UNIVERSITÄT SIEGEN
FACHBEREICH MATHEMATIK

Hauptseminar

PLTL Model Checking

Benedikt Meurer
22.01.2007

INTERNE BERICHTE
INTERNAL REPORTS

Hauptseminar im
Fachbereich Mathematik
der Universität Siegen

Betreuer:
Prof. Dr. Dieter Spreen
Privatdozent Dr. Kurt Sieber

Zusammenfassung

Model Checking ist ein formales Verfahren zur Validierung von reaktiven Systemen. Es ist anderen heute gängigen Verfahren in der Softwaretechnik, wie zum Beispiel dem Testing, insofern überlegen, als dass nicht nur die Anwesenheit von Fehlern bewiesen werden kann, sondern auch deren Abwesenheit. Es ermöglicht eine vollständige Verifikation anhand einer Spezifikation. PLTL ist eine lineare, temporale Logik, die im folgenden zur Formulierung der Spezifikation benutzt wird. Ziel dieses Dokuments ist es den grundlegenden PLTL Model Checking Algorithmus vorzustellen.

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Organisation dieses Dokuments

Ziel dieses Dokuments ist es, die Grundlagen des PLTL-Model Checking Algorithmus zu vermitteln. Dazu liefert dieses Kapitel zunächst eine kurze Einführung in die Thematik. Kapitel 2 enthält eine Kurzeinführung in die Propositional Linear Temporal Logic (PLTL). Kapitel 3 beschreibt die benötigten Automatenmodelle. Kapitel 4 stellt den Zusammenhang zwischen diesen Automatenmodellen und der PLTL her. Kapitel 5 beschreibt schließlich den Algorithmus zur Konstruktion eines Automaten zu einer gegebenen PLTL-Formel. Kapitel 6 enthält abschließende Betrachtungen.

1.2 Validierung von Systemen

Der Begriff der *Validierung von Systemen* beschreibt den Prozess zur Überprüfung der Korrektheit von Spezifikationen, Entwürfen und Produkten in der Informationsverarbeitung. Dieser Prozeß gewinnt zunehmend an Bedeutung, was nicht zuletzt darauf zurückzuführen ist, dass die Informationsverarbeitung immer mehr Einzug in das tägliche Leben hält. [Kat99] enthält eine Schätzung der zu Folge 1995 jeder Mensch täglich mit durchschnittlich 25 informationsverarbeitenden Geräten interagierte.

Die Menschheit macht sich zunehmend abhängiger von Computern. Fehlerhafte Software oder Hardware führen deshalb nicht nur zu erheblichen wirtschaftlichen Schäden[1], sondern gefährden mittlerweile Menschenleben.

[1] Die geschätzten jährlichen Kosten durch Computerfehler allein in den USA belaufen sich mittlerweile auf über 60 Mrd. USD.

Seit Ende der 60er Jahre des vergangenen Jahrhunderts wurden deshalb unterschiedliche Ansätze entwickelt, um die Korrektheit von Systemen nachzuweisen oder zumindest die Fehlerwahrscheinlichkeit zu reduzieren.

Die Softwaretechnik bemüht sich durch die Entwicklung immer neuer Programmiersprachen und Methoden des Software-Engineering, Fehler bereits in den frühen Phasen der Entwicklung zu entdecken. Allerdings mit bisher mäßigem Erfolg. Viele dieser Methoden können nur triviale Designfehler aufdecken, wenn überhaupt. Zudem werden die Beschreibungsformen für Designdokumente stetig komplexer und bieten somit neuen Nährboden für Fehler und Missverständnisse. Ein aktuelles Beispiel hierfür ist die UML 2.0 Spezifikation.

Da jedoch generell nicht alle Fehler vermieden werden können, wurden *Simulation* und *Testing* als Techniken zum Aufspüren von Fehlern entwickelt. Die *Simulation* ermöglicht das Nachweisen von Fehlern in der Designphase, das *Testing* kann nur auf das fertige Produkt oder Modul angewandt werden. Während diese Methoden leicht anzuwenden sind, auch für unerfahrene Entwickler, und darüber hinaus viele Fehler relativ schnell und kostengünstig aufdecken, können sie aber weder Fehlerfreiheit garantieren, noch gibt es eine verlässliche Möglichkeit, die Anzahl der verbliebenen Fehler einzuschätzen.

Eine erfolgversprechendere Alternative stellt die *formale Verifikation* dar. Hierbei wird mittels einer geeigneten Logik die Korrektheit des betrachteten (Teil-)Systems bewiesen.

1.3 Formale Verifikation

Heute existieren unterschiedliche Ansätze zur *formalen Verifikation* von Systemen, die grob nach folgenden Kriterien klassifiziert werden können (vgl. [HR04, S. 172f]):

- **Beweisbasiert vs. modellbasiert.** In einem beweisbasierten Verfahren wird das System als Menge von Formeln Γ (in einer geeigneten Logik) und die Spezifikation als Formel ϕ beschrieben. Die Verifikation erfolgt dann als Auffinden eines Beweises für $\Gamma \models \phi$.
  In einem modellbasierten Verfahren wird stattdessen das System als Modell $\mathcal{M}_\mathcal{S}$ dargestellt, und die Spezifikation wiederum als eine Formel ϕ. Die Verifikation beschränkt sich dann darauf zu prüfen, ob $\mathcal{M}_\mathcal{S} \models \phi$ gilt. Modellbasierte Verfahren sind üblicherweise einfacher, da sie auf einem einzigen Modell $\mathcal{M}_\mathcal{S}$ basieren, während zum Beweis von $\Gamma \models \phi$ für

alle Modelle $\mathcal{M}$ geprüft werden muss, ob $(\forall \psi \in \Gamma : \mathcal{M} \models \psi) \Rightarrow \mathcal{M} \models \phi$ gilt.

- **Grad der Automatisierung.** Die Verfahren unterscheiden sich darüber hinaus im Grad der Automatisierung. Die meisten Verfahren sind semiautomatisch, erfordern also immer noch einen gewissen Grad an Interaktion mit dem Programmierer.

- **Vollständige vs. teilweise Verifikation.** Die Spezifikationen können entweder das System als Ganzes beschreiben oder nur einen Teilbereich des Systems oder eine bestimmte interessante Eigenschaft. Üblicherweise will man nur bestimmte Eigenschaften verifizieren.

- **Anwendungsbereich,** in dem das Verfahren eingesetzt werden kann.

- **Phase der Entwicklung,** in der das Verfahren zum Einsatz kommt. Besonders interessant sind Verfahren, die schon in der Designphase zum Einsatz kommen, da die Kosten für Fehlerkorrekturen in frühen Phasen deutlich niedriger sind als in späteren Phasen.

Nach den Maßstäben der obigen Klassifizierung ist die in diesem Dokument vorgestellte Methode des *Model Checking* als automatisches, modellbasiertes Verfahren zum Beweisen von Eigenschaften eines nebenläufigen, reaktiven Systems einzuordnen. Kritische Fehler in nebenläufigen, reaktiven Systemen sind üblicherweise durch *Testing* schwer oder gar nicht zu finden, da sie selten einfach zu reproduzieren sind.

Eine für reaktive Systeme geeignete Logik stellt die *Propositional Linear Temporal Logic* (PLTL) dar, die in Kapitel 2 kurz eingeführt wird. Neben der PLTL gibt es noch weitere für Model Checking geeignete Logiken, zum Beispiel die *Computation Tree Logic* (CTL), auf die aber im Rahmen dieses Dokuments nicht weiter eingegangen werden soll.

Kapitel 2

Einführung in PLTL

In diesem Abschnitt wird eine kurze Einführung in die Propositional Linear Temporal Logic (PLTL) gegeben. Ausführliche Einführungen finden sich in [HR04], [Jun06] und [Kat99].

2.1 Syntax von PLTL

Die Grundlage bilden *Propositionalzeichen* (engl. *atomic propositions*), also Aussagen, die nicht weiter zerlegt werden können. Die Menge der Propositionalzeichen bezeichnen wir im folgenden mit AP. Hinter einem Propositionalzeichen kann sich zum Beispiel eine Aussage wie „x ist größer als 0“ oder „x ist gleich 1“ verbergen.

Definition 2.1 (Syntax von PLTL) *Sei AP eine Menge von Propositionalzeichen. Dann ist die Menge der PLTL-Formeln induktiv definiert durch:*

1. *Jedes $p \in AP$ ist eine Formel.*
2. *Ist ϕ eine Formel, so ist auch $\neg\phi$ eine Formel.*
3. *Sind ϕ und ψ Formeln, dann ist auch $\phi \vee \psi$ eine Formel.*
4. *Ist ϕ eine Formel, dann ist auch $\mathbf{X}\phi$ eine Formel.*
5. *Sind ϕ und ψ Formeln, dann ist auch $\phi \,\mathbf{U}\, \psi$ eine Formel.*

Die Aussagenlogik ist also eine echte Teilmenge der PLTL. Hinzu kommen die temporalen Operatoren **X** („neXt“) und **U** („Until“).

