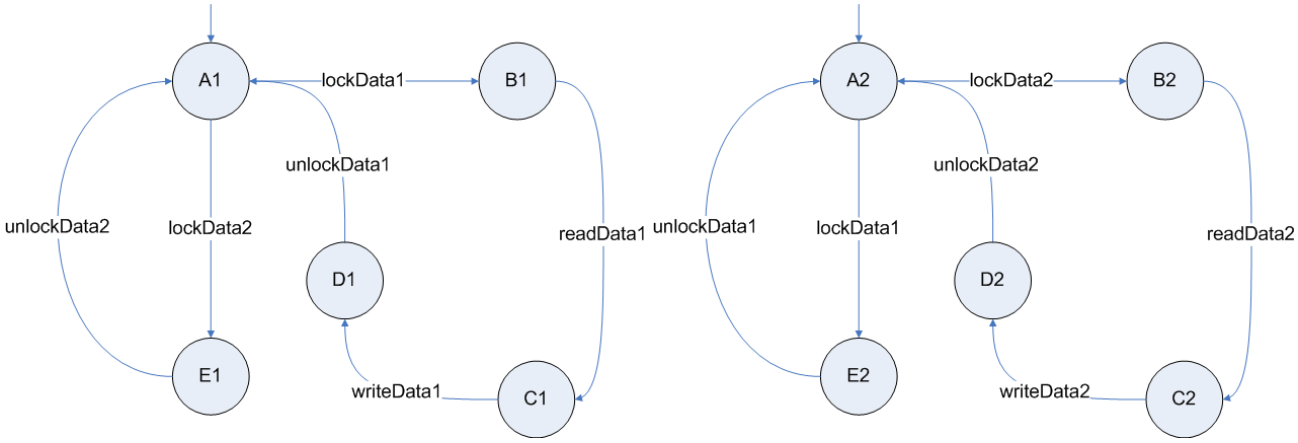


Abbildung 3.5: LTS L_1 und L_2

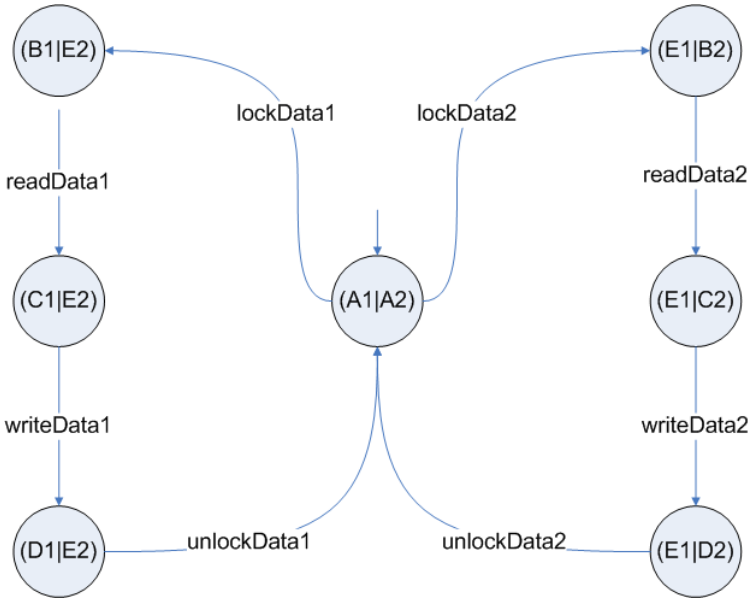


Abbildung 3.6: Parallele Komposition LTS $(L_1 \| L_2)$

Spezifikation geprüft werden. Sei A_s (Abbildung 3.7) ein endlicher Automat, der eine mögliche Spezifikation für den konfliktfreien Zugriff auf Daten darstellt.

Die Konstruktion $(L_1 \| L_2) \times \overline{A_s}$ (Abbildung 3.8) beinhaltet keinen erreichbaren Finalzustand. Das System $(L_1 | L_2)$ ist konform zu A_s.

Die Implementierung erfolgte analog zu den in den vorhergehenden Abschnitten besprochenen Verfahren mit Hilfe eines Breitensuche-ähnlichen Algorithmus. Bei der Initialisierung wird das Synchronisationsalphabet Θ ermittelt, beim Expandieren der Transitionen wird, abhängig davon ob das aktuelle Transitionssymbol in Θ enthalten ist ein *Rendevouz* oder *Interleaving* durchgeführt.

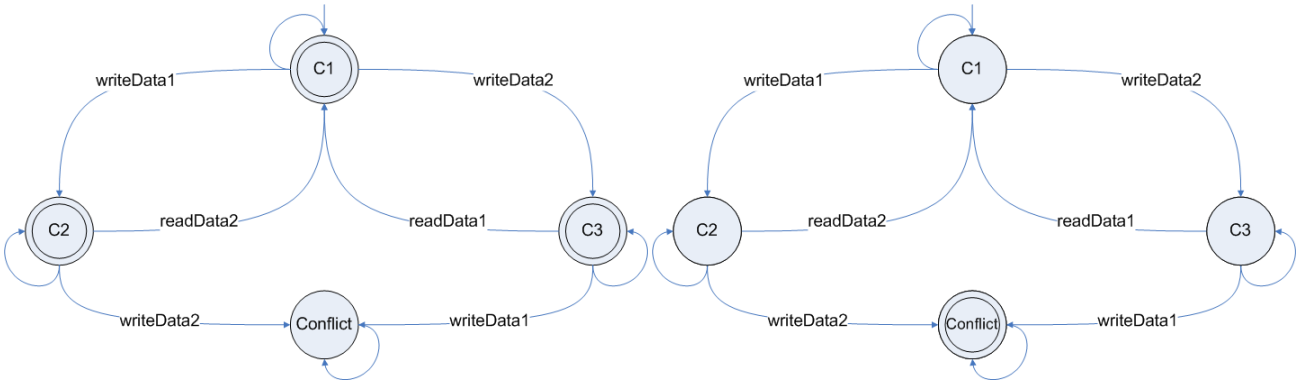


Abbildung 3.7: Die vollständige und deterministische Spezifikation A_s und $\overline{A_s}$. Reflexive Kanten sind unbeschriftet.

3.6 minimize()

Die *minimize()* Funktion berechnet ausgehend von einem Automaten A einen hinsichtlich der Zustandsanzahl minimalen Automaten A_{min}. Dabei gilt:

$$L(A) = L(A_{min})$$

$$\forall A_j : L(A_j) = L(A_{min}) \Longrightarrow |S_j| \geq |S_{min}|$$

Die Konstruktion des minimalen Automaten benötigt einen deterministischen und vollständigen Automaten als Ausgangspunkt. Falls der gegebene Automat diese Bedingungen verletzt, wird mit dem Power-Automaten gearbeitet. Der Algorithmus stammt aus Biere [WS2005/06].

Der Algorithmus zur Konstruktion des minimalen Automaten baut schrittweise eine Äquivalenzrelation zwischen den einzelnen Zuständen des Automaten auf. Beginnend bei $\sim_0 = (F \times F) \cup (\overline{F} \times \overline{F})$ werden $\sim_1, \sim_2, \ldots$ erzeugt, wobei $s \sim_{i+1} t$ genau dann gilt, wenn $s \sim_i t$ und

$$\begin{aligned} \forall \alpha \in \Sigma, s' \in S : (T(s, \alpha, s') &\Rightarrow (\exists t' \in S : T(t, \alpha, t') \wedge s' \sim_i t')) \\ \forall \alpha \in \Sigma, t' \in S : (T(t, \alpha, t') &\Rightarrow (\exists s' \in S : T(s, \alpha, s') \wedge s' \sim_i t')) \end{aligned}$$

Der Algorithmus läuft solange, bis $\sim_i = \sim_{i+1}$, was nach spätestens $|S|$ Schritten der Fall ist. Anschließend wird für jede Äquivalenzklasse in $\sim_i$ ein repräsentativer Zustand gewählt. Alle Transitionen zu den Elementen der Äquivalenzklassen werden schließlich auf den repräsentativen Zustand umgeleitet, die überflüssigen Zustände werden anschließend entfernt.

