Spielbasiertes Model Checking für den alternierungsfreien μ-Kalkül

von
Martin Lange
Matrikelnummer 201254

Diplomarbeit im Fach Informatik

vorgelegt der
Mathematisch-Naturwissenschaftlichen Fakultät
der Rheinisch-Westfälischen Technischen Hochschule Aachen
im August 1999

angefertigt am
Lehrstuhl für Informatik II
bei
Univ.-Prof. Dr. K. Indermark

Ich danke Herrn Prof. Dr. K. Indermark für die Überlassung des Themas sowie für die Bereitschaft, die vorliegende Arbeit zu begutachten, Herrn Prof. Dr. W. Thomas dafür, daß er sich als Zweitgutachter zur Verfügung gestellt hat, Martin Leucker für die Betreuung dieser Arbeit und allen, die mich in dieser Zeit moralisch oder in irgendeiner Weise überhaupt unterstützt haben, damit ich diese Arbeit verfassen konnte.

Hiermit versichere ich, daß ich die vorliegende Arbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Aachen, den 12.8.1999

Inhaltsverzeichnis

Einleitung

Das *automatische Verifizieren* gewisser Eigenschaften real existierender Software- oder Hardware-Systeme ist eine Herausforderung, der sich die Informatik heutzutage stellen muß, da sie als Wissenschaft für die Entwicklung automatisch arbeitender Systeme verantwortlich ist. Ähnlich wie in der Mathematik, wo der Beweis eines Satzes meistens als genauso wichtig erachtet wird wie der Satz selbst, kann sich die Informatik nicht damit begnügen, Programme bzw. Systeme zu entwickeln, ohne deren **Korrektheit** gewährleisten zu können.

Im wesentlichen gibt es zwei Ansätze, Fehler eines Systems auszuschließen. Beiden Ansätzen ist gemeinsam, daß die Hauptaufgabe darin liegt, einen Fehler zu finden und ihn zu verstehen. Das Beheben eines gefundenen Fehlers ist natürlich sehr systemspezifisch und wird u.a. deswegen in dieser Arbeit nicht behandelt.

Testen: Dies ist die wohl gängigste und auch natürlichste Methode. Sie beruht darauf, daß das zu untersuchende System bereits als solches vorliegt und in eine ähnliche Umgebung gebracht wird, wie die, in der es später einwandfrei arbeiten soll. Man sieht jedoch leicht, daß jedes *Testen* nur ein **Semi-Entscheidungsverfahren** liefern kann. Man kann Fehler immer nur finden, niemals jedoch ausschließen.

Automatisches Verifizieren: Gesucht werden also **Entscheidungsverfahren**, die Auskunft darüber geben können, ob ein System korrekt arbeitet *oder nicht*, bzw. ob ein bestimmter Fehler vorhanden ist *oder nicht*. Eine Möglichkeit, dies zu bewerkstelligen, ist **Model Checking** ([CES85]). Mathematisch gesehen ist Model Checking das Beantworten der Frage, ob eine gegebene Struktur eine gegebene logische Formel, die über solchen Strukturen interpretiert werden kann, erfüllt oder nicht. Aus der anwendungsbezogenen Sicht der Informatik dient Model Checking als ein Entscheidungsverfahren, das die oben gestellte Frage beantwortet.

Absolute Sicherheit über die Fehlerfreiheit eines realen Systems kann man auch mithilfe von Model Checking nicht erlangen. Der Grund dafür liegt in der notwendigen Abstraktion eines realen Systems in ein mathematisches Modell, dessen Eigenschaften dann mit einer geeigneten Software überprüft werden können. Ist diese Übersetzung fehlerhaft oder nicht ausreichend, d.h. werden die Eigenschaften, die später durch eine logische Formel beschrieben und getestet werden sollen, nicht in das Modell übernommen, so kann die Antwort des Model Checking Verfahrens unter Umständen wertlos sein. Zum anderen kann es sein, daß die Abstraktion eine Eigenschaft hat, die das reale System nicht hat. Prinzipiell ist dies jedoch kein Unterschied, da man dies auch als Fehlen der entsprechend negierten Eigenschaft ansehen kann.

Die Modelle, die dem Model Checking in dieser Arbeit zugrunde liegen, sind beschriftete Transitionssysteme, d.h. man modelliert Systeme durch atomare Zustände und (nichtdeterministische) Übergänge zwischen diesen Zuständen, die mit einer nach außen sichtbaren

Aktion verbunden sein können. Transitionssysteme sind mitunter die einfachsten Strukturen, die sich für die beschriebenen Zwecke eignen. Dadurch sind sie zwar wohlverstanden und hinreichend untersucht, aber in vielerlei Hinsicht nicht optimal. So kann ein Transitionssytem, das ein reales System, wie z. B. die Steuerungssoftware einen automatisierten Fertigungsprozeßes, ausreichend modelliert, leicht mehrere Millionen Zustände besitzen. Solche Größenordnungen können heutzutage noch nicht effizient verarbeitet werden. Selbst wenn ein Model Checking Verfahren eine lineare Laufzeit- oder Speicherplatzkomplexität bezüglich der Größe eines gegebenen Transitionssystems besitzt, so stößt man in der Praxis trotz der recht kleinen theoretischen Komplexität rasch an die Grenzen des Machbaren.

Aus diesen Gründen gibt es heutzutage in der Informatik Bestrebungen, andere mathematische Modelle zu untersuchen, die das Verhalten von realen Systemen genauso gut beschreiben, die jedoch in gewisser Weise kleiner sind als Transitionssysteme ([WN93]). Leider zeigt sich immer wieder, daß in solchen Fällen die Komplexitäten von Model Checking Verfahren für solche Strukturen viel schlechter sind, so daß man in vielen Fällen keinen Zeit- oder Speicherplatzgewinn erwarten kann. Schlimmer noch ist die Tatsache, daß sich viele Probleme, die andere Strukturen als Transitionssystemen behandeln, unentscheidbar sind, sich also gar nicht automatisch mithilfe eines Algorithmus lösen lassen. Als Beispiele für andere Strukturen, die in dieser Arbeit jedoch keine weitere Betrachtung finden werden, seien **Petri-Netze** ([Pet62, Pet86, Rei82]) und **Traces** ([Maz88, DR95]) genannt.