Des weiteren werden die aussagenlogischen Operatoren $\wedge$ (Konjunktion), $\rightarrow$ (Implikation) und $\leftrightarrow$ (Äquivalenz) sowie die temporalen Operatoren **F** („Future“) und **G** („Globally“) als syntaktischer Zucker eingeführt:

$$\begin{aligned} \phi \wedge \psi &= \neg(\neg\phi \vee \neg\psi) \\ \phi \rightarrow \psi &= \neg\phi \vee \psi \\ \phi \leftrightarrow \psi &= (\phi \rightarrow \psi) \wedge (\psi \rightarrow \phi) \end{aligned}$$

$$\begin{aligned} \mathbf{F}\phi &= \top \mathbf{U}\, \phi \\ \mathbf{G}\phi &= \neg\mathbf{F}\neg\phi \end{aligned}$$

Hierbei bezeichnet $\top$ die Formel $p \vee \neg p$ für ein beliebiges $p \in AP$. Das Falsum $\bot$ wird entsprechend als $\neg\top$ eingeführt.

2.2 Semantik von PLTL

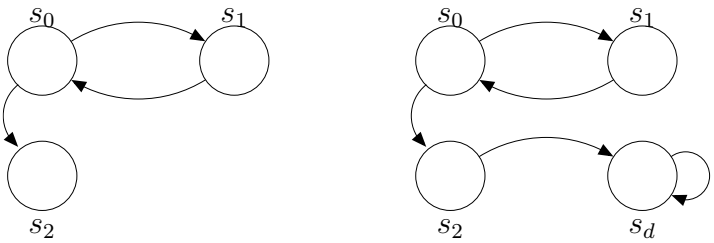
Definition 2.2 (Modelle für PLTL) *Ein* PLTL-Modell *ist ein Tripel* $\mathcal{M} = (S, \rightarrow, L)$. *Hierbei ist*

- S *eine nicht-leere, abzählbare Menge von Zuständen,*
- $\rightarrow \subseteq S \times S$ *eine binäre Relation, so dass zu jedem Zustand* $s \in S$ *genau ein Zustand* $s' \in S$ *mit* $(s, s') \in \rightarrow$ *existiert, und*
- $L : S \rightarrow \mathcal{P}(AP)$ *eine „labelling function“, die jedem Zustand aus* S *eine Menge von Propositionalzeichen zuordnet.*

Die Relation $\rightarrow$ ordnet jedem Zustand $s \in S$ in eindeutiger Weise einen Folgezustand zu. Für $s, s' \in S$ schreiben wir $s \rightarrow s'$, falls $(s, s') \in \rightarrow$.

Die Forderung, dass zu jedem Zustand $s \in S$ ein Folgezustand $s' \in S$ existieren muss, bedeutet, dass Modelle nicht in Zuständen stecken bleiben können (engl. *deadlock*). Hierbei handelt es sich in Wirklichkeit um eine technische Vereinfachung und nicht etwa um eine Beschränkung der Modelle. Denn jedes Modell, welches mindestens einen Zustand enthält, der stecken bleiben würde, kann stets um einen neuen Zustand s_d, der den „Deadlock“ repräsentiert, erweitert werden. Dies ist in Beispiel 2.1 dargestellt.

Beispiel 2.1 (Deadlocks) *Betrachten wir den Graphen*[1] *des Systems auf der linken Seite. Dieses entspricht nicht der Definition eines PLTL-Modells, da der Zustand s_2 keinen Folgezustand besitzt.*



Erweitern wir das System nun um einen Zustand s_d, wie auf der rechten Seite gezeigt, und die Relation $\rightarrow$ um Paare (s, s_d) für alle Zustände s, die zuvor steckengeblieben wären, so erhalten wir ein gültiges PLTL-Modell. Intuitiv entspricht nun das Erreichen des Zustands s_d dem „Deadlock“ im ursprünglichen System.

Definition 2.3 (Pfade in PLTL-Modellen) *Ein* Pfad *in einem PLTL-Modell $\mathcal{M} = (S, \rightarrow, L)$ ist eine unendliche Folge von Zuständen $s_1, s_2, s_3, \ldots \in S$, so dass gilt $\forall i \geq 1 : s_i \rightarrow s_{i+1}$. Wir schreiben dann $s_1 \rightarrow s_2 \rightarrow \ldots$ für den Pfad.*

Ein Pfad $\pi = s_1 \rightarrow s_2 \rightarrow \ldots$ stellt eine mögliche Zukunft eines Systems dar: Zuerst befindet sich das System in Zustand s_1, anschliessend in Zustand s_2 und so weiter. Wir schreiben π^n für des Restpfad beginnend beim Zustand s_n, zum Beispiel π^4 für den Pfad $s_4 \rightarrow s_5 \rightarrow \ldots$.

Definition 2.4 (Pfade und PLTL-Formeln) *Sei $\mathcal{M} = (S, \rightarrow, L)$ ein PLTL-Modell, $\pi = s_1 \rightarrow s_2 \rightarrow \ldots$ ein Pfad in $\mathcal{M}$, ϕ, ψ PLTL-Formeln und $p \in AP$. Die Relation $\models$ ist wie folgt induktiv definiert:*

1. *$\pi \models p$ gdw. $p \in L(s_1)$*
2. *$\pi \models \neg\phi$ gdw. $\pi \not\models \phi$*
3. *$\pi \models \mathbf{X}\phi$ gdw. $\pi^2 \models \phi$*
4. *$\pi \models \phi \vee \psi$ gdw. $\pi \models \phi$ oder $\pi \models \psi$*
5. *$\pi \models \phi \,\mathbf{U}\, \psi$ gdw. es existiert ein $i \leq 1$ mit $\pi^i \models \psi$ und für alle $1 \leq j < i$ gilt $\pi^j \models \phi$*

[1] Zur Vereinfachung wurden hierbei die Markierungen für die Zustände weggelassen.

Definition 2.5 (Semantik von PLTL) *Sei $\mathcal{M} = (S, \rightarrow, L)$ ein PLTL-Modell, $s \in S$ und ϕ eine PLTL-Formel. Dann schreiben wir $\mathcal{M}, s \models \phi$, wenn für jeden Pfad π von $\mathcal{M}$, der in s beginnt, $\pi \models \phi$ gilt.*

Gemäß der formalen Semantik bedeutet $\mathbf{X}\phi$ also intuitiv, dass im nächsten Zustand ϕ gelten muss, und $\phi \,\mathbf{U}\, \psi$, dass ϕ solange gelten muss, bis ein Zustand erreicht ist, in dem ψ gilt; dieser Zustand kann auch der aktuelle Zustand sein, ϕ muss also kein einziges Mal gelten, wenn ψ schon gilt.

Dementsprechend bedeutet $\mathbf{F}\psi$, dass es in der Zukunft mindestens einen Zustand geben muss, in dem ψ gilt, und $\mathbf{G}\phi$, dass im aktuellen Zustand und in allen folgenden Zuständen ϕ gelten muss.

Es ist trivial einzusehen, dass für ein beliebiges $p \in AP$ jedes Modell die Formel $\top = p \vee \neg p$ erfüllt, und kein Modell existiert, welches $\bot = \neg(p \vee \neg p)$ erfüllt.

Definition 2.6 (Äquivalenz von PLTL-Formeln) *Zwei PLTL-Formeln ϕ und ψ heissen (semantisch)* äquivalent*, geschrieben $\phi \equiv \psi$, wenn für alle Modelle $\mathcal{M}$ und alle Pfade π in $\mathcal{M}$ gilt: $\pi \models \phi$ gdw. $\pi \models \psi$.*

Sind zwei Formeln ϕ und ψ äquivalent, so heissen ϕ und ψ *semantisch austauschbar*. Wenn zum Beispiel ϕ Teilformel einer grösseren Formel χ ist, und $\phi \equiv \psi$, so kann ψ für ϕ in χ substituiert werden, ohne die Bedeutung von χ zu ändern.

2.3 Normalform von PLTL-Formeln

Die in Abschnitt 2.1 angegebene minimale Syntax für PLTL-Formeln ist zwar vollständig, aber für den im folgenden vorgestellten PLTL-Model Checking Algorithmus nicht direkt geeignet. Deshalb führen wir hier zunächst eine Normalform für PLTL-Formeln ein.

Definition 2.7 (Normalform von PLTL-Formeln) *Sei $p \in AP$ ein Propositionalzeichen, dann ist durch*

$$\phi ::= p \mid \neg p \mid \phi \vee \phi \mid \phi \wedge \phi \mid \mathbf{X}\phi \mid \phi \,\mathbf{U}\, \phi \mid \phi \,\bar{\mathbf{U}}\, \phi$$

die Menge $\mathbb{F}$ der gültigen PLTL-Formeln in Normalform beschrieben.

Eine Formel ist also in Normalform, wenn Negationen nur direkt vor einem Propositionalzeichen stehen und keinen der temporalen Operatoren **F** und **G** enthält. Letzteres kann erreicht werden, indem alle Vorkommen von **F** und **G** anhand ihrer Definitionen

$$\begin{aligned} \mathbf{F}\phi &\equiv \top \mathbf{U}\, \phi \\ \mathbf{G}\phi &\equiv \neg\mathbf{F}\neg\phi \end{aligned}$$

eliminiert werden. Darüber hinaus müssen sowohl Konjunktion ($\wedge$) als auch Disjunktion ($\vee$) aufgenommen werden, und es muss ein Hilfsoperator $\bar{\mathbf{U}}$ eingeführt werden, um eine Negation an **U** vorbeiziehen zu können.

$$\phi\, \bar{\mathbf{U}}\, \psi \equiv \neg((\neg\phi)\, \mathbf{U}\, (\neg\psi))$$

Außerdem müssen alle Vorkommen von $\top$ und $\bot$ eliminiert werden. Für ein beliebiges $p \in AP$ kann $\top$ durch $p \vee \neg p$ und $\bot$ durch $p \wedge \neg p$ ersetzt werden.