3.7 link()

Als Linking wird die komplette Ersetzung eines Transitionssymbols α durch ein anderes Symbol β bezeichnet. Der neue Automat $A_l = A \setminus [\beta / \alpha]$ gleicht dem Ursprungsautomaten A mit zwei Ausnahmen:

$$\Sigma_l = (\Sigma \setminus \{\alpha\}) \cup \{\beta\}$$

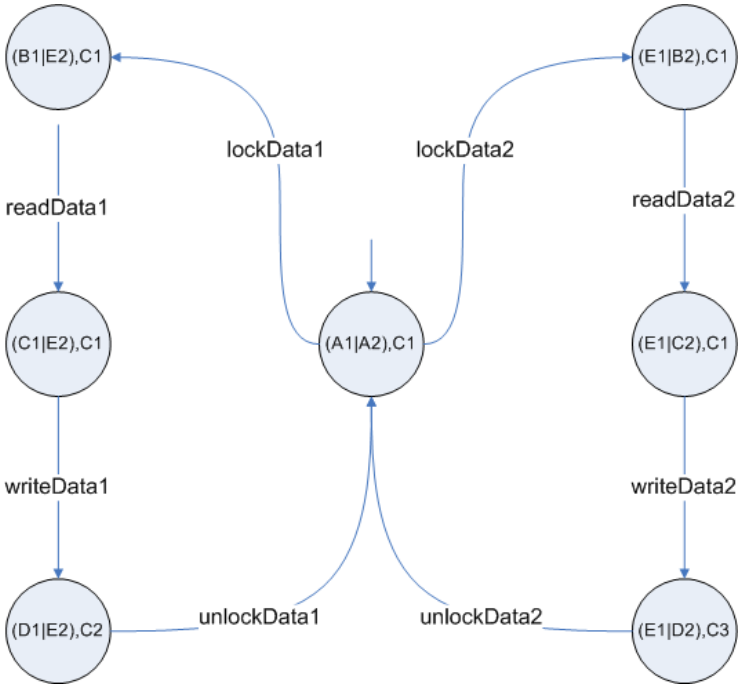


Abbildung 3.8: $(L_1|L_2) \times \overline{A_s}$

$$T_l(s,\gamma,s') \iff \begin{array}{ll} & T(s,\gamma,s') \\ \vee & (T(s,\alpha,s') \wedge \gamma = \beta) \end{array}$$

Linking kann eingesetzt werden um Automatentemplates zu instanziieren. So können etwa gleiche Systeme parallel geschaltet werden, indem derselbe Automat A_t zweimal verwendet wird und die lokalen Symbole durch Linking ersetzt werden. Würde kein Linking verwendet werden wäre $\Theta = \Sigma$ und daher $A_t \| A_t = A_t \times A_t$

3.8 partialOrderReduction()

Ein Problem bei der parallelen Komposition von Automaten ist die Explosion der Zustandsanzahl bei großen Systemen. Asynchrone Transitionen in den Komponenten der Komposition führen zu einer enormen Vergrößerung des Zustandsgraphen im resultierenden Automaten. Das Ausmaß der Vergrößerung wächst exponentiell mit der Anzahl der parallelen Komponenten.

Ein *Pfad* von einem Zustand $s \in S$ ist definiert als eine Sequenz von Zuständen $s_1, s_2, \ldots, s_n$ für die eine Reihe von Transitionssymbolen $\alpha_1, \ldots, \alpha_n$ existiert, sodass $s \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 \xrightarrow{\alpha_3} \ldots \xrightarrow{\alpha_n} s_n$. Wir nennen $\alpha_1, \ldots, \alpha_n$ die Transitionen des Pfades.

Sei A_{par} die parallele Komposition $A_{par} = A_1 \| A_2$ zweier Automaten und $(s_j, s_k) \in S_{par}$ ein Zustandspaar. Existiert nun sowohl in s_j als auch in s_k ein Pfad der Länge n, dessen Transitionssymbole $\alpha_1, \ldots, \alpha_n$ nicht im Synchronisationsalphabet enthalten sind, dann werden im resultierenden Automaten mindestens $(n+1)^2$ Zustände benötigt um die nebenläufige Ausführung dieser Transitionen zu repräsentieren.

Abbildungen 3.9, 3.10 und 3.11 veranschaulichen diese Problematik. A_α und A_β besitzen jeweils 5 Zustände. Von den Initialzuständen geht jeweils ein Pfad der Länge $n = 3$ mit asynchronen Transitionen aus. Die parallele Komposition $A_\alpha \| A_\beta$ benötigt bereits 17 Zustände. In realistischen Problemstellungen wächst so der

resultierende Automat sehr schnell auf eine nicht mehr handhabbare Größe an. Diese *Zustandsraumexplosion* bei nebenläufigen Systemen ist eine der größten Schwierigkeiten im *Model Checking*.

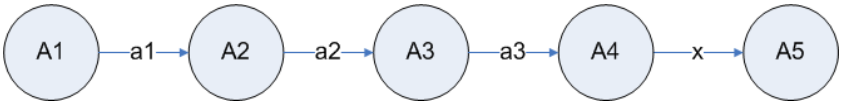


Abbildung 3.9: A_α

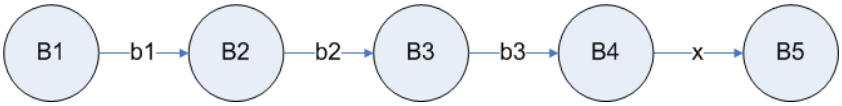


Abbildung 3.10: A_β

Die *Partial Order Reduction* ist eine Methode, die bei der Modellierung von nebenläufigen Systemen der *Zustandsraumexplosion* durch Konstruktion eines reduzierten Zustandsgraphen entgegenwirkt. Dabei wird unter anderem die Kommutativität von Transitionen ausgenutzt, die dann vorliegt, wenn unterschiedliche Ausführungsreihenfolgen zum selben Endzustand führen. Die Beschreibung des Verfahrens in diesem Abschnitt basiert im Wesentlichen auf Clarke et al. [2000, Kapitel 10].

Will man den Automaten $A_\alpha \| A_\beta$ (Abbildung 3.11) hinsichtlich der Erreichbarkeit des Zustands $(A5 \| B5)$ analysieren, so ist die Ausführungsreihenfolge der lokalen, asynchronen Transitionen ($a1 - a3$ sowie $b1 - b3$) von keinerlei Bedeutung. Aus den 20 unterschiedlichen Pfaden zwischen $(A1 \| B1)$ und $(A5 \| B5)$ kann daher genau einer repräsentativ ausgewählt werden, ohne dass dadurch relevante Informationen verloren gehen. Diese Auswahl von repräsentativen Teilgraphen aus dem vollständigen Zustandsgraphen ist die Grundidee, die dem Reduktionsverfahren zugrundeliegt. Die Methode wird daher auch *Model Checking using Representatives* genannt (Peled [1993]). Der Grund für den Namen *Partial Order Reduction* liegt in frühen Versionen des Algorithmus, denen das *Partial Order Modell* der Programmausführung zugrundelag. (Clarke et al. [2000, S. 141])

3.8.1 Auswahl von repräsentativen Pfaden

Bei der *Partial Order Reduction* wird für die parallele Komposition zweier Automaten ein reduzierter Zustandsgraph konstruiert, der ein für eine spezielle Problemstellung äquivalentes Verhalten besitzt wie der komplette Zustandsgraph. Der reduzierte Graph wird dann mit einem *Model Checking*-Algorithmus überprüft. Da der vollständige Zustandsraum nie explizit aufgebaut wird, wird das Problem der *Zustandsraumexplosion* umgangen.