Zumeist besteht das Problem der richtigen Abstraktion zum mathematischen Modell aus zwei Schritten, die jeweils alle wichtigen Eigenschaften erhalten sollten. Zum einen beschreibt man das zu untersuchende System in einer Spezifikationssprache, wie z. B. **CCS** ([Mil80]), **CSP** ([Hoa78b]), usw. Zum anderen ergibt sich ein Transitionssystem als Semantik eines Programms einer dieser Spezifikationssprachen. Letzterer Schritt geht leicht automatisch vonstatten, da es normalerweise effektive Regeln zur Erzeugung eines Transitionssystems aus einer Spezifikation gibt. [1] Damit ist dann auch gewährleistet, daß beim Übergang zur Transitionssemantik einer Spezifikation keine interessanten Eigenschaften verloren gehen. Aus einem realen System eine geeignete Spezifikation zu generieren, erweist sich jedoch als wesentlich schwieriger und führt in das Forschungsgebiet des *Software-Engineering*.

Die wichtigste Anwendung der automatischen Verifikation von Systemen liefern **verteilte Systeme**. Solche bestehen aus einzelnen Komponenten, die parallel arbeiten, dabei aber miteinander kommunizieren oder auf eine andere Art und Weise interagieren. So kann man sich z. B. die Steuerung eines Flugzeugs aus miteinander kommunizierenden Komponenten, wie Höhensensor, Leitwerk, Kontrollanzeigen, usw. vorstellen. Es ist weiterhin Aufgabe von Ingenieuren, die korrekte Funktion solcher einzelnen Bauteile zu garantieren. Jedoch beherbergt die Interaktion solcher Bauteile ebenso ein Fehlerpotential, das mit Methoden der automatischen Verifikation untersucht werden kann. So ist es in diesem kleinen Beispiel schon schwer genug zu wissen, ob die Interaktion dieser Komponenten nicht dazu führen kann, daß die Steuerung ausfällt, weil z. B. eine Komponente auf eine „Nachricht“ einer anderen Komponente und umgekehrt wartet.

Besonders das Modellieren verteilter Systeme durch Transitionssysteme läßt uns sehr leicht an die Grenzen des Machbaren stoßen, da man meistens den sogenannten **Interleaving**-Ansatz verfolgt, der oft zu einem exponentiellen *blow-up* der Größe des Transitionssystems führt, weil mit diesem Ansatz alle möglichen Zustandsänderungen einer einzelnen Kom-

[1] Im Falle von **CCS** z. B. durch sogenannte *SOS*-Regeln.

ponente zu einer Zustandsänderung des Gesamtsystems führen. Aus diesem Grund ist es wichtig, Model Checking Algorithmen zu haben, die nicht unbedingt durch die Größe eines Transitionssystems praktisch unverwendbar sind.

Oft ist man, falls ein Transitionssystem eine gegebene Formel nicht erfüllt, daran interessiert zu wissen, warum dies der Fall ist. Dies ist wichtig im Hinblick darauf, daß man sich nicht damit zufrieden geben kann zu wissen, daß die Abstraktion eines realen Systems einen Fehler hat. Im allgemeinen wird man ein neues Modell finden wollen, daß diesen Fehler nicht beinhaltet, um später das reale System an das neue, weniger fehlerhafte Modell anzupassen. Aus diesem Grund werden kombinierte Verfahren gebraucht, die sowohl Model Checking durchführen, als auch einen *interaktiven Prozess* zwischen Verifikationssoftware und Benutzer zulassen, der es gestattet, ein abstraktes Modell in Bezug auf eine wünschenswerte Eigenschaft genauer zu untersuchen.

Somit stellen wir drei Anforderungen an einen Model Checking Algorithmus:

- **Komplexität**: Die Laufzeit des Algorithmus sollte nicht nur polynomiell, sondern linear, höchstens jedoch quadratisch in der Größe der Eingabe sein. Die Speicherplatzkomplexität sollte ebenfalls linearen Aufwand nicht überschreiten.

- **Lokalität**: Wir unterscheiden zwischen solchen Algorithmen, die das zu betrachtende Transitionssystem nur bei Bedarf und eventuell nur teilweise aufbauen, und solchen, die es bereits zu Beginn vollständig aufgebaut brauchen. Diese Unterscheidung ist möglich, weil sich Transitionssysteme als Semantik einer Spezifikation ergeben und somit **on the fly** generiert werden können. Ein lokaler Algorithmus, der das Transitionssystem also eventuell gar nicht komplett aufbaut, bietet somit unter Umständen den Vorteil, sehr große Transitionssysteme handhaben zu können.

- **Gegenbeispiel generierend**: Der Algorithmus sollte dem Benutzer die Möglichkeit bieten zu zeigen, warum die gegebene Formel erfüllt bzw. nicht erfüllt ist.

In der vorliegenden Arbeit werden wir, wie bereits erwähnt, davon ausgehen, daß ein Transitionssystem bereits gegeben ist, ohne uns darüber Gedanken zu machen, wie es erzeugt wurde. Dadurch geht zwar der direkte Bezug zu der Modellierung verteilter Systeme verloren, weil einem Transitionssystem nicht mehr angesehen werden kann, ob es ein Einzelsystem oder ein System mit verschiedenen Komponenten beschreibt. Dennoch bleibt die Frage als Anwendung der in dieser Arbeit vorgestellten Theorie bestehen, wie man gewisse Eigenschaften solcher Transitionssysteme effizient nachweisen oder widerlegen kann.