Nachdem alle Vorkommen von **F**, **G**, $\top$ und $\bot$ in einer PLTL-Formel wie oben angegeben ersetzt wurden, kann diese mit Hilfe der folgenden Gleichungen in Normalform überführt werden:

$$\begin{aligned} \neg(\phi \vee \psi) &\equiv (\neg\phi) \wedge (\neg\psi) \\ \neg(\phi \wedge \psi) &\equiv (\neg\phi) \vee (\neg\psi) \\ \neg\mathbf{X}\phi &\equiv \mathbf{X}(\neg\phi) \\ \neg(\phi\, \mathbf{U}\, \psi) &\equiv (\neg\psi)\, \bar{\mathbf{U}}\, (\neg\phi) \\ \neg(\phi\, \bar{\mathbf{U}}\, \psi) &\equiv (\neg\psi)\, \mathbf{U}\, (\neg\phi) \end{aligned}$$

Liest man die obigen Gleichungen von links nach rechts, so stellt man fest, dass jeweils die *äußerste* Negation nach *innen* verschoben wird. Es ist offensichtlich, dass durch iteratives Anwenden der Regeln Negationen soweit wie möglich nach *innen* verschoben werden, d.h. am Ende stehen Negationen nur noch vor Propositionalzeichen.

Beispiel 2.2 *Als Beispiel betrachten wir hier die Überführung der PLTL-Formel $\phi = \mathbf{G}(p \wedge \neg\mathbf{X}q)$ mit $AP = \{p, q\}$ in Normalform.*

$$\begin{aligned} & \mathbf{G}(p \wedge \neg\mathbf{X}q) \\ \equiv\ & \neg\mathbf{F}\neg(p \wedge \neg\mathbf{X}q) \\ \equiv\ & \neg(\top\, \mathbf{U}\, \neg(p \wedge \neg\mathbf{X}q)) \\ \equiv\ & \bot\bar{\mathbf{U}}(p \wedge \neg\mathbf{X}q) \\ \equiv\ & (p \wedge \neg p)\bar{\mathbf{U}}(p \wedge \neg\mathbf{X}q) \\ \equiv\ & (p \wedge \neg p)\bar{\mathbf{U}}(p \wedge \mathbf{X}\neg q) \end{aligned}$$

$(p \wedge \neg q)\bar{\mathbf{U}}(p \wedge \mathbf{X}\neg q)$ *erfüllt die Normalform Bedingungen, da die Negation nur noch vor* p *und* q *steht und als temporale Operatoren nur noch* $\mathbf{X}$ *und* $\bar{\mathbf{U}}$ *vorkommen.*

Es ist offensichtlich, dass die Zeit, die für die Überführung einer PLTL-Formel ϕ in Normalform benötigt wird, linear von der Größe von ϕ abhängt. Die Größe von ϕ, im folgenden bezeichnet als $|\phi|$, ist dabei induktiv über den Aufbau von ϕ definiert.

Definition 2.8 (Größe von PLTL-Formeln) *Sei* $p \in AP$ *und* ϕ, ψ *gültige PLTL-Formeln. Die Funktion* $|\cdot| : \mathbb{F} \to \mathbb{N}$*, induktiv definiert durch*

$$\begin{aligned} |p| &= 1 \\ |\neg\phi| &= 1 + |\phi| \\ |\phi \vee \psi| &= 1 + |\phi| + |\psi| \\ |\mathbf{X}\phi| &= 1 + |\phi| \\ |\phi\, \mathbf{U}\, \psi| &= 1 + |\phi| + |\psi|, \end{aligned}$$

ordnet jeder PLTL-Formel $\phi \in \mathbb{F}$ *eine Größe zu.*

Der Algorithmus für die Überführung von ϕ in Normalform hat also eine *worst-case Laufzeit* von $\mathcal{O}(|\phi|)$.

Kapitel 3

Automatenmodelle

PLTL Model Checking basiert auf endlichen Automaten, die in der Lage sind unendliche Wörter[1] zu akzeptieren. Im folgenden werden deshalb zunächst die zugrundeliegenden Automatenmodelle eingeführt.

3.1 Automaten auf endlichen Wörtern

Dieser Abschnitt definiert ein Automatenmodell, welches in der Lage ist, endliche Wörter mit einer endlichen Menge von Zuständen zu akzeptieren. Der nächste Abschnitt erweitert dieses einfache Automatenmodell für unendliche Wörter.

Definition 3.1 (Labeled Finite State Automata) *Ein Labeled Finite State Automaton (kurz: LFSA) A ist definiert als Sixtupel $(\Sigma, S, S_0, \rho, F, l)$. Hierbei ist*

- Σ *das (nicht leere) Eingabealphabet,*
- S *der endliche Zustandsraum,*
- $\emptyset \subset S_0 \subseteq S$ *die (nicht leere) Menge von Startzuständen,*
- $\rho : S \to \mathcal{P}(S)$ *die Funktion, die jedem Zustand aus S eine Menge von möglichen Folgezuständen zuordnet,*

[1] In diesem Kapitel werden zunächst Wörter nur allgemein als Zeichenfolgen über dem Eingabealphabet behandelt. Das nächste Kapitel stellt den Zusammenhang zur PLTL her.

- $F \subseteq S$ *die Menge der akzeptierenden Endzustände,*
- $l : S \to \Sigma$ *die Funktion, die jedem Zustand aus* S *ein Zeichen aus* Σ *zuordnet.*

Für jeden Zustand $s \in S$ ist $\rho(s)$ die Menge der Zustände, in die der Automat A übergehen kann, wenn er sich im Zustand s befindet. ρ wird deshalb als die *Übergangsfunktion* des Automaten A bezeichnet. Wir schreiben $s \to s'$ gdw. $s' \in \rho(s)$.

Die Funktion l ordnet jedem Zustand $s \in S$ einen Markierung (engl. *label*) aus Σ zu, und wird als *labeling function* bezeichnet. Ferner bezeichne Σ^* die Menge der endlichen Wörter über Σ.

Definition 3.2 (Deterministischer LFSA) *Ein LFSA* A *ist genau dann* deterministisch*, wenn für alle Markierungen* $a \in \Sigma$ *gilt*

$$|\{s \in S_0 \mid l(s) = a\}| \leq 1$$

und für alle $a \in \Sigma$ *und* $s \in S$ *gilt*

$$|\{s' \in \rho(s) \mid l(s') = a\}| \leq 1.$$

Also ist A genau dann *deterministisch*, wenn die Anzahl gleichbezeichneter Startzustände höchstens eins ist, und für jede Markierung und jeden Zustand der Folgezustand eindeutig bestimmt ist.

Umgekehrt ist ein LFSA A genau dann *nicht-deterministisch*, wenn mehreren Startzuständen die gleiche Markierung zugeordnet wird oder die Übergangsfunktion für einen Zustand mehrere mit der gleichen Markierung versehene Folgezustände liefert.

Definition 3.3 (Lauf eines LFSA) *Für einen LFSA* A *ist ein* Lauf von A *definiert als eine endliche Folge von Zuständen* $\sigma = s_0 \dots s_n$ *mit* $s_0 \in S_0$ *und* $s_i \to s_{i+1}$ *für* $0 \leq i < n$. σ *heißt* akzeptierend *genau dann wenn* $s_n \in F$.

Definition 3.4 (Akzeptierte Sprache eines LFSA) *Ein LFSA* A *akzeptiert ein endliches Wort* $w = a_0 \dots a_n \in \Sigma^*$ *genau dann wenn ein akzeptierender Lauf* $\sigma = s_0 \dots s_n$ *existiert mit* $l(s_i) = a_i$ *für* $0 \leq i \leq n$. *Die von* A *akzeptierte Sprache* $\mathcal{L}(A) \subseteq \Sigma^*$ *ist*

$$\mathcal{L}(A) = \{w \in \Sigma^* \mid A \text{ akzeptiert } w\}.$$

Im speziellen Fall von $F = \emptyset$ existiert kein akzeptierender Lauf und es gilt $\mathcal{L}(A) = \emptyset$.

Beispiel 3.1 *Sei* $A = (\{a\}, \{s_0, s_1\}, \{s_0\}, \rho, \{s_1\}, l)$ *mit* $\rho(s_0) = \{s_1\}$, $\rho(s_1) = \{s_0\}$ *und* $l(s_0) = l(s_1) = a$ *der in Abbildung 3.1 dargestellte LFSA.*

A akzeptiert offensichtlich die Menge der endlichen Wörter über $\{a\}$*, deren Länge ein Vielfaches von* 2 *ist, also* $\mathcal{L}(A) = \{a^{2n} \mid n \in \mathbb{N}\}$.

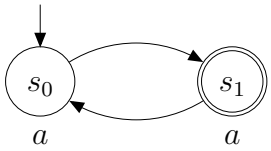


Abbildung 3.1: Ein einfacher LFSA

Definition 3.5 (Äquivalenz von Automaten) *Zwei LFSAs* A_1 *und* A_2 *heißen äquivalent genau dann wenn* $\mathcal{L}(A_1) = \mathcal{L}(A_2)$.

3.2 Automaten auf unendlichen Wörtern

Die Forderung, unendliche Wörter akzeptieren zu können, ist maßgeblich für das PLTL Model Checking, da reaktive Programme i.d.R. nicht terminieren, und somit keine endliche, akzeptierende Folge von Zuständen existieren kann. Allerdings erfüllen weder die einfachen, aus den Grundvorlesungen des Informatikstudiums bekannten Automatenmodelle, noch das Automatenmodell des LFSA, welches im vorherigen Abschnitt vorgestellt wurde, diese Forderung, da das Akzeptanzkriterium für Wörter stets über das Erreichen eines Endzustandes definiert ist.