Um die Effizienz des Verfahrens zu erhöhen, können die Konstruktion des reduzierten Zustandsgraphen und die Überprüfung mit dem *Model Checking*-Algorithmus in einem gemeinsamen Schritt durchgeführt werden.

Angenommen m parallele Komponenten eines Systems besitzen jeweils n konsekutive asynchrone Transitionen, dann wären für die Repräsentation des vollen Zustandsraumes des parallelen Produktes $(n+1)^m$ Zustände notwendig. Durch die Auswahl eines repräsentativen Pfades reduziert sich diese Zahl auf $n * m + 1$. Im Idealfall kann also eine *exponentielle Reduktion* des Zustandsraumes erreicht werden. Durch die verwendeten Auswahlregeln muss dabei garantiert sein, dass keine relevanten Informationen verlorengehen. Das Ergebnis der

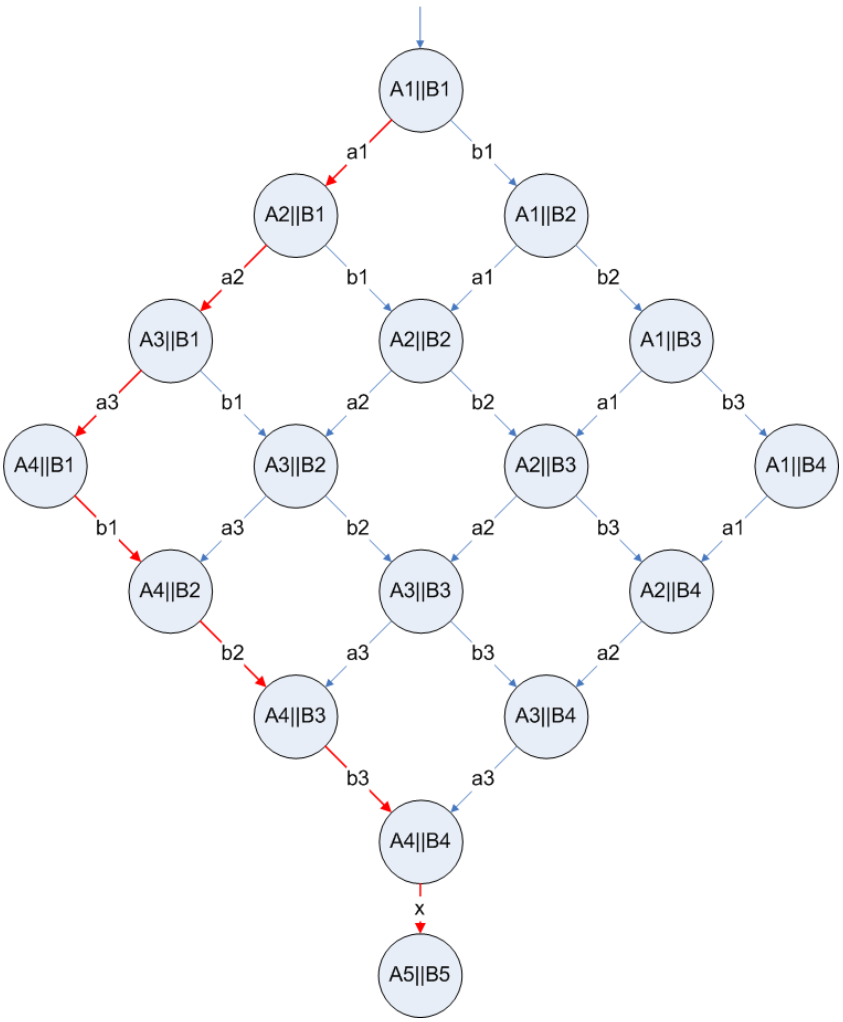


Abbildung 3.11: $A_\alpha || A_\beta$

Analyse wird dann trotz des stark eingeschränkten Suchraumes nicht beeinträchtigt.

Clarke et al. [2000] verwenden für ihre Beschreibung des Algorithmus *State Transition Systems*, die sich leicht von den hier besprochenen *finiten Automaten* unterscheiden. Weiters ist der ursprüngliche Algorithmus für die Überprüfung von *LTL*$_{-X}$ Bedingungen formuliert, während in der vorliegenden Arbeit parallele Systeme auf ihre Konformität zu einer Spezifikation geprüft werden sollen.

Im Folgenden wird daher eine leicht modifizierte Variante des Verfahrens von Clarke et al. [2000] präsentiert, die für die Arbeit mit finiten Automaten adaptiert ist. Im Speziellen bedeutet dies, dass der Algorithmus für n parallele Komponenten $A_1, \ldots, A_n$ und eine Spezifikation S einen reduzierter Zustandsgraphen für den vollen Graphen $(A_1 \| \ldots \| A_n) \times \overline{S}$ konstruiert. Der Automat $\overline{S}$ wird im Folgenden auch als Checker-Automat C bezeichnet.

Eine Transitionssymbol $\alpha \in \Sigma$ heißt *enabled* in einem Zustand s, wenn $\exists s' \in S : T(s, \alpha, s')$, ansonsten ist das Transitionssymbol *disabled*. Weiters ist *enabled*(s) die Menge aller Transitionsymbole die in s *enabled* sind. Bei der *Partial Order Reduction* kommt ein modifizierter Tiefensuchealgorithmus zum Einsatz, der ausgehend von den Initialzuständen rekursiv die Nachfolgezustände des parallelen Produkts expandiert. Anstatt jedoch alle Transitionen mit den Symbolen *enabled*(s) zu expandieren, wird mit einer repräsentativen Teilmenge *ample*$(s) \subseteq$ *enabled*(s) expandiert. Wenn *ample*$(s) \subset$ *enabled*(s) spricht man von einer partiellen Expansion, im komplementären Fall, dass *ample*$(s) =$ *enabled*(s), von einer vollen Expansion.

Der grundlegende Aufbau des Referenzalgorithmus nach Clarke et al. [2000] ist in Abbildung 3.12 dargestellt. Man beachte, dass der Algorithmus der Einfachheit halber eine deterministische Struktur des Systems annimmt. Der Aufruf von $\alpha(s)$ ließe sich im hier verwendeten Formalismus mit der Ermittlung von $s \overset{\alpha}{\rightarrowtail}$ vergleichen.

```
1  hash(s0);
2  set on_stack(s0);
3  expand_state(s0);
4
5  procedure expand_state(s)
6        work_set(s) := ample(s);
7        while work_set(s) is not empty do
8              let α ∈ work_set(s);
9              work_set(s) = work_set(s) \ α;
10             s' := α(s)
11             if new(s') then
12                   hash(s');
13                   set on_stack(s');
14                   expand_state(s');
15             end if;
16             create_edge(s, α, s');
17       end while;
18       set completed(s);
19 end procedure;
```

Abbildung 3.12: Partial Order Algorithmus von Clarke et al. [2000, Kapitel 10, S.143] für *State Transition Systems*

3.8.2 Berechnung von *ample(s)*

Die Berechnung von *ample*(s) stellt den Kernpunkt der *Partial Order Reduction* dar und ist verantwortlich für die angestrebte Reduktion des Suchraumes. Ersetzt man den Aufruf von *ample*(s) durch *enabled*(s), so erhält man die volle parallele Komposition. Clarke et al. [2000] geben für den Algorithmus für *ample*(s) folgende Kriterien an:

- Bei der Verwendung von *ample*(s) anstatt von *enabled*(s) dürfen keine Transitionen entfernt werden, deren Abwesenheit beim *Model Checking* zu einem falschen Ergebnis führt.
- Die Verwendung von *ample*(s) sollte zu einem Zustandsgraphen führen, der wesentlich kleiner ist als der volle Zustandsgraph.
- Da *ample*(s) in jedem Rekursionsschritt der Tiefensuche aufgerufen wird, muss die Zeit, die für die Berechnung nötig ist, ausreichend klein sein.