Ebenso wichtig wie die richtige Wahl der Abstraktionsstruktur ist dabei die Wahl der zugrundeliegenden Logik, also der Wahl der Möglichkeiten, was gewünschte oder ungewünschte Eigenschaften sind, die später getestet werden sollen. Im allgemeinen wird man dazu eine **temporale Logik** verwenden, weil sie Aussagen über zeitliche Abläufe eines Systems zuläßt. Die wichtigsten temporalen Logiken sind heutzutage **CTL**, **CTL***, **LTL** ([Eme90]) und der **modale μ-Kalkül** ([Koz83, Sti92]). Nicht jede Eigenschaft eines Transitionssystems läßt sich in jeder Logik ausdrücken. So ist es z. B. nicht möglich, eine **CTL**-Formel anzugeben, die besagt, daß eine bestimmte Aktion immer nur unendlich oft auftritt. Somit lassen sich Logiken bezüglich ihrer Ausdrucksstärke anordnen.

Von den genannten Logiken ist der modale μ-Kalkül bei weitem die ausdrucksstärkste, jedoch auch die unhandlichste. Es ist nicht einfach, einer gegebenen Formel des modalen

μ-Kalküls anzusehen, welche Eigenschaft sie umgangssprachlich beschreibt. Genauso ist es oft schwierig, eine Eigenschaft, die man logisch ausdrücken möchte, als modale μ-Kalkül-Formel aufzuschreiben. Da jedoch effektive Übersetzungen der anderen Logiken in den modalen μ-Kalkül existieren, macht es trotzdem Sinn, Model Checking für diese Logik zu betreiben.

Kap. 1 der vorliegenden Arbeit gibt einen Einblick in den modalen μ-Kalkül.

Kap. 2 ist der Hauptteil dieser Arbeit. Es wird eine Theorie von Spielen entwickelt, deren Sinn es ist, Model Checking für den modalen μ-Kalkül durchzuführen. Diese Theorie geht auf die Arbeiten von Stirling zurück ([Sti97]). Wir geben einen Algorithmus an, der auf diesen Spielen aufbaut und Model Checking für ein Fragment des modalen μ-Kalküls, den sogenannten alternierungsfreien μ-Kalkül erlaubt. Wir zeigen, daß man das Model Checking Problem auch indirekt lösen kann, indem wir angeben, wie es sich auf andere Probleme reduzieren läßt. Somit ließen sich Model Checking Algorithmen aus Verfahren, die verwandte Probleme lösen, herleiten. Am Schluß dieses Kapitels zeigen wir, wie sich die Eigenschaft, Gegenbeispiel generierend zu sein, ausnutzen läßt, um ein Verfahren zu entwickeln, das einem Benutzer interaktiv zeigt, warum eine Formel durch ein Transitionssystem erfüllt oder nicht erfüllt wird.

In Kap. 3 stellen wir drei weitere Algorithmen vor, die das Model Checking Problem für den alternierungsfreien μ-Kalkül lösen. Wir vergleichen diese mit dem Algorithmus aus Kap. 2 im Hinblick auf die drei oben genannten Kriterien. Dadurch wird ersichtlich, warum es sinnvoll ist, noch einen weiteren Algorithmus für dieses Problem zu haben.

Kap. 4 beschreibt schließlich die Implementierung des Algorithmus aus Kap. 2 in dem Verifikationstool TRUTH ([LT98, LLNT99]).

1 Der modale μ-Kalkül

Der **modale μ-Kalkül** ist eine **temporale Logik**, d. h. mit seinen **Formeln** lassen sich Aussagen über zeitliche Abläufe eines modellierten Systems machen. Dazu müssen geeignete **Interpretationen** zur Verfügung stehen.

1.1 Transitionssysteme

Def. 1.1.1 Sei *Acts* eine Menge von **Aktionsnamen**, kurz **Aktionen**. Ein beschriftetes **Transitionssystem** (LTS) ist ein Tupel $\mathcal{T} = (S, T, Acts, \lambda)$. Dabei ist S eine Menge von **Zuständen** und $T \subseteq S \times Acts \times S$ eine Menge von **beschrifteten Transitionen.**[1] Wegen der besseren Lesbarkeit schreiben wir

$$s \xrightarrow{a} t \;\; :\Leftrightarrow \;\; (s, a, t) \in T\,. \qquad \diamond$$

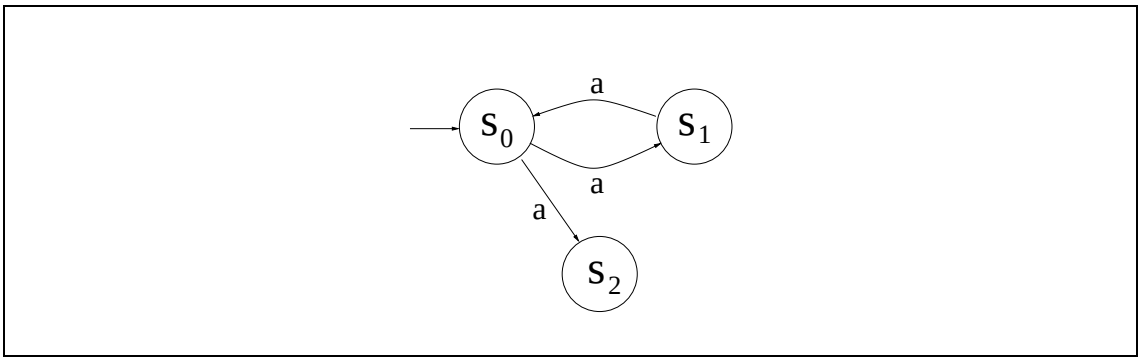

Abbildung 1.1: Ein einfaches Transitionssystem.

Um Transitionssysteme als **Abstraktionen** realer Systeme zu benutzen, muß oft ein ausgezeichneter **Anfangszustand** eingeführt werden. Damit erhalten Pfade (oder Bäume) durch das Transitionssystem die Bedeutung von zeitlichen Abläufen. In grafischen Darstellungen werden wir diese durch einen Pfeil kennzeichnen, der an keinem anderen Knoten beginnt.