3.2.1 Labeled Büchi Automata

Eine Klasse von Automaten, welche diese Forderung erfüllen, stellen die sogenannten *Labeled Büchi Automata (LBA)* dar. Ein Labeled Büchi Automaton ist eine Variante eines *Labeled Finite State Automaton (LFSA)* mit einem speziellen Akzeptanzkriterium für Wörter, dem *Büchi Kriterium.*

Definition 3.6 (Lauf eines LBA) *Für einen LBA A ist ein* Lauf von A *definiert als eine unendliche Folge von Zuständen $\sigma = s_0 s_1 \ldots$ mit $s_0 \in S_0$ und $s_i \to s_{i+1}$ für $i > 0$. Sei* $\lim(\sigma)$ *die Menge der Zustände, die durch σ unendlich oft besucht werden, so heißt σ* akzeptierend *genau dann wenn* $\lim(\sigma) \cap F \neq \emptyset$.

Bezeichne Σ^ω die Menge der unendlichen Wörter über Σ. Ein LBA A akzeptiert ein unendliches Wort $w = a_0 a_1 \ldots \in \Sigma^\omega$ genau dann wenn es einen akzeptierenden Lauf $\sigma = s_0 s_1 \ldots$ gibt mit $l(s_i) = a_i$ für $i > 0$. Die von dem Automaten A akzeptierte unendliche Sprache ist dann definiert durch:

$$\mathcal{L}_\omega(A) = \{w \in \Sigma^\omega \mid A \text{ akzeptiert } w\}$$

Im speziellen Fall von $F = \emptyset$ ist $\lim(\sigma) \cap \emptyset = \emptyset$ und es gilt $\mathcal{L}_\omega(A) = \emptyset$.

In Worten ausgedrückt besagt die obige Definition also, dass ein LBA A alle unendlichen Wörter $w \in \Sigma^\omega$ akzeptiert, für die ein Lauf σ existiert, in dem mindestens ein Endzustand aus F unendlich oft besucht wird.

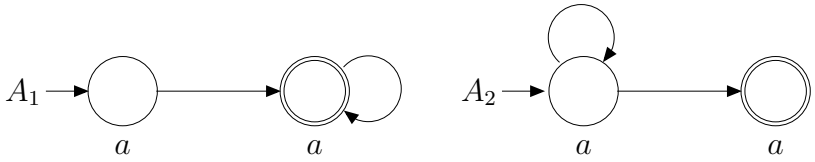
Die Äquivalenz von Büchi Automaten ist analog zur Äquivalenz von Automaten auf endlichen Wörtern im vorherigen Abschnitt definiert.

Definition 3.7 (Äquivalenz von Büchi Automaten) *Zwei LBAs A_1 und A_2 heißen äquivalent genau dann wenn $\mathcal{L}_\omega(A_1) = \mathcal{L}_\omega(A_2)$.*

Interessant ist hierbei insbesondere die Beziehung zwischen $\mathcal{L}(A)$, der Menge der endlichen Wörter, die von A akzeptiert wird, und $\mathcal{L}_\omega(A)$, der Menge der unendlichen von A akzeptierten Wörter. Denn zwei Automaten, die bezüglich $\mathcal{L}$ äquivalent sind, sind nicht zwangsläufig auch auf $\mathcal{L}_\omega$ äquivalent, wie im folgenden Beispiel zu sehen.

Beispiel 3.2 *Seien A_1 und A_2 zwei Automaten, und bezeichne $\mathcal{L}(A_i)$ die Menge der endlichen und $\mathcal{L}_\omega(A_i)$ die Menge der unendlichen von A_i akzeptierten Wörter für $i \in \{1, 2\}$.*

1. *Akzeptieren A_1 und A_2 die gleiche Menge endlicher Wörter, so kann die Menge der akzeptierten unendlichen Wörter dennoch unterschiedlich sein.*



Hier gilt $\mathcal{L}(A_1) = \mathcal{L}(A_2) = \{a^n \mid n \geq 2\}$*, jedoch* $\mathcal{L}_\omega(A_1) = \{a^\omega\} \neq \emptyset = \mathcal{L}_\omega(A_2)$.

2. *Ebensowenig folgt aus der Äquivalenz bzgl.* $\mathcal{L}_\omega$ *die Äquivalenz bzgl.* $\mathcal{L}$*, was nicht unbedingt direkt ersichtlich ist. Aber basierend auf dem Automaten aus Abbildung 3.1 läßt sich ein einfaches Beispiel konstruieren.*

Es gilt $\mathcal{L}_\omega(A_1) = \mathcal{L}_\omega(A_2) = \{a^\omega\}$*, aber* $\mathcal{L}(A_1) = \left\{a^{2(n+1)} \mid n \in \mathbb{N}\right\} \neq \{a^{2n+1} \mid n \in \mathbb{N}\} = \mathcal{L}(A_2)$.

3. *Sind allerdings* A_1 *und* A_2 *deterministisch, dann gilt* $\mathcal{L}(A_1) = \mathcal{L}(A_2) \Rightarrow \mathcal{L}_\omega(A_1) = \mathcal{L}_\omega(A_2)$*. Die Umkehrung gilt, wie im vorherigen Beispiel gezeigt, nicht.*

In diesem Zusammenhang ist noch ein weiterer Unterschied zwischen LFSAs und LBAs interessant, der die Ausdrucksstärke betrifft. Während deterministische und nicht deterministische LFSAs die gleiche Ausdrucksstärke besitzen, sind nicht deterministische LBAs echt ausdrucksstärker als deterministische LBAs.

Neben dem Büchi Kriterium gibt es noch weitere Akzeptanzkriterien für Automaten auf unendlichen Wörtern, auf die hier allerdings – mit Ausnahme des *Generalized LBA* – nicht eingegangen werden soll (vgl. [Kat99, S. 73]).

3.2.2 Generalized Labeled Büchi Automata

Eine Verallgemeinerung des Büchi Automatenmodells stellt der sogenannte *Generalized LBA* dar, der als Zwischenschritt des PLTL Model Checking Algorithmus auftritt, und wie folgt definiert ist.

Definition 3.8 (Generalized LBA) *Ein Generalized Label Büchi Automaton (kurz: GLBA)* A *ist definiert als Sixtupel* $(\Sigma, S, S_0, \rho, \mathcal{F}, l)$*, wobei* Σ*,* S*,* S_0*,* ρ *und* l *die gleiche Bedeutung wie im Falle des LBA zukommt, und* $\mathcal{F}$ *eine Menge von akzeptierenden Zustandsmengen* $\{F_1, \ldots, F_k\}$ *ist, mit* $k \geq 0$ *und* $F_i \subseteq S$ *für* $i = 1, \ldots, k$.

Statt einer einzigen Menge von akzeptierenden Endzuständen gibt es nun eine Menge von Mengen akzeptierender Endzustände $\mathcal{F} \subseteq \mathcal{P}(S)$.

Definition 3.9 (Lauf eines GLBA) *Für einen GLBA A heißt ein Lauf $\sigma = s_0 s_1 \ldots$ akzeptierend genau dann wenn*

$$\forall i : [(0 < i \leq k) \Rightarrow \lim(\sigma) \cap F_i \neq \emptyset] ,$$

also für jede Menge von akzeptierenden Endzuständen $F_i \in \mathcal{F}$ ein Zustand in F_i existiert, der unendlich oft in σ auftaucht.

Der Fall $\mathcal{F} = \emptyset$ bedeutet also hierbei, dass alle Läufe beliebig oft alle akzeptierenden Endzustände besuchen, folglich also jeder Lauf akzeptierend ist.

3.2.3 Äquivalenz der Modelle

Man sieht leicht, dass zu jedem LBA ein äquivalenter GLBA angegeben werden kann, indem $\mathcal{F} = \{F\}$ gewählt wird.

Wohl interessanter ist, dass man umgekehrt auch zu jedem GLBA einen äquivalenten LBA konstruieren kann. Die grundsätzliche Idee bei der Umwandlung eines GLBA A mit $\mathcal{F} = \{F_1, \ldots, F_k\}$ in einen LBA A' ist, k Kopien von A zu machen, eine für jede Menge $F_i \in \mathcal{F}$, und Zustände als Paare (s, i) anzugeben mit $0 < i \leq k$.

Definition 3.10 (Umwandlung eines GLBAs in einen LBA) *Sei $A =$*

$(\Sigma, S, S_0, \rho, \mathcal{F}, l)$ ein GLBA mit $\mathcal{F} = \{F_1, \ldots, F_k\}$. Dann ist $A' = (\Sigma, S', S'_0, \rho', F', l')$ mit

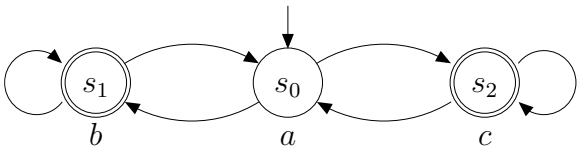
- $S' = S \times \{i \mid 0 < i \leq k\}$
- $S'_0 = S_0 \times \{i\}$ *für ein beliebiges i mit $0 < i \leq k$*
- $(s, i) \rightarrow' (s', i)$ *gdw. $s \rightarrow s'$ und $s \notin F_i$*
  $(s, i) \rightarrow' (s', (i \bmod k) + 1)$ *gdw. $s \rightarrow s'$ und $s \in F_i$*
- $F' = F_i \times \{i\}$ *für ein beliebiges i mit $0 < i \leq k$*
- $l'(s, i) = l(s)$

ein äquivalenter LBA, es gilt also $\mathcal{L}_\omega(A) = \mathcal{L}_\omega(A')$.