Außerdem werden 4 Bedingungen genannt, die *ample*(s) erfüllen muss. Deren Einhaltung garantiert, dass das Verhalten des reduzierten Zustandsgraphen zum Verhalten des vollen Graphen äquivalent im Bezug auf die zu überprüfende Eigenschaft ist. Die folgende Aufzählung basiert auf Clarke et al. [2000], ist aber an die Konformitätsprüfung bei *finiten Automaten* angepasst.

C0 $\mathit{ample}(s) = \emptyset \iff \mathit{enabled}(s) = \emptyset$

Wenn also im vollen Zustandsgraphen eine Transition in s existiert, dann muss im reduzierten Zustandsgraphen ebenfalls mindestens eine Transition existieren.

C1 Alle Transitionen in $\mathit{enabled}(s) \setminus \mathit{ample}(s)$ sind unabhängig von denen in $\mathit{ample}(s)$.

In einem Zustand s sind zwei Transitionen $T(s, \alpha, s')$ und $T(s, \beta, s'')$ dann unabhängig voneinander, wenn sie folgende Bedingungen erfüllen:

$$\alpha \in \mathit{enabled}(s'') \wedge \beta \in \mathit{enabled}(s')$$

$$\forall t \in S : T(s'', \alpha, t) \iff T(s', \beta, t)$$

Zwei Transitionen sind also unabhängig, wenn sie sich gegenseitig nicht deaktivieren und zusätzlich kommutieren, also jede Hintereinanderausführung der beiden Transitionen in denselben Zustand führt. Transitionen sind in einem Zustand s abhängig, wenn sie in s nicht unabhängig sind.

Die Kommutativität von Transitionen ist das grundlegende Kriterium für die Auswahl eines repräsentativen Pfades. Die Idee lässt sich am besten anhand von Abbildung 3.11 illustrieren. Da in jedem Zustand die Transitionen miteinander kommmutieren, und somit die Transitionsreihenfolge irrelevant ist, kann ein repräsentativer Pfad ausgewählt werden.

Wenn zwei Pfade in einem Zustand s mit voneinander abhängigen Transitionen beginnen und daher nicht kommutativ sind, können sie in verschiedene Bereiche des Zustandsgraphen führen, die komplett unterschiedliches Verhalten bezüglich der zu prüfenden Eigenschaft zeigen. Die partielle Expansion eines der beiden Pfade könnte das Ergebnis des *Model Checking* beeinflussen und potentiell verfälschen.

C2 Wenn s partiell expandiert wird, dann sind alle $\alpha \in \textit{ample}(s)$ hinsichtlich der zu prüfenden Eigenschaft unsichtbar.

Eine Transitionssymbol α ist unsichtbar hinsichtlich einer Spezifikation S mit dem Komplementautomaten $\overline{S} = C$, wenn in jedem Zustand s_C des Checker-Automaten C gilt, dass $s_C \xrightarrow{\alpha}= \{s_C\}$. Im Folgenden wird diese Eigenschaft auch als Invarianz des Checker-Automaten gegenüber einem Symbol α bezeichnet.

Die Bedingung *C2* stellt sicher, dass keine Transition aus dem vollen Zustandsgraphen entfernt wird, die möglicherweise für den *Model Checking*-Algorithmus relevantes Verhalten zeigt.

Eine wichtige Folgerung ist, dass das Ausmaß der Reduktion durch die Struktur des Checker-Automaten C beeinflusst wird. Im Idealfall ist C nur bei solchen Transitionssymbolen nicht invariant, deren Transitionen häufig voneinander abhängig sind, und daher bereits laut Bedingung *C1* Teil von *ample*(s) sein müssen.

C3 Im resultierenden Graphen darf kein zyklischer Pfad $s_0, s_1, \ldots, s_{n-1}, s_n, s_0$ enthalten sein, für den gilt, dass

- er einen Zustand beinhaltet, in dem ein Symbol α *enabled* ist
- und für jeden Zustand $s_k \in s_0, \ldots, s_n$ des Zyklus gilt $\alpha \notin \textit{ample}(s_k)$.

Gibt es in einem reduzierten Graphen einen solchen Zyklus, dann kann eine Transition (s, α, s') für immer verzögert werden. Ein Pfad ρ ausgehend von s, der mit der Transition (s, α, s') beginnt, wird in diesem Fall kein Teil des reduzierten Zustandsgraphen werden, obwohl er unter Umständen für das Ergebnis des *Model Checking* relevantes Verhalten zeigt.

3.8.3 Implementierung

Für die Implementierung musste ein einfacher und schneller Algorithmus zur Berechnung von *ample*(s) gefunden werden. Die Basisidee für den Algorithmus ist der Beschreibung in Biere [WS2005/06] entnommen. Das Verfahren zur Ermittlung des reduzierten Zustandsgraphen gleicht oberflächlich der Konstruktion der parallelen Komposition aus Abschnitt 3.5. Dabei werden aber nicht immer alle möglichen Transitionen verfolgt, sondern nur solche, die in $\textit{ample}(s) \subseteq \textit{enabled}(s)$ sind. Ausgehend von einem Zustandstupel $s = (s_1, \ldots, s_n)$ der parallelen Komposition $A_1 \| \ldots \| A_n$ wird

- partiell expandiert ($\textit{ample}(s) \subset \textit{enabled}(s)$) wenn ein Zustand s_k im Zustandstupel existiert, der ausschließlich asynchrone Transitionen $\alpha \notin \Theta$ besitzt. Bei mehreren solchen Zuständen kann ein beliebiger gewählt werden. Die partielle Expansion *ample*(s) besteht dabei aus allen Transitionssymbolen α, mit denen von s_k ausgehende Transitionen existieren ($\exists s_{succ} : T_k(s_k, \alpha, s_{succ})$).
- Wenn hingegen alle Zustände Transitionen mit Symbolen aus dem Synchronisationsalphabet Θ besitzen wird voll expandiert, sodass $\textit{ample}(s) = \textit{enabled}(s)$.

Wenn bei der partiellen Expansion eines Zustandes ein Zyklus geschlossen wird, der Bedingung *C3* verletzt, wird die partielle Expansion auf eine volle Expansion erweitert. Der grundlegende Aufbau des Verfahrens ist in Abbildung 3.13 ersichtlich.

```
1  Θ′ := Θ ∪ {α | is_not_checker_invariant(α)};
2
3  for   s_0 ∈ I_1 × ... × I_n
4        hash(s_0);
5        push(expansion_stack, s_0);
6  end for;
7
8  while not empty(expansion_stack)
9        s = (s_1, ..., s_n) := pop(expansion_stack);
10       expand_partial := false;
11
12       for i = 1 to n
13            if enabled(s_i) ≠ ∅ ∧ ∀α ∈ enabled(s_i) : α ∉ Θ′
14                 ample := enabled(s_i);
15
16                 if closes_unexpanded_cycle(ample(s));
17                      ample := enabled(s);
18                 end if
19
20                 expand_partial := true;
21                 break;
22            end if;
23       end for;
24
25       if not expand_partial
26            ample := enabled(s);
27       end if;
28
29       expand(s, ample);
30
31 end while;
32
33 procedure expand(s, ample)
34       for α ∈ ample
35            for s′ ∈ s --α-->
36                 if not hashed(s′)
37                      hash(s′);
38                      push(expansion_stack, s′);
39                 end if;
40                 create_edge(s, α, s′);
41            end for;
42       end for;
43 end procedure;
```

Abbildung 3.13: Schematischer Aufbau des implementierten Verfahrens

Zuallererst wird ein erweitertes Synchronisationsalphabet $\Theta' \supseteq \Theta$ erzeugt, das zusätzlich alle Symbole enthält, die gegenüber dem Checker nicht unsichtbar sind. Der Algorithmus wird mit dem vollen Kreuzprodukt $I_1 \times \ldots \times I_n$ auf dem Expansionsstack initialisiert, anschließend tritt er in eine entrekursivierte Tiefensucheschleife ein. Wenn eine Möglichkeit zur partiellen Expansion gefunden wird, wird überprüft, ob diese Expansion einen Zyklus verursacht, der Bedingung *C3* verletzt. Falls dies der Fall ist, wird voll expandiert. Wenn keine Möglichkeit zur partiellen Expansion gefunden werden kann, wird ebenfalls voll expandiert.