[1] $\mathcal{T}$ kann also als **gerichteter Graph** angesehen werden. Dabei entsprechen die Knoten Zuständen und die Kanten Zustandsübergängen (Transitionen), die mithilfe von Aktionen durchgeführt werden.

1.2 Syntax und Semantik

Sei V eine Menge von **propositionalen Variablen** und $Acts$ eine Menge von Aktionsnamen. Formeln des modalen μ-Kalküls $\mathcal{L}_\mu^\infty$ in **positiver Form**, wie er in [Koz83] vorgestellt wurde, sind folgendermaßen induktiv definiert:

Def. 1.2.1 Sei $X \in V$ und $K \subseteq Acts$. Die **Syntax** von $\mathcal{L}_\mu^\infty$ ist gegeben durch:

$$\varphi ::= \mathtt{tt} \mid \mathtt{ff} \mid X \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid [K]\varphi \mid \langle K\rangle\varphi \mid \nu X.\varphi \mid \mu X.\varphi$$

O.B.d.A. seien Variablen, die durch die Operatoren μ und ν gebunden werden, stets paarweise verschieden, d. h. Formeln wie $\mu X.\langle a\rangle X \vee \mu X.[a]X$ werden nicht zugelassen. Stattdessen betrachten wir entsprechende Formeln, die durch Umbenennen der Variablen entstehen, in diesem Fall also z. B. $\mu X.\langle a\rangle X \vee \mu Y.[a]Y$. ◇

Zur Vereinfachung verwenden wir folgende Abkürzungen: [2]

$$\begin{array}{lll} \langle a\rangle\varphi & \text{für} & \langle\{a\}\rangle\varphi \\ \langle -a\rangle\varphi & \text{für} & \langle Acts\backslash\{a\}\rangle\varphi \\ \langle -K\rangle\varphi & \text{für} & \langle Acts\backslash K\rangle\varphi \\ \langle -\rangle\varphi & \text{für} & \langle Acts\rangle\varphi \end{array}$$

Def. 1.2.2 Die **freien Variablen** einer Formel lassen sich induktiv definieren:

$$\begin{array}{lcl} Free(\mathtt{tt}) & = & \emptyset \\ Free(\mathtt{ff}) & = & \emptyset \\ Free(X) & = & \{X\} \\ Free(\varphi \wedge \psi) & = & Free(\varphi) \cup Free(\psi) \\ Free(\varphi \vee \psi) & = & Free(\varphi) \cup Free(\psi) \\ Free(\langle K\rangle\varphi) & = & Free(\varphi) \\ Free([K]\varphi) & = & Free(\varphi) \\ Free(\mu X.\varphi) & = & Free(\varphi) \setminus \{X\} \\ Free(\nu X.\varphi) & = & Free(\varphi) \setminus \{X\} \end{array}$$ ◇

Def. 1.2.3 Genauso läßt sich induktiv die **Fixpunkttiefe** einer Formel φ definieren:

$$\begin{array}{lcl} fd(\mathtt{tt}) & = & 0 \\ fd(\mathtt{ff}) & = & 0 \\ fd(X) & = & 0 \\ fd(\varphi \wedge \psi) & = & max\{fd(\varphi), fd(\psi)\} \\ fd(\varphi \vee \psi) & = & max\{fd(\varphi), fd(\psi)\} \\ fd(\langle K\rangle\varphi) & = & fd(\varphi) \\ fd([K]\varphi) & = & fd(\varphi) \\ fd(\mu X.\varphi) & = & fd(\varphi) + 1 \\ fd(\nu X.\varphi) & = & fd(\varphi) + 1 \end{array}$$ ◇

[2] für den [ ]-Fall analog.

Def. 1.2.4 Für Komplexitätsbetrachtungen z. B. ist die Größe einer Formel wichtig. Dazu brauchen wir die Menge der **Unterformeln** einer Formel φ:

$$\begin{array}{lcl}
Sub(\mathtt{tt}) & = & \{\mathtt{tt}\} \\
Sub(\mathtt{ff}) & = & \{\mathtt{ff}\} \\
Sub(X) & = & \{X\} \\
Sub(\varphi \wedge \psi) & = & \{\varphi \wedge \psi\} \cup Sub(\varphi) \cup Sub(\psi) \\
Sub(\varphi \vee \psi) & = & \{\varphi \vee \psi\} \cup Sub(\varphi) \cup Sub(\psi) \\
Sub(\langle K \rangle \varphi) & = & \{\langle K \rangle \varphi\} \cup Sub(\varphi) \\
Sub([K]\varphi) & = & \{[K]\varphi\} \cup Sub(\varphi) \\
Sub(\mu X.\varphi) & = & \{\mu X.\varphi\} \cup Sub(\varphi) \\
Sub(\nu X.\varphi) & = & \{\nu X.\varphi\} \cup Sub(\varphi)
\end{array}$$

Als **Größe einer Formel** definieren wir $|\varphi| := |Sub(\varphi)|$.[3] ◇

Def. 1.2.5 Die Menge der Unterformeln einer Formel φ bildet mit folgender Ordnung einen Halbverband:[4]

$$\varphi \leq \psi \quad :\Leftrightarrow \quad \varphi \in Sub(\psi)$$

Wir nennen ψ dann größer als φ. Zu je zwei Formeln aus der gleichen Unterformelmenge existiert aufgrund des induktiven Formelaufbaus stets ein Supremum bzgl. $\leq$. ◇

Def. 1.2.6 Schränkt man die Syntax von $\mathcal{L}_\mu^\infty$ auf **fixpunkt-** und **variablenfreie Formeln** ein, so erhält man die Logik **HML**[5]. Es ist also $\varphi \in \mathbf{HML}$, gdw. für alle Variablen X und ein $\psi \in \mathcal{L}_\mu^\infty$ gilt:

$$X \notin Sub(\varphi) \text{ und } \sigma X.\psi \notin Sub(\varphi)$$ ◇

Nachdem wir beschriftete Transitionssysteme eingeführt haben, können wir die Semantik $[\![\varphi]\!]$ einer μ-Kalkül-Formel φ definieren.