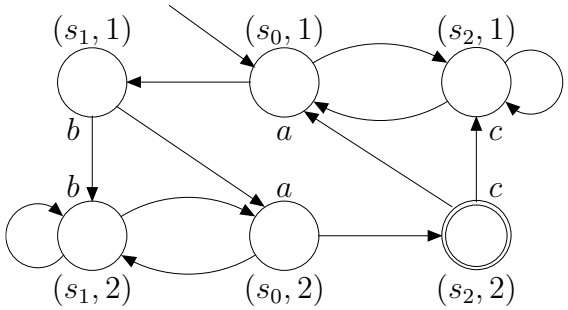
Es gilt zu beachten, dass für die Definition der Anfangs- und Endzustände von A' ein beliebiges i gewählt werden kann, und somit der Automat A' nicht eindeutig bestimmt ist.

Gemäß der obigen Definition ist ein Lauf von A' akzeptierend, wenn die Zustände (s, i) mit $s \in F_i$ unendlich oft durchlaufen werden. Sobald ein Lauf einen solchen Zustand (s, i) erreicht, geht der Automat A' zur $(i+1)$-ten Kopie über. Von dieser Kopie kann die nächste Kopie über den Zustand $(s', i+1)$ mit $s' \in F_{i+1}$ erreicht werden, usw. A' kann zu (s, i) nur zurückkehren nachdem vorher alle k Kopien durchlaufen wurden, und für jede Kopie ein akzeptierender Endzustand erreicht wurde. Damit ein Lauf einen Zustand (s, i) unendlich oft durchlaufen kann, muss also aus jeder Kopie ein beliebiger akzeptierender Endzustand beliebig oft durchlaufen werden. Hiermit ist klar, dass A und A' die gleiche Sprache akzeptieren, folglich äquivalent sind.

Beispiel 3.3 *Sei A der im folgenden dargestellte GLBA*



mit zwei Mengen von akzeptierenden Endzuständen $F_1 = \{s_1\}$ und $F_2 = \{s_2\}$. Die Zustandsmenge des äquivalenten LBAs A' ist $S = \{s_0, s_1, s_2\} \times \{1, 2\}$. Eine mögliche Konstruktion für A' ist



wobei $(s_0, 1)$ als Anfangszustand und $(s_2, 2)$ als akzeptierender Endzustand gewählt wurde. Jeder akzeptierende Lauf muss also $(s_2, 2)$ unendlich oft durchlaufen ($s_2 \in F_2$). Dazu muss allerdings auch ein mit s_1 markierter Zustand ($s_1 \in F_1$) unendlich oft durchlaufen werden.

Die Anzahl der Zustände eines so konstruierten einfachen LBAs ist $\mathcal{O}(k \times |S|)$, wobei S die Zustandsmenge des GLBA und k die Anzahl der akzeptierenden Endzustandsmengen ist.

Kapitel 4

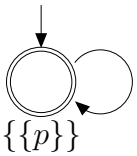
Automaten für PLTL-Formeln

Nachdem nun in den vorangegangenen Kapiteln die PLTL und Automatenmodelle für unendliche Wörter eingeführt wurden, soll in diesem Kapitel der Zusammenhang zwischen PLTL-Formeln und Büchi Automaten dargestellt werden. Anschließend wird ein Algorithmus zur Konstruktion von Büchi Automaten für PLTL-Formeln vorgestellt.

Angenommen die Zustände eines LBA seien statt mit einzelnen Symbolen aus Σ mit Mengen von Symbolen markiert, zum Beispiel $l : S \to \mathcal{P}(\Sigma)$ für ein beliebiges Alphabet Σ. In diesem Fall wird ein Wort $w = a_0 a_1 \ldots$ genau dann akzeptiert, wenn ein akzeptierender Lauf $\sigma = s_0 s_1 \ldots$ existiert mit $a_i \in l(s_i)$ für alle $i \geq 0$.

Sei nun $\Sigma = \mathcal{P}(AP)$, wobei AP die Menge der Propositionalzeichen ist. Dann werden die Zustände mit Mengen von Mengen von Propositionalzeichen markiert, und Wörter bestehen aus einer Folge von Zeichen, wobei jedes Zeichen eine Menge von Propositionalzeichen ist. Ein LBA für eine PLTL-Formel ϕ akzeptiert also alle unendlichen Wörter, d.h. Folgen von Mengen von Propositionalzeichen, die ϕ erfüllen. Dies soll in den folgenden Beispielen erläutert werden (vgl. [Kat99, S. 73f]).

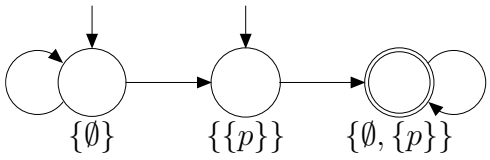
Beispiel 4.1 *Sei $AP = \{p\}$ und A der im folgenden dargestellte LBA.*



Jeder akzeptierende Lauf von A durchläuft den mit $\{\{p\}\}$ bezeichneten Zustand unendlich oft. Dementsprechend ist die von A akzeptierte unendliche

Sprache $\mathcal{L}_\omega(A) = (\{\{p\}\})^\omega$, kurz p^ω. Aus dieser Beobachtung folgt, dass die akzeptierenden Läufe von A exakt der Folge von Propositionalzeichen entsprechen, für die die PLTL-Formel $\mathbf{G}p$ gilt.

Beispiel 4.2 *Sei nun wiederum $AP = \{p\}$, und A' der folgende LBA:*



Dieser Automat kann entweder in einem Zustand starten, der p erfüllt (dem mit $\{\{p\}\}$ markierten Zustand), oder in einem Zustand, der p nicht erfüllt (dem mit $\{\emptyset\}$ markierten Zustand). Jeder akzeptierende Lauf muss den mit $\{\{p\}\}$ markierten Zustand einmalig durchlaufen und anschliessend unendlich oft den rechten, mit $\{\emptyset, \{p\}\}$ markierten Zustand, durchlaufen. Vereinfacht ausgedrückt akzeptiert A' also jeden Lauf, für den mindestens einmal p gilt. Dies entspricht der PLTL-Formel $\mathbf{F}p$.

Aus diesen Beispielen ergibt sich, dass $\{\emptyset\}$ für die Formel $\neg p$, $\{\{p\}\}$ für die Formel p und $\{\emptyset, \{p\}\}$ für die Formel $\neg p \vee p = \top$ steht. Allgemein stehen Mengen von Mengen von Propositionalzeichen also für aussagenlogische Formeln. Genauer gesagt, Mengen von Teilmengen einer gegebenen Menge AP von Propositionalzeichen dienen zur Kodierung von aussagenlogischen Formeln über AP.

Formal ausgedrückt, seien $AP_1, \ldots, AP_n \subseteq AP$, dann steht die Menge AP_i mit $0 < i \leq n$ für die Formel

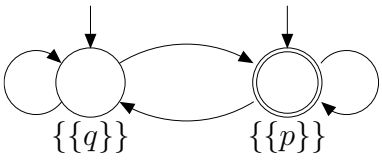
$$(\forall p \in AP_i : p) \wedge (\forall p \in AP \setminus AP_i : \neg p),$$

geschrieben als $[\![AP_i]\!]$. Die Menge $\{AP_{k_1}, \ldots, AP_{k_m}\}$ mit $m \geq 1$ und $0 < k_j \leq n$ kodiert nun die aussagenlogische Formel

$$[\![AP_{k_1}]\!] \vee \ldots \vee [\![AP_{k_m}]\!].$$

Man beachte, dass die Mengen von Mengen von Propositionalzeichen nicht leer sein dürfen.

Beispiel 4.3 *Betrachten wir nun den folgenden Automaten mit $AP = \{p, q\}$.*



Basierend auf den vorhergehenden Erkenntnissen können wir die Markierung des linken Zustands als $\neg p \wedge q$ und die Markierung des rechten Zustands als $p \wedge \neg q$ interpretieren. Jeder akzeptierende Lauf des Automaten muss folglich den Zustand, der $p \wedge \neg q$ erfüllt, unendlich oft durchlaufen, wobei jeweils zwischen zwei aufeinanderfolgenden Besuchen des zweiten Zustands, der erste Zustand, welcher $\neg p \wedge q$ erfüllt, beliebig oft durchlaufen werden kann. Folglich entsprechen die akzeptierenden Läufe des Automaten exakt den Folgen von Mengen von Propositionalzeichen, die die PLTL-Formel

$$\mathbf{G}\,[(\neg p \wedge q)\ \mathbf{U}\ (p \wedge \neg q)]$$

erfüllen.

Es liegt nun die Vermutung nahe, dass sich zu jeder PLTL-Formel (über einer gegebenen Menge von Propositionalzeichen AP) ein Büchi Automat konstruieren lässt.

Satz 4.1 *Zu jeder PLTL-Formel ϕ kann ein Büchi Automat A über dem Alphabet $\Sigma = \mathcal{P}(AP)$ konstruiert werden, so dass die akzeptierte Sprache $\mathcal{L}_\omega(A)$ den Folgen von Mengen von Propositionalzeichen entspricht, die ϕ erfüllen.*

Ein Beweis findet sich in [WVS83].

Basierend auf diesem Ergebnis können wir nun ein Verfahren zum PLTL Model Checking formulieren. Der einfachste, aber auch naivste Ansatz ist in Tabelle 4.1 dargestellt.