Der hier präsentierte Algorithmus für das reduzierte parallele Produkt ist der einzige der in dieser Arbeit vorgestellten konstruktiven Algorithmen, der für mehr als zwei Komponenten implementiert wurde. Dieser Umstand lässt sich darauf zurückführen, dass das hier vorgestellte Verfahren der *Partial Order Reduction* als einziges implementiertes Verfahren nicht assoziativ ist. Die Konstruktion $(A_1 \| _{PO} A_2) \| _{PO} A_3$ liefert normalerweise einen anderen Automaten als $A_1 \| _{PO} (A_2 \| _{PO} A_3)$, da sich die in die Berechnung involvierten Synchronisationsalphabete unterscheiden, die sehr wesentlich das Ausmaß der Reduktion bestimmen.

Eine Neuformulierung der Transitionsregeln im parallelen Produkt für mehr als zwei Komponenten ist daher notwendig. In der parallelen Komposition $A_1 \| A_2 \| \ldots \| A_n$ gilt:

$$\Theta = \{\alpha \in \Sigma_1 \cup \ldots \cup \Sigma_n \mid \exists A_j, A_k : A_j \neq A_k \wedge \alpha \in \Sigma_j \cap \Sigma_k\}$$

$$(s_1, \ldots, s_n) \xrightarrow{\alpha} (t_1, \ldots, t_n) \Longleftrightarrow \forall i \leq n : \ (\alpha \in \Sigma_i \Rightarrow s_i \xrightarrow{\alpha} t_i) \wedge (\alpha \notin \Sigma_i \Rightarrow s_i = t_i)$$

Um zu zeigen, dass der mit unserem Verfahren ermittelte reduzierte Zustandsgraph verhaltensäquivalent zum vollen Graphen ist, muss die Einhaltung der Bedingungen *C0* bis *C3* geprüft werden.

C0 Sei in einem Schritt der Tiefensuche $s = (s_1, \ldots, s_n)$ das aktuelle Zustandstupel und s_k eine Komponente dieses Tupels. Wenn nun eine partielle Expansion mit *ample*$(s) =$ *enabled*(s_k) versucht wird, dann ist durch die vorhergehende *if*-Abfrage gesichert, dass *enabled*$(s_k) \neq \emptyset$.

ample$(s) = \emptyset$ kann also nur bei vollständigen Expansionen wahr sein. Da für vollständige Expansionen gilt *ample*$(s) =$ *enabled*(s), gilt aber *ample*$(s) = \emptyset \Longleftrightarrow$ *enabled*$(s) = \emptyset$ und Bedingung *CO* ist erfüllt.

C1 Im Falle einer vollen Expansion ist die Erfüllung von *C1* trivial, da *enabled*$(s) \setminus$ *ample*$(s) = \emptyset$. Zu zeigen, dass unser Auswahlkriterium bei partiellen Expansionen *C1* erfüllt, ist ein wenig schwerer.

Bei einer partiellen Expansion werden die Überführungen einer der Komponenten s_k des Zustandstupels $s = (s_1, \ldots, s_n)$ expandiert. *ample*(s) ist daher die Menge an Transitionssymbolen, mit denen aus s_k überführt werden kann, und durch das Auswahlkriterium ist sichergestellt, dass Symbole in *ample*(s) nicht Teil des Synchronisationsalphabets Θ sind. Wir bezeichnen einen Zustand wie s_k, von dem ausschließlich asynchrone, beziehungsweise lokale Transitionen ausgehen, als lokalen Zustand.

Die Bedingung *C1* gilt für unseren Algorithmus also dann, wenn die Transitionen eines solchen lokalen Zustands s_k aus dem Zustandstupel s unabhängig von allen anderen Transitionen in *enabled*(s) sind.

Sei nun β ein beliebiges Transitionssymbol aus *enabled*$(s) \setminus$ *ample*(s) und seien $s_{j1}, \ldots, s_{jn}$ von s_k verschiedene Komponenten des Zustandstupels s, sodass für alle $1 \leq i \leq n$ gilt, dass $T(s_{ji}, \beta, s'_{ji})$. Der Beweis wird für den allgemeinen Fall geführt, dass n Komponenten mit β überführen. Wenn $\beta \notin \Theta$ dann ist $n = 1$, ansonsten gilt $n > 1$. Alle Transitionen werden der Einfachheit halber als determinis-

tisch angenommen, der Beweis ließe sich aber relativ einfach auf nicht-deterministische Überführungen erweitern.

Die erste Bedingung für Unabhängigkeit verlangt, dass sich zwei Transitionen nicht gegenseitig deaktivieren, also nach der Ausführung einer der beiden Transitionen immer die andere ausgeführt werden kann.

Sei nun α ein beliebiges Transitionssymbol aus *ample*(s) , sodass $T_k(s_k, \alpha, s'_k)$. Da kein Symbol aus *ample*(s), also auch nicht α, Teil des Synchronisationsalphabets ist, existiert in der parallelen Komposition eine *Interleaving*-Transition $T(s, \alpha, s')$, sodass s' beinahe ident mit s ist. Nur die Komponente s_k wurde durch ihren Nachfolger $s'_k \in s_k \overset{\alpha}{\longmapsto}$ ersetzt. Da $\beta \in$ *enabled*(s) ist, wissen wir, dass das Symbol β nur in den Alphabeten $\Sigma_{j1}, \ldots, \Sigma_{jn}$ enthalten ist. In jeder Komponente von s' deren Alphabet β enthält, existiert also weiterhin eine Transition mit β, womit gilt $\beta \in$ *enabled*(s').

Andererseits können wir auch mit der Transition $T(s, \beta, s'')$ überführen. Das Zustandstupel s'' ist bis auf die Nachfolger $s'_{j1} \in s_{j1} \overset{\beta}{\longmapsto}, \ldots, s'_{jn} \in s_{jn} \overset{\beta}{\longmapsto}$ ident mit s. Da s_k eine Komponente von s'' ist, gilt weiterhin $\alpha \in$ *enabled*(s''). Es gilt $\alpha \in$ *enabled*$(s'') \wedge \beta \in$ *enabled*(s'), wir haben nun die erste Bedingung für Unabhängigkeit gezeigt.

Um die Kommutativität der Transitionen zu zeigen, führen wir den Zustand s' nun mit dem Transitionssymbol β in den Nachfolgezustand t' über. s' enthält die Komponenten s'_k und $s_{j1}, \ldots, s_{jn}$. Wie wir bereits wissen, ist $\beta \notin \Sigma_k$. t' enthält daher weiterhin die Komponente s'_k und zusätzlich die Nachfolger $s'_{j1} \in s_{j1} \overset{\beta}{\longmapsto}, \ldots, s'_{jn} \in s_{jn} \overset{\beta}{\longmapsto}$ als Komponenten. Bis auf die genannten Komponenten s'_k anstatt von s_k und $s'_{j1}, \ldots, s'_{jn}$ anstatt von $s_{j1}, \ldots, s_{jn}$ ist t' ident mit s.