Def. 1.2.7 Sei $\mathcal{T} = (S, T, Acts, \lambda)$ ein LTS, und $\rho : V \to 2^S$ eine **Bewertungsfunktion**, die jeder Variable eine Menge von Zuständen zuordnet. Außerdem sei

$$\rho[X/M](Y) = \begin{cases} M, & \text{für } Y = X \\ \rho(Y), & \text{sonst} \end{cases}$$

diejenige Bewertungsfunktion, die bis auf die Stelle X mit ρ übereinstimmt und dort den Wert M annimmt. Die **Semantik**

$$[\![\_]\!]_\rho^{\mathcal{T}} : \ \mathcal{L}_\mu^\infty \to 2^S$$

[3] Es erscheint sinnvoller, als Größe einer Formel deren syntaktische Länge zu verwenden, jedoch ist die Anzahl der Unterformeln linear darin beschränkt.

[4] D.h. zu je zwei Elementen der Struktur existiert bzgl. der Ordnung jeweils ein Supremum, nicht unbedingt jedoch ein Infimum.

[5] **HML** steht für Hennessy-Milner-Logic.

ordnet jeder μ-Kalkül-Formel die Menge von Zuständen des Transitionssystems zu, in der die Formel gilt.

$$
\begin{array}{lcl}
[\![\mathtt{tt}]\!]^{\mathcal{T}}_{\rho} & := & S \\
[\![\mathtt{ff}]\!]^{\mathcal{T}}_{\rho} & := & \emptyset \\
[\![X]\!]^{\mathcal{T}}_{\rho} & := & \rho(X) \\
[\![\varphi \wedge \psi]\!]^{\mathcal{T}}_{\rho} & := & [\![\varphi]\!]^{\mathcal{T}}_{\rho} \ \cap \ [\![\psi]\!]^{\mathcal{T}}_{\rho} \\
[\![\varphi \vee \psi]\!]^{\mathcal{T}}_{\rho} & := & [\![\varphi]\!]^{\mathcal{T}}_{\rho} \ \cup \ [\![\psi]\!]^{\mathcal{T}}_{\rho} \\
[\![\langle K \rangle \varphi]\!]^{\mathcal{T}}_{\rho} & := & \{s \in S \mid \exists\, t \in S,\ \exists a \in K :\ s \xrightarrow{a} t \text{ und } t \in [\![\varphi]\!]^{\mathcal{T}}_{\rho}\} \\
[\![[K] \varphi]\!]^{\mathcal{T}}_{\rho} & := & \{s \in S \mid \forall\, t \in S,\ \forall a \in K :\ \text{wenn } s \xrightarrow{a} t \text{ dann } t \in [\![\varphi]\!]^{\mathcal{T}}_{\rho}\} \\
[\![\mu X.\varphi]\!]^{\mathcal{T}}_{\rho} & := & \bigcap \{R \subseteq S \mid R \subseteq [\![\varphi]\!]^{\mathcal{T}}_{\rho[X/R]}\} \\
[\![\nu X.\varphi]\!]^{\mathcal{T}}_{\rho} & := & \bigcup \{R \subseteq S \mid [\![\varphi]\!]^{\mathcal{T}}_{\rho[X/R]} \subseteq R\}
\end{array}
$$

Außerdem definieren wir:

$$
\begin{array}{rcll}
(\mathcal{T}, s) \models \varphi & :\Leftrightarrow & s \in [\![\varphi]\!]^{\mathcal{T}}_{\rho} & \\
\varphi \equiv \psi & :\Leftrightarrow & \text{für alle } \mathcal{T} \text{ und } \rho\ [\![\varphi]\!]^{\mathcal{T}}_{\rho} = [\![\psi]\!]^{\mathcal{T}}_{\rho} & \diamond
\end{array}
$$

Offensichtlich hängt die Semantik von dem gewählten Transitionssystem ab. Da die Bewertungsfunktion die Semantik mitbestimmt, ist es sinnvoll, nur **Sätze**[6] zu betrachten. Dadurch läßt sich jede beliebige Bewertungsfunktion zur Berechnung der Semantik benutzen, z. B.

$$\rho(X) = \emptyset \text{ für } \forall X \in V$$

Durch die Berechnungsvorschrift für den μ- und den ν-Fall ist garantiert, daß die Bewertungsfunktion beim Auftreten einer Variablen eindeutig ist.
Neben den üblichen **booleschen Operatoren** $\wedge$ und $\vee$ stellt die temporale Logik $\mathcal{L}^{\infty}_{\mu}$ die **Modalitäten** [ ] und $\langle\ \rangle$ zur Verfügung. Mit ihrer Hilfe lassen sich Aussagen über strukturelle Zusammenhänge, genauer über die Nachfolgerrelation in der Interpretation machen. So gilt z. B. die Formel $\langle a \rangle \mathtt{tt}$ in allen Zuständen, die einen a-Nachfolger haben. Allein mit diesen Operatoren ist $\mathcal{L}^{\infty}_{\mu}$ jedoch sehr ausdrucksschwach.[7] Was vor allem fehlt, sind **Quantoren** über Zuständen. Diese werden in gewisser Weise durch den μ- und den ν-Operator bereitgestellt: $[\![\mu X.\varphi(X)]\!]^{\mathcal{T}}_{\rho}$ läßt sich auch interpretieren als in Bezug auf die Mengenkardinalität *kleinste* Lösung der Gleichung

$$Y = \varphi(Y) \text{ für } Y \in 2^S \tag{1.1}$$

wobei φ als Transformation auf Zuständen aufgefaßt wird.[8] Genauso liefert $[\![\nu X.\varphi(X)]\!]^{\mathcal{T}}_{\rho}$ die *größte* Lösung von Gl. 1.1. Dadurch wird der μ-Operator zu einem **existentiellen Quantor** (**zweiter Stufe**) über Mengen von Zuständen, und der ν-Operator entspricht einem **universellen Quantor**. Aufgrund der Art der Gl. 1.1 nennt man μ und ν auch **Fixpunktoperatoren** und ihre Semantik **Fixpunkte**.