1. Konstruktion des Büchi Automaten für ϕ, A_ϕ
2. Konstruktion des Büchi Automaten für die Implementierung, A_{sys}
3. Prüfung ob $\mathcal{L}_\omega(A_{sys}) \subseteq \mathcal{L}_\omega(A_\phi)$

Tabelle 4.1: Naiver PLTL Model Checking Algorithmus

Hierbei werden zunächst Büchi Automaten für die Spezifikation (notiert als PLTL-Formel) und die Implementierung des Systems erstellt, und anschließend geprüft, ob sämtliche Verhaltenweisen der Implementierung A_{sys} erwünscht sind, also durch die Spezifikation A_ϕ erlaubt sind. Dieser Ansatz erscheint zunächst einmal einfach genug für eine Realisierung. Allerdings ist das Problem zu entscheiden, ob die Sprache des Automaten A_{sys} in der Sprache des Automaten A_ϕ enthalten ist, also $\mathcal{L}_\omega(A_{sys}) \subseteq \mathcal{L}_\omega(A_\phi)$, PSPACE-vollständig, folglich handelt es sich um ein schwieriges NP-Problem.

Mittels einer einfachen Umformung läßt sich die Mengeninklusion vermeiden,

$$\mathcal{L}_\omega(A_{sys}) \subseteq \mathcal{L}_\omega(A_\phi) \Leftrightarrow \mathcal{L}_\omega(A_{sys}) \cap \mathcal{L}_\omega(\bar{A}_\phi) = \emptyset$$

so dass stattdessen lediglich die Schnittmenge auf Leerheit geprüft werden muss. Hierbei steht $\bar{A}_\phi$ für das Komplement von A_ϕ, also den Automaten, der die Sprache $\Sigma^\omega \backslash \mathcal{L}_\omega(A)$ akzeptiert. Allerdings ist $\bar{A}_\phi$ nach der Konstruktion sehr groß[1]: Hat zum Beispiel A_ϕ n Zustände, so hat $\bar{A}_\phi$ c^{n^2} Zustände, für ein $c > 1$.

Das Komplement des Automaten A_ϕ ist aber gerade äquivalent zu dem Automaten $A_{\neg\phi}$, das heisst es gilt $\mathcal{L}_\omega(\bar{A}_\phi) = \mathcal{L}_\omega(A_{\neg\phi})$. Diese Erkenntnis ermöglicht es uns einen effizienten Algorithmus für das PLTL Model Checking anzugeben, wie in Tabelle 4.2 gezeigt.

1. Konstruktion des Büchi Automaten für $\neg\phi$, $A_{\neg\phi}$
2. Konstruktion des Büchi Automaten für die Implementierung, A_{sys}
3. Prüfung ob $\mathcal{L}_\omega(A_{sys}) \cap \mathcal{L}_\omega(A_{\neg\phi}) = \emptyset$

Tabelle 4.2: Effizienter PLTL Model Checking Algorithmus

Die zugrundeliegende Idee hierbei entspricht der Überlegung, einen Automaten $A_{\neg\phi}$ zu konstruieren, der genau das unerwünschte Verhalten repräsentiert. Anschließend wird geprüft, ob die Implementierung A_{sys} keine dieser unerwünschten Verhaltensweisen beinhaltet, also ob die Schnittmenge der beiden Automaten leer ist. Ist dies der Fall, entspricht die Implementierung der Spezifikation.

[1] Für deterministische Büchi Automaten existiert ein Algorithmus, der polynomiell in der Größe des Automaten ist, allerdings sind deterministische Büchi Automaten auch weitaus weniger mächtig als nicht-deterministische.

Oder anders ausgedrückt, hat A_{sys} einen akzeptierenden Lauf, der zugleich auch ein akzeptierender Lauf von $A_{\neg\phi}$ ist, so ist dies ein Beispiel für einen Lauf, der ϕ widerspricht. Existiert kein solcher Lauf so ist die Spezifikation ϕ erfüllt. In [EL85] findet sich ein Beweis, dass dieser dritte Schritt in linearer Zeit entscheidbar ist.

Kapitel 5

Von PLTL-Formeln zu Büchi Automaten

Das Ziel ist es einen Büchi Automaten zu konstruieren, der in der Lage ist alle unendlichen Zustandsfolgen zu generieren, die eine gegebene PLTL-Formel ϕ erfüllen. Der hier beschriebene Algorithmus ist in Abbildung 5.1 schematisch dargestellt und entspricht dem in [GPVW95] und [Kat99] angegebenen Algorithmus.

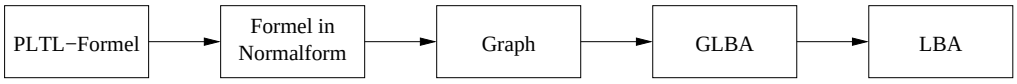


Abbildung 5.1: Übersicht über den PLTL zu LBA Algorithmus

Die Umwandlung von PLTL-Formeln in Normalform wurde bereits in Abschnitt 2.3 behandelt. In den nachfolgenden Abschnitten wird jeweils angenommen, dass PLTL-Formeln bereits in Normalform vorliegen.

5.1 Graphkonstruktion

Der nächste Schritt des Algorithmus ist also die Konstruktion eines Graphen für eine gegebene PLTL-Formel ϕ. Der Quelltext der Funktion *CreateGraph* wird, um unabhängig von einer speziellen Programmiersprache zu bleiben, in einer abstrakten Notation angegeben[1], die auf Dijkstra's Guarded Command Language basiert (vgl. [Dij75]).

[1] Anhang A enthält eine detailierte Beschreibung der Notation.

Ein Graph $\mathcal{G}$ ist ein Paar (V, E) mit einer Menge von Knoten V und einer Menge von Kanten $E \subseteq V \times V$, wobei Knoten wie folgt definiert sind.

Definition 5.1 (Knoten) *Ein Knoten v ist ein Quadrupel (P, N, O, Sc), wobei*

- $P \in 2^{V \cup \{init\}}$ *die Menge der Vorgänger ist, und*
- N, O, Sc *sind Mengen von PLTL-Formeln (in Normalform).*

Für $v = (P, N, O, Sc)$ schreiben wir $P(v) = P$, $N(v) = N$, $O(v) = O$ und $Sc(v) = Sc$.

P (Predecessors) ist die Menge der Vorgänger eines Knotens, und definiert damit die Struktur des Graphen. $N(v) \cup O(v)$ sind die für v zu prüfenden Formeln, wobei N (New) die Menge der Formeln repräsentiert, die noch nicht bearbeitet wurden, und O (Old) die bereits bearbeiteten. Sc (Successors) beinhaltet diejenigen Formeln, die für alle Knoten gelten müssen, die direkte Nachfolger von Knoten sind, welche die Formeln in O erfüllen.

init ist ein spezieller Bezeichner, der keinem realen Knoten in $\mathcal{G}$ entspricht, d.h. $init \notin V$, sondern als Markierung für Startknoten im Graphen dient. Ein Knoten v wird als Startknoten bezeichnet gdw. $init \in P(v)$.

Der in Listing 5.1 abgedruckte Algorithmus zur Graphkonstruktion und die folgenden Erläuterungen zum Algorithmus sind im wesentlichen identisch zu dem in [Kat99] enthaltenen Algorithmus, mit Ausnahme kleinerer Korrekturen und stilistischer Anpassungen.

Der Algorithmus konstruiert den Graphen $\mathcal{G}_\phi$ in *depth-first traversal order*, beginnend bei dem mit der Eingabeformel ϕ markierten Knoten. Die Liste S, die in diesem Fall als Stack implementiert werden kann, enthält alle Knoten, die bisher noch nicht vollständig erforscht worden sind. Zu Beginn enthält S nur den mit ϕ markierten Knoten *init*. Die Menge Z enthält die bereits erzeugten Knoten für den Graphen. Der Algorithmus terminiert sobald alle Knoten vollständig erforscht sind ($S = \langle\rangle$), und Z enthält anschließend den fertig konstruierten Grahpen. Ein Knoten v gilt als vollständig erforscht wenn alle Formeln, die in v gelten müssen, geprüft worden sind ($N(v) = \emptyset$).