Analog führen wir nun s'' mit dem Transitionssymbol α in einen Zustand t'' über. s'' enthält anstatt der Komponenten $s_{j1}, \ldots, s_{jn}$ ihre Nachfolger $s'_{j1}, \ldots, s'_{jn}$, ansonsten ist s'' ident mit s. α ist ein asynchrones Transitionssymbol, das ausschließlich im Alphabet Σ_k enthalten ist. t'' enthält daher anstatt von s_k den Zustand s'_k, ist aber ansonsten ident mit s''. t'' enthält also s'_k anstatt von s_k und $s'_{j1}, \ldots, s'_{jn}$ anstatt von $s_{j1}, \ldots, s_{jn}$ ist aber ansonsten ident zu s. t'' enthält damit genau dieselben Komponenten wie t'. Damit gilt $t' = t''$, was die Kommutativität der Transitionssymbole α und β im Zustand s beweist.

Die Symbole α und β erfüllen somit beide Kriterien für Unabhängigkeit. Die hier vorgestellte Berechnung von *ample*(s) erfüllt die Bedingung *C1*.

C2 Die Bedingung *C2* wird durch die Konstruktion eines erweiterten Synchronisationsalphabets Θ' sichergestellt. Θ' enthält alle Symbole aus Θ und zusätlich alle jene Symbole, die gegenüber dem Checker-Automaten C nicht unsichtbar sind.

Da bei der Auswahl eines Kandidaten für die partielle Expansion nur lokale Zustände in Frage kommen, die nicht mit Symbolen aus Θ' überführen, ist damit sichergestellt, dass *ample*(s) bei einer partiellen Expansion nur unsichtbare Symbole beinhaltet.

C3 Wenn im vorgestellten Algorithmus eine partielle Expansion durchgeführt wird, wird überprüft, ob durch die Expansion ein Zyklus auftritt, der *C3* verletzt. Falls ein solcher Zyklus gefunden wird, wird die partielle Expansion auf eine volle Expansion erweitert.

Die Überprüfung erfolgt mithilfe des Tiefensuche-Suchstacks, der den aktuellen Suchpfad vom aktuellen Zustand bis zur Wurzel beinhaltet. Da das Verfahren nicht rekursiv arbeitet, muss dieser explizit konstruiert werden:

Ein Zustand wird auf den Suchstack gelegt, bevor er expandiert wird. Nachdem ein Zustand s vom Expansionsstack geholt wurde, wird ein *null-Zeiger* abgelegt. Wenn dieser wieder vom Expansionsstack geholt wird, steht fest, dass alle Nachfolger von s erfolgreich abgearbeitet wurden. Der Zustand s ist zu diesem Zeitpunkt das oberste Element des Suchstacks und wird von diesem entfernt.

Zu jedem Element des Suchstacks wird mitgespeichert, welche Art der Expansion bei diesem Element durchgeführt wurde: parallel oder vollständig. Findet sich bei der partiellen Expansion eines Zustandes s ein Nachfolger s', der schon einmal erreicht wurde, dann wird überprüft ob sich s' bereits auf dem Suchstack befindet. Falls dies der Fall ist und alle Zustände auf dem Suchstack beginnend bei s bis hin zu s' partiell expandiert wurden, muss die partielle Expansion von s auf eine volle Expansion erweitert werden, damit *C3* nicht verletzt wird.

Der implementierte Algorithmus kann sowohl mit als auch ohne Checker-Automat aufgerufen werden. Wenn ein Checker angegeben wird, wird gleich das synchrone Produkt des Checker-Automaten mit dem reduzierten Zustandsgraphen konstruiert, ansonsten wird lediglich ein reduzierter Zustandsgraph aufgebaut.

Es ist wichtig zu beachten, dass die Reduktion des Zustandsgraphen durch Bedingung *C2* abhängig vom verwendeten Checker-Automaten ist. Wird bei der *Partial Order Reduction* kein Checker angegeben und später das synchrone Produkt aus dem resultierenden Zustandsgraphen und einem Checker gebildet, ist das Ergebnis unter Umständen nicht richtig, wenn der Checker nicht-invariant gegenüber den asynchronen Transitionen der Komponenten des parallelen Produktes ist.

4 Aufbau der Implementierung

Bei der Entwicklung von FSMCalc gab es zwei wesentliche funktionale Zielsetzungen. Das erste Ziel war die Entwicklung einer Bibliothek, die Funktionen bereitstellt, mit denen Manipulationen und Konstruktionen von *finiten Automaten* und *Labelled Transition Systems* durchgeführt werden können, die unter anderem auf Anforderungen des *Model Checkings* zugeschnitten sind. Das zweite Ziel war die Kapselung der Hauptfunktionalitäten in einem Benutzerinterface, also die Schaffung eines „Schweizer Taschenmessers“ für die Arbeit mit *finiten Automaten*.

Die Wahl fiel dabei auf ein Kommandozeileninterface, da dieses einige Vorteile gegenüber anderen Interfaces bietet:

- Hohe Flexibilität
- Einfache Implementierung
- Einfache Automatisierung von komplexen Vorgängen durch betriebssystemeigene Skripts

Um die Kommandozeileneingabe von *finiten Automaten* zu ermöglichen war es notwendig, dem Benutzer eine Schnittstelle in Form einer Eingabesprache zur Verfügung zu stellen. Als weitere Komponenten musste also ein Parser entwickelt werden, der einen *finiten Automaten* in Textdarstellung einliest und in die Zieldatenstruktur transformiert, auf der später die relevanten Algorithmen operieren.

Der letzte wesentliche Bestandteil des Systems ist eine einfache grafische Oberfläche, die finite Automaten visuell darstellt. Die grafische Oberfläche wird über die Kommandozeile aufgerufen und stellt den Automaten mit einem einfachen Layouting-Algorithmus dar. Der Benutzer kann die einzelnen Zustände des Automaten beliebig arrangieren.

Der folgende Abschnitt befasst sich mit den technischen Aspekten der Implementierung der besprochenen Systemkomponenten. Die Systemarchitektur wird präsentiert, die getroffenen Designentscheidungen werden erläutert und die implementierten Algorithmen besprochen.

4.1 Wahl der Programmiersprache

Bei der Auswahl der Programmiersprache waren mehrere Faktoren ausschlaggebend. FSMCalc wurde für den Einsatz im akademischen Bereich entwickelt. Es bestehen viele Möglichkeiten für Erweiterung, etwa eine komplexe grafische Benutzeroberfläche, oder erweiterte Funktionalität. Aus diesem Grund ist es wünschenswert, den Code leicht verständlich und einfach erweiterbar zu halten. Die verwendete Programmiersprache muss diese Anforderungen reflektieren.

Die Struktur von *finiten Automaten* legt außerdem objektorientierte Programmierung nahe, um die die Übersichtlichkeit und Verständlichkeit des Quellcodes zu erhöhen und einfache Erweiterbarkeit zu gewährleisten.

Die gewählte Programmiersprache Java erfüllt alle diese Anforderungen und bietet zusätzlich den Vorteil der Plattformunabhängigkeit von kompiliertem Code.