[6] Ein Satz ist eine Formel ohne freie Variablen
[7] s. Abschn. 1.4
[8] vgl. Def. 1.2.8

Def. 1.2.8 Sei $\mathcal{T} = (S, T, Acts, \lambda)$. Zu jeder Formel φ und einer Variablen X definieren wir die **Transformationsfunktion** [9]

$$\begin{aligned} \tau_{\varphi,X} : 2^S &\to 2^S \\ \tau_{\varphi,X}(R) &\mapsto [\![\varphi]\!]^{\mathcal{T}}_{\rho[X/R]} \end{aligned}$$

◇

Bem. 1.2.9 Die Funktionen $\tau_{\varphi,X}$ sind für $\varphi \in \mathcal{L}^{\infty}_{\mu}$ **monoton** über dem Potenzmengenverband der Zustände des zugrundeliegenden Transitionssystems, d. h.

$$S \subseteq S' \;\Rightarrow\; \tau_{\varphi,X}(S) \subseteq \tau_{\varphi,X}(S')$$

Dadurch ist gewährleistet, daß die durch μ und ν definierten Fixpunkte unter den Transformationsfunktionen $\tau_{\varphi,X}$ existieren ([Tar55]).

1.3 Beispiele

Bsp. 1.3.1 Einige Beispiele sollen verdeutlichen, welche Aussagen sich in $\mathcal{L}^{\infty}_{\mu}$ machen lassen:

1. $\langle - \rangle \mathtt{tt}$ besagt, daß es einen Folgezustand gibt.
2. $[-]\mathtt{ff}$ besagt, daß es keinen Folgezustand gibt.
3. $\langle a \rangle \varphi$ Es gibt einen Folgezustand, der durch eine a-Aktion erreichbar ist und in dem φ gilt.
4. $[-a]\mathtt{ff}$ Der Zustand kann *höchstens* durch a-Aktionen verlassen werden.
5. aus 3. und 4. zusammen ergibt sich z. B.: $\langle a \rangle \mathtt{tt} \wedge [-a]\mathtt{ff}$, d. h. es sind genau nur a-Aktionen möglich.
6. Die sogenannte **Liveness-Eigenschaft**, daß es einen Pfad gibt, auf dem irgendwann ψ erfüllt wird, ist $\mu X.(\psi \vee \langle - \rangle X)$.
7. Dagegen läßt sich die sogenannte **Safety-Eigenschaft**, daß ψ immer erfüllt wird, so formalisieren: $\nu X.(\psi \wedge [-]X)$.
8. Eine weitere wichtige, temporallogische Aussage ist die **Until**-Bedingung:
   $$\mu X.(\psi \vee (\varphi \wedge \langle - \rangle \mathtt{tt} \wedge [-]X))$$
   besagt, daß auf allen Pfaden φ solange gilt, bis irgendwann einmal ψ gilt.
9. Wenn man ausdrücken möchte, daß nach jeder a-Aktion irgendwann auch noch eine b-Aktion möglich sein soll, dann kann man dies so tun:
   $$\nu X.([a](\mu Y.(\langle b \rangle \mathtt{tt} \vee \langle - \rangle Y)) \wedge [-]X)$$
   Hier ist lediglich die *Safety*-Formel aus 7. mit der *Liveness*-Formel aus 6. und der „für alle a-Aktionen“-Modalität verknüpft worden.

[9] Damit $\tau_{\varphi,X}$ nicht konstant ist, muß $X \in Free(\varphi)$ gelten.

10. Wenn man ausdrücken möchte, daß a- und b-Aktionen auf unendlichen Pfaden immer höchstens abwechselnd auftreten, dann kann man dies ebenfalls mit zwei Fixpunktoperatoren tun:

$$\nu X.[b]\mathtt{ff} \wedge ([a]\nu Y.[a]\mathtt{ff} \wedge [b]X \wedge [-a,b]Y) \wedge [-a,b]X$$

Hier werden zwei *Safety*-Varianten benutzt: Es soll gesichert sein, daß nach einer a-Aktion keine weitere folgt, ohne daß vorher eine b-Aktion erfolgen konnte. Genauso soll der umgekehrte Fall gelten.

11. Daß auf jedem Pfad des Transitionssystems die Aktion a nur endlich oft auftreten darf, läßt sich folgendermaßen formalisieren:

$$\mu X.\nu Y.([a]X \wedge [-a]Y)$$

1.4 Komplexität und Ausdrucksstärke

Def. 1.4.1 Sei φ eine Formel einer Logik $\mathbf{L}$, die über einer Struktur S interpretiert werden kann. Das **Model Checking Problem** besteht dann darin, zu entscheiden, ob $S \models \varphi$ gilt oder nicht. $\diamond$

Def. 1.4.2 Ein Model Checking Algorithmus für $\mathbf{L}$, $\mathbf{L} \leq \mathcal{L}_\mu^\infty$, [10] ist eine Prozedur $\mathbf{A}$ mit Eingabe $\mathcal{T}, s, \varphi$, wobei $\mathcal{T} = (S, T, Acts, \lambda), s \in S, \varphi \in \mathbf{L}$, und Ausgabe `yes` oder `no`, mit

$$\mathbf{A}(\mathcal{T}, s, \varphi) = \mathtt{yes} \;\Leftrightarrow\; (\mathcal{T}, s) \models \varphi$$

$\diamond$

Das Ziel dieses Abschnittes ist, das Model Checking Problem für $\mathcal{L}_\mu^\infty$ komplexitätstheoretisch einzuordnen. Um die Angabe eines nichtdeterministischen Algorithmus **MP**, der polynomielle Zeitkomplexität hat, zu vereinfachen, zeigen wir zunächst:

Lemma 1.4.3 $\mathcal{L}_\mu^\infty$ ist unter Negation abgeschlossen.