Sei $N(v) = \emptyset$ für einen Knoten v. Dann wird zunächst geprüft, ob Z bereits einen Knoten mit identischem O und Sc enthält. Falls ja wird kein neuer Knoten erstellt. Stattdessen wird vermerkt, dass der existierende Knoten nun zusätzlich über alle Vorgänger von v erreichbar ist. Existiert noch kein solcher Knoten, wird v zu Z hinzugefügt und die Nachfolger von v müssen

```
function CreateGraph (φ: Formula): set of Vertex;
(* Vorbedingung: φ ist PLTL–Formel in Normalform *)
begin var S: sequence of Vertex,
          Z: set of Vertex, (* bereits erforschte Knoten *)
          w1, w2: Vertex,
          ψ: Formula;
  S, Z := ⟨({init}, {φ}, ∅, ∅)⟩, ∅;
  do S = ⟨v⟩⌢S' ⟶
    (* v ist erster Knoten, S' ist Rest *)
    if N(v) = ∅ ⟶ (* alle Formeln für v geprüft *)
      if (∃w ∈ Z : Sc(v) = Sc(w) ∧ O(v) = O(w)) ⟶
        P(w), S := P(w) ∪ P(v), S' (* w ist Kopie von v *)
      [] ¬(∃w ∈ Z : ...) ⟶
        S, Z := ⟨({v}, Sc(v), ∅, ∅)⟩⌢S', Z ∪ {v}
      fi
    [] N(v) ≠ ∅ ⟶ (* noch Formeln für v übrig *)
      let ψ in N(v);
      N(v) := N(v) \ {ψ};
      if ψ ∈ AP ∨ (¬ψ) ∈ AP ⟶
        if (¬ψ) ∈ O(v) ⟶ S := S'
        [] (¬ψ) ∉ O(v) ⟶ skip
        fi
      [] ψ = (ψ1 ∧ ψ2) ⟶ N(v) := N(v) ∪ ({ψ1, ψ2} \ O(v))
      [] ψ = X φ ⟶ Sc(v) := Sc(v) ∪ {φ}
      [] ψ ∈ {ψ1 U ψ2, ψ1 Ū ψ2, ψ1 ∨ ψ2} ⟶
        (* ψ aufspalten *)
        w1, w2 := v, v;
        N(w1) := N(w1) ∪ (F1(ψ) \ O(w1));
        N(w2) := N(w2) ∪ (F2(ψ) \ O(w2));
        O(w1), O(w2) := O(w1) ∪ {ψ}, O(w2) ∪ {ψ};
        S := ⟨w1⟩⌢(⟨w2⟩⌢S')
      fi;
      O(v) := O(v) ∪ {ψ}
    fi
  od;
  return Z;
end
(* Nachbedingung: Z ist Menge der Knoten für den Graph *)
(* Gφ wobei die Startknoten durch init ∈ P und die Kanten *)
(* durch die P–Komponenten der Knoten bestimmt sind *)
```

Listing 5.1: Algorithmus zur Konstruktion eines Graphen für ϕ

erforscht werden. Hierzu wird ein neuer mit $Sc(v)$ markierter Knoten zu S hinzugefügt. Anschließend wird v aus S entfernt, da v ja bereits vollständig erforscht ist.

Anderenfalls, wenn $N(v) \neq \emptyset$, gibt es noch Formeln für v, die geprüft werden müssen. In jedem Schritt wird nun eine Formel ψ aus $N(v)$ entfernt, und nach folgenden Fällen unterschieden bearbeitet.

- Wenn ψ die Negation eines Aussagezeichens ist, welches bereits zuvor bearbeitet worden ist, haben wir einen Widerspruch gefunden, und der Knoten v wird nicht weiter erforscht ($S := S'$).

- Falls ψ ein Aussagezeichen ist, dessen Negation noch nicht bearbeitet worden ist, ist nichts zu tun.

- Ist $\psi = \psi_1 \wedge \psi_2$, dann müssen, damit ψ erfüllt wird, sowohl ψ_1 als auch ψ_2 erfüllt sein. D.h. ψ_1 und ψ_2 werden, sofern sie nicht bereits geprüft worden sind, also $\psi_1 \notin O(v)$ bzw. $\psi_2 \notin O(v)$, zu $N(v)$ hinzugenommen.

- Für $\psi = \mathbf{X}\varphi$ muss für alle direkten Nachfolger φ erfüllt sein, folglich wird φ zu $Sc(v)$ hinzugefügt.

- Bleiben noch die Fälle Disjunktion, $\mathbf{U}$-Formel und $\bar{\mathbf{U}}$-Formel zu betrachten. Hierbei wird der Knoten v jeweils aufgespalten in zwei Knoten w_1 und w_2. Die beiden Knoten entsprechen den beiden Möglichkeiten die Formel ψ zu erfüllen, jeweils in Abhängigkeit von der Struktur von ψ. Grundsätzlich wird ψ dazu in eine Disjunktion umgeformt, so dass beide Teilformeln einzeln geprüft werden können.

  - $\psi = \psi_1 \vee \psi_2$. Um ψ zu erfüllen genügt es, dass entweder ψ_1 erfüllt wird oder ψ_2. Mit $F_i(\psi) = \{\psi_i\}$ für $i = 1, 2$ reduziert das die Prüfung auf ψ_1 (in w_1) oder ψ_2 (in w_2).
  - $\psi = \psi_1 \mathbf{U} \psi_2$ läßt sich umschreiben in disjunktiver Form (vgl. [HR04, S. 184ff]) als $\psi_2 \vee [\psi_1 \wedge \mathbf{X}(\psi_1 \mathbf{U} \psi_2)]$.
  - $\psi = \psi_1 \bar{\mathbf{U}} \psi_2$ läßt sich ebenfalls umschreiben in disjunktiver Form als $(\psi_2 \wedge \psi_1) \vee \left[\psi_2 \wedge \mathbf{X}\left(\psi_1 \bar{\mathbf{U}} \psi_2\right)\right]$.

Beispiel 5.1 *Abbildung 5.2 zeigt das Ergebnis der Anwendung von CreateGraph auf die PLTL-Formel p $\mathbf{U}$ q mit $p, q \in AP$.*

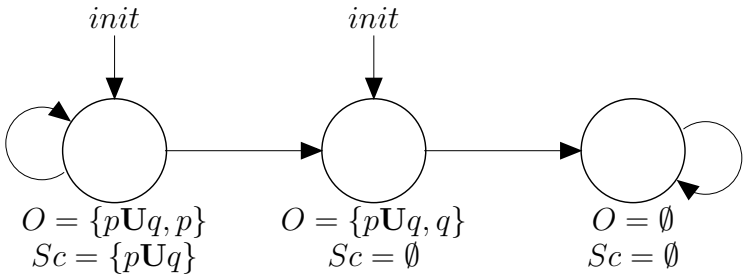


Abbildung 5.2: Ergebnis von $CreateGraph(p\ \mathbf{U}\ q)$

Abschließend soll hier noch eine kurze Analyse der Laufzeitkomplexität der *CreateGraph*-Funktion gegeben werden. Prinzipiell kann jede Menge von Teilformeln von ϕ zu einem Knoten im Graphen $\mathcal{G}_\phi$ werden, so dass die Anzahl der Knoten schlimmstenfalls proportional zur Anzahl der Teilmengen von Formeln von ϕ ist. Die Anzahl der Teilformeln einer Formel ist aber gerade gleich der Länge der Formel, so dass die Anzahl der Knoten schlimmstenfalls gleich $2^{|\phi|}$ ist. Es ergibt sich folglich eine Laufzeitkomplexität von $\mathcal{O}(2^{|\phi|})$ für die Konstruktion des Graphen.

5.2 Konstruktion des GLBA

Der dritte Schritt des Model Checking Algorithmus ist die Konstruktion eines *generalized* LBA für den zuvor konstruierten Graphen. Die folgende Definition fasst beide Schritte zusammen.

Definition 5.2 (Konstruktion eines GLBA für eine PLTL-Formel) *Sei ϕ eine PLTL-Formel in Normalform. Dann ist der zugehörige GLBA $A = (\Sigma, S, S_0, \rho, \mathcal{F}, l)$ definiert durch*

- $\Sigma = 2^{AP}$
- $S = CreateGraph(\phi)$
- $S_0 = \{s \in S \mid init \in P(s)\}$
- $s \to s' \Leftrightarrow s \in P(s') \wedge s \neq init$
- $\mathcal{F} = \{\{s \in S \mid \phi_1\mathbf{U}\phi_2 \notin O(s) \ \vee \ \phi_2 \in O(s)\} \mid \phi_1\mathbf{U}\phi_2 \in Sub(\phi)\}$
- $l(s) = \{\mathcal{P} \subseteq AP \mid Pos(s) \subseteq \mathcal{P} \ \wedge \ \mathcal{P} \cap Neg(s) = \emptyset\}$,

wobei $Sub(\phi)$ die Menge aller Teilformeln von ϕ darstellt, $Pos(s) = O(s) \cap AP$ die Menge der Aussagezeichen, die erfüllt sind in s, und $Neg(s) = \{p \in AP \mid \neg p \in O(s)\}$ die Menge der negierten Aussagezeichen, die erfüllt sind in s.

Für die Zustandsmenge des Automaten A_ϕ wird die Menge der Knoten des durch *CreateGraph* erzeugten Graphen $\mathcal{G}_\phi$ gewählt. Die Startzustände entsprechen den Startknoten, also den Knoten v für die $init \in P(v)$ gilt. Ein Übergang von einem Zustand s in einen Zustand s' ist genau dann möglich, wenn s ein Vorgänger von s' ist und s nicht der spezielle Knoten *init* ist. Für jede Teilformel der Form $\phi_1 \mathbf{U} \phi_2$ in ϕ wird eine Menge von akzeptierenden Endzuständen erzeugt, die alle Zustände s enthält, für die entweder $\phi_2 \in O(s)$ oder $\phi_1 \mathbf{U} \phi_2 \notin O(s)$ gilt. Die Anzahl der akzeptierenden Endzustandsmenge von A_ϕ ist daher gleich der Anzahl der **U**-Teilformeln in ϕ. Ein Zustand s ist markiert mit allen Mengen von Aussagezeichen, die die gültigen Aussagen in s enthalten und keine der nicht gültigen Aussagen.

Satz 5.1 *Ein gemäß Listing 5.1 und Definition 5.2 für eine PLTL-Formel ϕ in Normalform konstruierter GLBA A_ϕ akzeptiert genau diejenigen Folgen über $\left(2^{AP}\right)^\omega$, die die Formel ϕ erfüllen.*

Ein Beweis dieses Satzes findet sich in [GPVW95].

In [GPVW95] findet sich darüber hinaus eine Aufstellung über die Größenordnung einiger mit dem obigen Verfahren konstruierter Automaten für bestimmte PLTL-Formeln, die exemplarisch in Tabelle 5.1 wiedergegeben ist.