4.2 Systemarchitektur

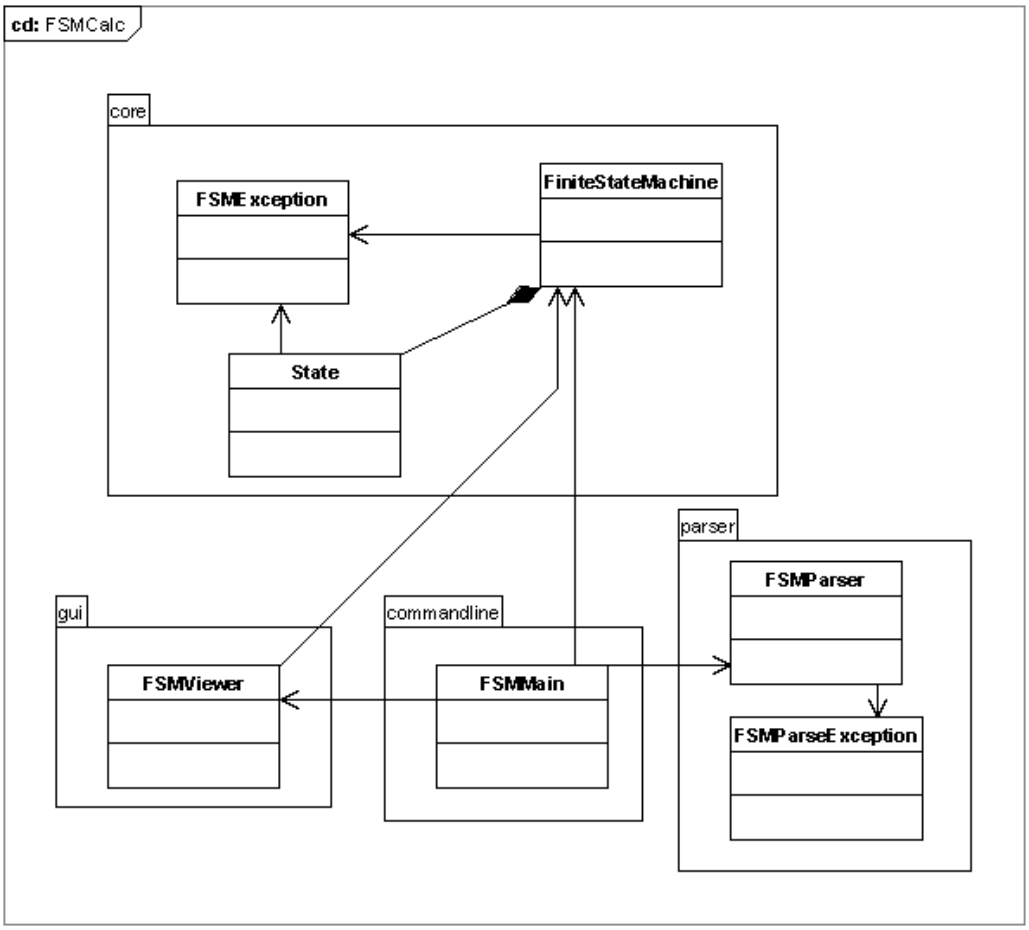


Abbildung 4.1: Pakete und Klassen

Das System besteht aus vier Komponenten die jeweils einem Paket entsprechen. Wie im Klassendiagramm in Abbildung 4.2 ersichtlich, ist die Klasse *FSMMain* im Paket *commandline* die zentrale Verbindung zwischen den einzelnen Komponenten. *FSMMain* ist eine ausführbare Klasse und bietet dem Benutzer über ein Kommandozeileninterface eine Schnittstelle zu den Funktionen der Programmbibliothek.

Das *core* Paket enthält die gemeinsame Datenstruktur für *finite Automaten* und *Labelled Transition Systems*. Eine Instanz der Klasse *FiniteStateMachine* repräsentiert einen *finiten Automaten*, der die ihm zugehörigen Zustände verwaltet. Transitionen zwischen den einzelnen Zuständen werden als Relationen innerhalb der Zustandsklasse State modelliert. Zusätzlich enthält das *core* Paket Algorithmen, mit denen Manipulations-, Tranformations- und Analyseaufgaben durchgeführt werden können.

Das Paket *parser* stellt eine Klasse zur Verfügung, die *finite Automaten* in textuellller Form von einem Stream einliest, und in die Datenstruktur des *core* Pakets transformiert. Treten beim Lesen Fehler auf, können Informationen über die Art des Fehlers und die Stelle im Stream, an der der Fehler auftritt, ausgegeben werden.

Das letzte Paket *gui* enthält eine einfache Visualisierungsklasse, die bei Instanzierung ein Fenster öffnet, in dem ein als Parameter übergebener *finiter Automat* dargestellt wird. Die Zustände werden durch einen einfachen Layout-Algorithmus platziert und können vom Benutzer beliebig angeordnet werden.

4.3 Datenstruktur

Finite Automaten und *Labelled Transition Systems* teilen sich eine gemeinsame Datenstruktur, die Klasse *FiniteStateMachine*. Die Klasse orientiert sich an der mathematischen Struktur eines endlichen Automaten und verwaltet drei Mengen an Zuständen: die Menge aller Zustände, die Menge der Initialzustände und die Menge der Finalzustände. Für schnellen Zugriff werden alle diese Mengen in Baumstrukturen verwaltet.

Bei der Arbeit mit der Programmbibliothek bietet die Klasse *FiniteStateMachine* Methoden zum Erzeugen, Löschen und Typisieren von Zuständen. Die implementierten FSM-Algorithmen können als Membermethoden der Klasse aufgerufen werden.

Labelled Transition Systems werden durch dieselbe Klasse repräsentiert, die Menge der Finalzustände werden dabei alle Zustände des Automaten hinzugefügt, jeder Zustand wird also als Finalzustand betrachtet.

Die Klasse *State* modelliert einen einzelnen Zustand. Innerhalb eines endlichen Automaten ist ein Zuständ über seinen Namen eindeutig identifizierbar. Jeder Zustand verwaltet seine eigenen Transitionen als Relationen zwischen Transitionssymbol und Nachfolgezustand und stellt Methoden zur Verfügung, mit denen Transitionen erzeugt oder vorhandene gelöscht werden können. Transitionen können nicht-deterministisch sein. Das bedeutet in der Implementierung, dass ein Transitionssymbol auf eine Liste an Transitionszielen verweist.

Das Alphabet des Automaten wird nicht global verwaltet, die Klasse *FiniteStateMachine* stellt aber eine Methode zur Verfügung, mit der das Alphabet durch Traversierung aller Zustände ermittelt werden kann.

Das mathematische Modell eines endlichen Automaten wird in der Implementierung also folgendermaßen repräsentiert:

Zustandsmenge S Explizit als Menge in der Klasse *FiniteStateMachine* realisiert

Inital- ($I \subseteq S$) und Finalzustände ($F \subseteq S$) Explizit als Mengen in *FiniteStateMachine* realisiert

Eingabealphabet Σ Implizit durch die verwendeten Transitionssymbole der einzelnen Zustände gegeben. Es ist daher nicht möglich, dass ein Symbol $l \in \Sigma$ niemals tatsächlich Teil eines Elements der Transitionsrelation ist.

Transitionsrelation $T \subseteq S \times \Sigma \times P(S)$ Nicht global definiert, sondern lokal für jeden Zustand. Der Zustand, in dem eine Transition definiert ist, ist der Ausgangszustand der Transition.

4.4 Parsing der Eingabesprache

Für die Eingabesprache für finite Automaten musste ein Parser implementiert werden, der die textuelle Repräsentation eines finiten Automaten einliest und eine Instanz der Klasse *FiniteStateMachine* zurückliefert.

Die Eingabesprache wird durch eine sehr einfache Grammatik erzeugt. Die Klasse *FSMParser* besitzt eine Methode die ausgehend von einem Eingabestream im *Single-Pass*-Verfahren das *Parsen* und die Transformation in die Zieldatenstruktur übernimmt.