Beweis: Dazu erweitern wir die Syntax und Semantik von $\mathcal{L}_\mu^\infty$ um den **Negationsoperator** $\neg$:

$$[\![\neg\varphi]\!]_\rho^\mathcal{T} := S - [\![\varphi]\!]_\rho^\mathcal{T}$$

[10] $\mathbf{L}$ ist eine Sublogik von $\mathcal{L}_\mu^\infty$, also z. B. **HML** oder $\mathcal{L}_\mu^k$ für ein $k \in \mathbb{N}$

Eingabe: $\mathcal{T} = (S, T, Acts, \lambda), s \in S, \varphi \in \mathcal{L}_\mu^\infty$

1. $\psi := \varphi^{\mathcal{C}}$
2. `if A(`$\mathcal{T}, s, \psi$`) = yes then return no`
   `else return yes`

Abbildung 1.2: Algorithmus $\mathbf{B}(\mathcal{T}, s, \varphi)$.

Damit lassen sich einige Operatoren durch andere ausdrücken. Sei $\mathcal{T} = (S, T, Acts, \lambda)$:

1. $\mathtt{ff} \equiv \neg\mathtt{tt}$:
   $[\![\neg\mathtt{tt}]\!]_\rho^\mathcal{T} = S - [\![\mathtt{tt}]\!]_\rho^\mathcal{T} = S - S = \emptyset = [\![\mathtt{ff}]\!]_\rho^\mathcal{T}$
2. $\varphi \wedge \psi \equiv \neg(\neg\varphi \vee \neg\psi)$:
   $[\![\neg(\neg\varphi \vee \neg\psi)]\!]_\rho^\mathcal{T} = S - [\![\neg\varphi \vee \neg\psi]\!]_\rho^\mathcal{T} = S - ([\![\neg\varphi]\!]_\rho^\mathcal{T} \cup [\![\neg\psi]\!]_\rho^\mathcal{T}) = S - ((S - [\![\varphi]\!]_\rho^\mathcal{T}) \cup (S - [\![\psi]\!]_\rho^\mathcal{T})) =^{11} [\![\varphi]\!]_\rho^\mathcal{T} \cap [\![\psi]\!]_\rho^\mathcal{T} = [\![\varphi \wedge \psi]\!]_\rho^\mathcal{T}$
3. $[a]\varphi \equiv \neg\langle a\rangle\neg\varphi$:
   $[\![\neg\langle a\rangle\neg\varphi]\!]_\rho^\mathcal{T} = S - [\![\langle a\rangle\neg\varphi]\!]_\rho^\mathcal{T} = S - \{s \in S \mid \exists t \in S,\ s \xrightarrow{a} t \text{ und } t \in [\![\neg\varphi]\!]_\rho^\mathcal{T}\} = \{s \in S \mid \forall t \in S, \text{ nicht } s \xrightarrow{a} t \text{ oder } t \notin [\![\neg\varphi]\!]_\rho^\mathcal{T}\} = \{s \in S \mid \forall t \in S, \text{ nicht } s \xrightarrow{a} t \text{ oder } t \in [\![\varphi]\!]_\rho^\mathcal{T}\} = [\![[a]\varphi]\!]_\rho^\mathcal{T}$
4. $\nu X.\varphi(X) \equiv \neg\mu X.\neg\varphi(\neg X)$:
   $[\![\neg\mu X.\neg\varphi(\neg X)]\!]_\rho^\mathcal{T} = S - [\![\mu X.\neg\varphi(\neg X)]\!]_\rho^\mathcal{T} =$
   $S - \bigcap \{R \subseteq S \mid [\![\neg\varphi(\neg X)]\!]_{\rho[X/R]}^\mathcal{T} = R\} =$
   $\bigcup_{R\subseteq S} \{S - R \mid [\![\neg\varphi(\neg X)]\!]_{\rho[X/R]}^\mathcal{T} = R\} =$
   $\bigcup_{R\subseteq S} \{S - R \mid [\![\neg\varphi(X)]\!]_{\rho[X/(S-R)]}^\mathcal{T} = R\} =$
   $\bigcup_{R\subseteq S} \{S - R \mid [\![\varphi(X)]\!]_{\rho[X/(S-R)]}^\mathcal{T} = S - R\} =$
   $\bigcup\{R \subseteq S \mid [\![\varphi]\!]_{\rho[X/R]}^\mathcal{T} = R\} = [\![\nu X.\varphi(X))]\!]_\rho^\mathcal{T}$ □

Def. 1.4.4 Unter $\varphi^\mathcal{C}$, dem **Komplement** von φ, verstehen wir diejenige Formel, die entsteht, wenn man die Äquivalenzen aus Lemma 1.4.3 auf die Formel $\neg\varphi$ anwendet, bis sie keine Negation mehr enthält. [12] ◇

Bem. 1.4.5 Offensichtlich gelten dann:
a) $\varphi^{\mathcal{C}\mathcal{C}} = \varphi$
b) $(\mathcal{T}, s) \not\models \varphi \Leftrightarrow (\mathcal{T}, s) \models \varphi^\mathcal{C}$

[11] nach deMorgan
[12] Dies ist in linearer Zeit bezogen auf die Größe der Formel möglich.

Eingabe: $\mathcal{T} = (S, T, Acts, \lambda)$, $s \in S$, $\varphi \in \mathcal{L}_\mu^\infty$
`global` $\forall X \in Var(\varphi) : \rho(X) := \emptyset$
`case` φ of

$\mathtt{tt}$	$\rightarrow$	`return True`
X	$\rightarrow$	`if` $s \in \rho(X)$ `then return True`
		`else return False`
$\neg\varphi$	$\rightarrow$	`return (Not` $\mathbf{MP}(s, \varphi)$`)`
$\varphi_1 \vee \varphi_2$	$\rightarrow$	`errate` $i \in \{1, 2\}$
		`return` $MP(s, \varphi_i)$
$\langle a \rangle \psi$	$\rightarrow$	`if` $T(s, a) = \emptyset$ `then return False`
		`else errate` $t \in \{\ t' \mid s \xrightarrow{a} t'\ \}$
		`return` $\mathbf{MP}(t, \psi)$
$\mu X.\psi(X)$	$\rightarrow$	`errate` $S' \subseteq S$
		$\rho(X) := S'$
		`return` $\mathbf{MP}(s, \psi)$

Abbildung 1.3: Algorithmus $\mathbf{MP}(\mathcal{T}, s, \varphi)$.