PLTL-Formel	Zustände (S)	Übergänge (ρ)	Endzustandsm. ($\mathcal{F}$)
$\phi_1 \mathbf{U} \phi_2$	3	4	1
$\phi_1 \mathbf{U} (\phi_2 \mathbf{U} \phi_3)$	4	6	2
$\neg(\phi_1 \mathbf{U} (\phi_2 \mathbf{U} \phi_3))$	7	15	0
$\mathbf{GF}\phi_1 \rightarrow \mathbf{GF}\phi_2$	9	15	2
$\mathbf{F}\phi_1 \mathbf{U} \mathbf{G}\phi_2$	8	15	2
$\mathbf{G}\phi_1 \mathbf{U} \phi_2$	5	6	1
$\neg(\mathbf{FF}\phi_1 \leftrightarrow \mathbf{F}\phi_2)$	22	41	2

Tabelle 5.1: Größenordnungen von GLBAs für PLTL-Formeln

Interessant zu beobachten ist hierbei, dass für die Formel $\neg(\phi_1 \mathbf{U} (\phi_2 \mathbf{U} \phi_3))$ keine Endzustände erzeugt werden, was darauf zurückzuführen ist, dass die Normalform-Darstellung keine **U**-Formeln mehr enthält[2]. Folglich ist jeder Lauf dieses Automaten akzeptierend.

[2]Dies zu überprüfen bleibt als Übung für den Leser.

Kapitel 6

Abschliessende Betrachtungen

Auf den dritten und letzten Schritt des PLTL Model Checking Algorithmus, also auf die Prüfung ob

$$\mathcal{L}_\omega(A_{sys}) \cap \mathcal{L}_\omega(A_{\neg\phi}) = \emptyset$$

gilt, wird im Rahmen dieses Dokuments nicht weiter eingegangen, stattdessen wird auf [Jun06] und [Kat99] verwiesen, wo ein Algorithmus zur Lösung des *checking for emptiness*-Problems beschrieben wird.

Es sei noch erwähnt, dass die Komplexität des hier beschriebenen Verfahrens zur Konstruktion eines Graphen für die Spezifikation ϕ proportional zur Anzahl der Teilformeln von ϕ ist, also $\mathcal{O}(2^{|\phi|})$, da für jede Teilformel von ϕ ein Knoten im Graph erzeugt wird. Die weiteren Schritte der Transformation einer PLTL-Formel ϕ in einen LBA A_ϕ beinflussen diese worst-case Komplexität nicht.

Insgesamt ergibt sich für die Überprüfung ob ein System *sys* einer Spezifikation ϕ genügt, also für den vollständigen PLTL Model Checking Algorithmus, nach [Kat99] eine Komplexität von $\mathcal{O}(|S_{sys}|^2 \times 2^{|\phi|})$, wobei S_{sys} die Zustände des Automaten A_{sys} bezeichnet.

Anhang A

Guarded Command Language

Der in diesem Dokument beschriebene Algorithmus zur Konstruktion von Graphen für PLTL-Formeln wird in einer abstrakten Programmnotation angegeben, um unhabhängig von einer speziellen Programmiersprache zu sein. Diese Notation basiert auf Dijkstra's Guarded Command Language (vgl. [Dij75]).

Syntax

Ein Programm ist eine Folge von Anweisungen, die durch Semikolon getrennt werden, und ihrerseits aus Anweisungen bestehen können.

$$
\begin{array}{rcl}
\textit{statement} & ::= & \textbf{skip} \\
& | & \textbf{let } \textit{variable} \textbf{ in } \textit{set} \\
& | & \textit{variable} \ := \ \textit{expression} \\
& | & \textit{variable}, \textit{variable} \ := \ \textit{expression}, \textit{expression} \\
& | & \textbf{if } \textit{guarded command} \ [] \ldots [] \ \textit{guarded command} \textbf{ fi} \\
& | & \textbf{do } \textit{guarded command} \ [] \ldots [] \ \textit{guarded command} \textbf{ od} \\
& | & \textit{statement} \ ; \ \textit{statement}
\end{array}
$$

$$
\textit{guarded command} \ ::= \ \textit{boolean expression} \longrightarrow \textit{statement}
$$

Kommentare werden wie in Pascal oder O'Caml von (* und *) umschlossen. Die Syntax von Ausdrücken und Bedingungen wird nicht näher festgelegt, stützt sich aber im wesentlichen auf die in der Mathematik benutzte Schreibweise.

Semantik

Die Semantik entspricht größtenteils der intuitiven Bedeutung der Anweisungen im Kontext einer imperativen Programmiersprache, allerdings sind bedingte Anweisungen nicht deterministisch.

- **skip** steht für die leere Anweisung und bedeutet einfach „nichts tun“.
- Die Anweisung **let** x **in** V bedeutet, dass auf nicht deterministische Weise ein beliebiges Element aus V ausgewählt und an x zugewiesen werden soll. Hierbei muss V eine nicht leere Menge sein.
- Die einfache Zuweisung $x := e$ weist der Variablen x den Wert des Ausdrucks e zu.
- $x_1, x_2 := e_1, e_2$ steht für die *gleichzeitige* Zuweisung des Wertes von e_1 an x_1 und e_2 an x_2. Insbesondere kann e_2 die Variable x_1 enthalten und die Auswertung wird auf dem ursprünglichen Wert von x_1 durchgeführt.
- *Guarded commands* bestehen aus einem boolschen Ausdruck (dem sog. *guard*) und einer Anweisung. Die Anweisung darf nur dann ausgeführt werden, wenn der boolsche Ausdruck initial (siehe **if** und **do**) wahr ist. $(V \neq \emptyset) \longrightarrow x, y := 1, 2$ bedeutet zum Beispiel, dass die Zuweisung $x, y := 1, 2$ nur dann ausgeführt werden kann, wenn V eine nicht leere Menge ist.
- **if** *guarded command*$_1$ [] ... [] *guarded command*$_n$ **fi** steht für eine bedingte Ausführung, bei der zunächst die boolschen Ausdrücke aller *guarded command*$_i$ mit $i \in \{1, \ldots, n\}$ geprüft werden und dann auf nicht deterministische Weise aus der Menge der wahren Ausdrücke ein *guarded command*$_j$ ausgewählt, dessen Anweisung dann ausgeführt wird. Existiert kein solches *guarded command*$_j$ wird die Ausführung der **if** Anweisung sofort beendet.
- **do** *guarded command*$_1$ [] ... [] *guarded command*$_n$ **od** steht für eine wiederholte Ausführung. Hierbei werden in jedem Durchlauf die boolschen Ausdrücke aller *guarded command*$_i$ mit $i \in \{1, \ldots, n\}$ geprüft und wie bei der **if** Anweisung auf nicht deterministische Weise die Anweisung eines *guarded command*$_j$ ausgeführt, dessen Bedingung wahr ist. Existiert kein solches *guarded command*$_j$ wird die wiederholte Ausführung sofort abgebrochen.

Um die Beschreibung des Algorithmus kurz zu halten, erlauben wir auch implizite Zuweisungen in den boolschen Ausdrücken der *guarded commands*[1]. Zum Beispiel ist es für eine PLTL-Formel ϕ zulässig im Anweisungsteil s von $\phi = \phi_1 \wedge \phi_2 \longrightarrow s$ die Variablen ϕ_1 und ϕ_2 zu benutzen.

Datentypen

Als Datentypen wollen wir Mengen und Listen mit beliebigen Elementtypen zulassen. Für Mengen verwenden wir die übliche mathematische Notitation. Listen von Elementen schreiben wir als $\langle x_1, \ldots, x_n \rangle$, das Hinzufügen eines Elements x zu einer Liste L als $L^\frown\langle x \rangle$ bzw. $\langle x \rangle^\frown L$.

[1] Diese impliziten Zuweisungen entsprechen in etwa dem Pattern Matching Mechanismus von O'Caml und ähnlichen Programmiersprachen.

Literaturverzeichnis

[Dij75] Dijkstra, Edsger: Guarded commands, non-determinacy and formal derivation of programs. In: *Communications of the ACM* 18 (1975), August, Nr. 8, S. 453–457

[EL85] Emerson, E. A. ; Lei, C. L.: Modalities for Model Checking: Branching Time Strikes Back. In: *ACM Symposium on Principles of Programming Languages*, 1985, S. 84–96

[GPVW95] Gerth, Rob ; Peled, Doron ; Vardi, Moshe ; Wolper, Pierre: Simple On-the-fly Automatic Verification of Linear Temporal Logic. In: *Protocol Specification Testing and Verification*. Warsaw, Poland : Chapman & Hall, 1995, S. 3–18

[HR04] Huth, Michael ; Ryan, Mark: *Logic in Computer Science*. Cambridge University Press, 2004. – ISBN 0–521–54310–X

[Jun06] Jung, Johannes: Model Checking mittels PLTL. In: *Seminar zur Theoretischen Informatik* (2006)

[Kat99] Katoen, Joost-Pieter: *Arbeitsberichte der Informatik*. Bd. 32-1: *Concepts, Algorithms, and Tools for Model Checking*. Friedrich-Alexander-Universität Erlangen Nürnberg, 1999

[WVS83] Wolper, Pierre ; Vardi, Moshe Y. ; Sistla, A. P.: Reasoning about Infinite Computation Paths. In: *Proc. 24th IEEE Symposium on Foundations of Computer Science*. Tucson, 1983, S. 185–194