Die Auflösung der Transitionen, die beliebig geschachtelt werden können, stellt den schwierigsten Teil des Vorganges dar. Für alle sequentiellen Transitionen, die nicht direkt auf einen Nachfolgezustand verweisen, wird ein anonymer Zwischenzustand im resultierenden Automaten erzeugt. Um etwa die Sequenz $A = a.b.c.D$ in die Zieldatenstruktur zu transformieren, werden zusätzlich zu den Zuständen A und D zwei anonyme Zustände An_0 und An_1 erzeugt, die das Transitionsziel der Symbole a beziehungsweise b sind.

Um Schachtelungen, wie etwa $A = (a.(b+c)+(d.e.f+g)).A$ aufzulösen, steigt der Algorithmus rekursiv in die einzelnen Schachtelungsebenen ab, erzeugt dabei die benötigten anonymen Zwischenzustände für sequentielle Transitionen, und liefert an die nächsthöhere Ebene die tiefsten gefundenen Transitionen des ermittelten Teilgraphen zurück. Im angebenen Fall wären die tiefsten Transitionen etwa jene, die mit den Symbolen b, c, f, und g überführen, da diese auf den endgültigen Nachfolger A verweisen.

Falls ein unerwartetes Symbol angetroffen wird, terminiert der Algorithmus mit einer Fehlermeldung, die Informationen über Zeile und Spalte und die Art des aufgetretenen Fehlers gibt.

4.5 Visualisierung

Die Visualisierungsklasse *FSMViewer* greift auf die *Open Source*-Programmbibliothek *JGraph* zurück, einem Framework zur Darstellung vernetzter Graphen, das sich als *Swing*-Komponente in ein *Java*-GUI integrieren lässt. Beim Initialisieren der Visualisierungsklasse werden die Zustände mit Hilfe eines Breitensuche-

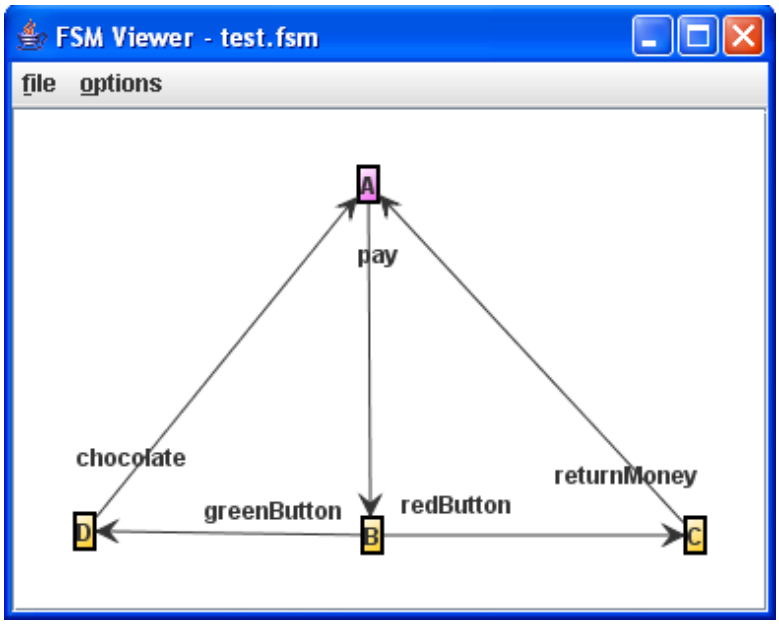


Abbildung 4.2: Der Schokoladeautomat (Abbildung 3.1) dargestellt von der Visualisierungsklasse *FSMViewer*

algorithmus in Zeilen und Spalten angeordnet. Die Zeilen korrespondieren mit der Länge des minimalen Pfades von einem Initialzustand. Initial- und Finalzustände werden durch farbliche Markierungen gekennzeichnet.

Der Benutzer kann Zustände und Beschriftungen von Zuständen und Transitionen beliebig selbst anordnen. Außerdem können verschiedene Einstellungen des *JGraph*-Frameworks, wie etwa die Routing-Optionen für Transitionskanten, angepasst werden.

5 Konklusion

In der vorliegenden Arbeit wurde ein Programm zur Arbeit mit finiten Automaten vorgestellt. Die grundlegenden formalen Strukturen der *Labelled Transition Systems* und *finiten Automaten* wurden erläutert und die für die Eingabe dieser Strukturen entwickelte Sprache besprochen.

Anschließend wurden die implementierten Algorithmen zur Konstruktion, Transformation und Analyse beschrieben. Die formalen Hintergründe wurden erläutert, wichtige Einsatzmöglichkeiten beschrieben und die konkrete Implementierung besprochen.

Schließlich wurde ein Überblick über die Programmarchitektur und die Implementierung der einzelnen Komponenten gegeben. Die grundlegend verwendeten Datenstrukturen, sowie die Spezialklassen zur visuellen Darstellung finiter Automaten und zum Einlesen der Eingabesprache, wurden beschrieben.

A EBNF-Grammatik der Eingabesprache

In diesem Kapitel wird eine *ISO-14977* EBNF-Grammatik für die entwickelte Eingabesprache, die in den Abschnitten 2.1 und 4.4 besprochen wird, präsentiert.

Das Startsymbol der Grammatik ist *FSMDef*.

```
FSMDef =
    BlanksOrNewlines , [StateDef],
    ['\n' {BlanksOrNewlines , [StateDef]}];

BlanksOrNewlines =
    {'\n' | Blanks};

Blanks =
    {'␣'};

StateDef =
    Blanks , StatePrefix , StateName , Blanks
    ['=', Blanks , TransitionDef , Blanks ,
    {'+', Blanks , TransitionDef , Blanks}];

StatePrefix =
    (['i'],['f']) | (['f'], ['i']);

TransitionDef =
    TransitionPart , Blanks , '.',
    Blanks , StateName;

TransitionPart =
    ( '(' , Blanks , TransitionPart , Blanks ,
    {'+', Blanks , TransitionPart , Blanks} ')' )
    | ( {TransitionSymbol , Blanks , '.', Blanks},
    TransitionSymbol );

StateName =
    CapitalLetter , ValidSequence;

TransitionSymbol =
    LowercaseLetter , ValidSequence;
```

```
ValidSequence =
    {CapitalLetter | LowercaseLetter |
    Number | AllowedSpecialChars};

CapitalLetter =
    'A' | 'B' | 'C' | 'D' | 'E' | 'F' |
    'G' | 'H' | 'I' | 'J' | 'K' | 'L' |
    'M' | 'N' | 'O' | 'P' | 'Q' | 'R' |
    'S' | 'T' | 'U' | 'V' | 'W' | 'X' |
    'Y' | 'Z';

LowercaseLetter =
    'a' | 'b' | 'c' | 'd' | 'e' | 'f' |
    'g' | 'h' | 'i' | 'j' | 'k' | 'l' |
    'm' | 'n' | 'o' | 'p' | 'q' | 'r' |
    's' | 't' | 'u' | 'v' | 'w' | 'x' |
    'y' | 'z';

AllowedSpecialChar =
    '{' | '}' | '|' | ',';
```

Literatur- und Quellenverzeichnis

Armin Biere. *Vorlesung Formale Grundlagen 3*. Johannes Kepler Universität, SS2005.

Armin Biere. *Vorlesung Systemtheorie 1*. Johannes Kepler Universität, WS2005/06.

Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, January 2000. ISBN 0262032708.

Doron Peled. All from one, one for all: on model checking using representatives. In *CAV '93: Proceedings of the 5th International Conference on Computer Aided Verification*, pages 409–423, London, UK, 1993. Springer-Verlag. ISBN 3-540-56922-7.