Satz 1.4.6 $\mathcal{L}(\mathbf{MP}) \in \mathbf{NP} \Leftrightarrow \mathcal{L}(\mathbf{MP}) \in \mathbf{co-NP}$

Beweis: Angenommen, ein nichtdeterministischer Algorithmus **A** entscheidet in polynomieller Zeit, ob $(\mathcal{T}, s) \models \varphi$ gilt. Dann entscheidet der Algorithmus **B** aus Abb. 1.2 ebenfalls nichtdeterministisch in polynomieller Zeit, ob $(\mathcal{T}, s) \not\models \varphi$ gilt. □

Satz 1.4.7 Das Model Checking Problem für $\mathcal{L}_\mu^\infty$ liegt in **NP**.

Beweis: Sei $\mathcal{T} = (S, T, Acts, \lambda)$. Der nichtdeterministische Algorithmus aus Abb. 1.3 löst das Model Checking Problem für $\mathcal{L}_\mu^\infty$ in polynomieller Zeit. Die Korrektheit folgt sofort nach den Semantikregeln aus Def. 1.2.7. Die Laufzeit ist linear in $|\varphi|$, da in jedem Schritt zu einer Unterformel der aktuellen Formel übergegangen wird, und linear in $|S|$, wenn man die Größe der Formel festhält. □

Satz 1.4.8 Das Model Checking Problem für $\mathcal{L}_\mu^\infty$ liegt in $\mathbf{NP} \cap \mathbf{co-NP}$.

Beweis: folgt sofort aus den Sätzen 1.4.7 und 1.4.6. □

Bsp. 1.3.1 und die in Def. 1.2.7 eingeführten Regeln zeigen, welche Aussagen sich in $\mathcal{L}_\mu^\infty$ machen lassen. $\mathcal{L}_\mu^\infty$ hat aber auch Schwächen. So ist es z. B. nicht möglich auszudrücken, daß man über zwei verschiedene Wege zum selben Zustand gelangt, da keine Variablen über Transitionen zur Verfügung stehen. $\mathcal{L}_\mu^\infty$ ist außerdem **nichtzählend**, d. h. man kann ebenfalls keine Aussagen wie „es gibt einen Weg der Länge n“ machen. Es gibt auch keine **inversen Modalitäten**, so daß man nicht ausdrücken kann, daß ein Zustand Vorgänger eines anderen ist.

1.4.1 Die Alternierungstiefe

Die Anzahl der Wechsel von größten und kleinsten Fixpunktoperatoren in einer Formel ist ein wichtiges Maß für die **Ausdrucksstärke** der Logik, aus der die Formel stammt, und die **Komplexität** des Model Checking Problems.

Def. 1.4.9 Für $\varphi \in \mathcal{L}_\mu^\infty$ ist die **Alternierungstiefe** $ad(\varphi)$ induktiv definiert durch

$$
\begin{array}{lcl}
ad(\mathtt{tt}) & = & 1 \\
ad(\mathtt{ff}) & = & 1 \\
ad(X) & = & 1 \\
ad(\varphi \vee \psi) & = & max(ad(\varphi), ad(\psi)) \\
ad(\varphi \wedge \psi) & = & max(ad(\varphi), ad(\psi)) \\
ad(\langle K \rangle \varphi) & = & ad(\varphi) \\
ad([K]\varphi) & = & ad(\varphi) \\
ad(\mu X.\varphi) & = & \begin{cases} ad(\varphi)+1 & \text{falls } \exists \psi \in Sub(\varphi), \psi = \nu Y.\chi \text{ und } X \in Free(\chi) \\ ad(\varphi) & \text{sonst} \end{cases} \\
ad(\nu X.\varphi) & = & \begin{cases} ad(\varphi)+1 & \text{falls } \exists \psi \in Sub(\varphi), \psi = \mu Y.\chi \text{ und } X \in Free(\chi) \\ ad(\varphi) & \text{sonst} \end{cases}
\end{array}
$$

Mithilfe der Alternierungstiefe lassen sich **Fragmente** der Logik $\mathcal{L}_\mu^\infty$ definieren:

$$\mathcal{L}_\mu^k := \{ \ \varphi \in \mathcal{L}_\mu^\infty \ \mid \ ad(\varphi) \leq k \ \}$$

Unter dem Ausdruck **Alternierungshierarchie** verstehen wir die Folge

$$\mathcal{L}_\mu^1 \ \subseteq \ \mathcal{L}_\mu^2 \ \subseteq \ \mathcal{L}_\mu^3 \ \subseteq \ \ldots \qquad \diamond$$

So gehören z. B. die Formeln 9 und 10 aus Bsp. 1.3.1 zu $\mathcal{L}_\mu^1$, die Formel 11 jedoch echt zu $\mathcal{L}_\mu^2$.

Def. 1.4.10 Wir nennen eine Formel $\varphi \in \mathcal{L}_\mu^\infty$ **alternierungsfrei**, wenn gilt:

$$\varphi \in \mathcal{L}_\mu^1 \qquad \diamond$$

Satz 1.4.11 Die Alternierungshierarchie des μ-Kalküls ist strikt, d.h.:

$$\forall k \in \mathbb{N} \ \exists \varphi \in \mathcal{L}_\mu^{k+1} \text{ und } \nexists \psi \in \mathcal{L}_\mu^k \text{ mit } \varphi \equiv \psi$$

Beweis: Siehe [Bra96]. [13] Es sei angemerkt, daß der Beweis zeigt, daß die Alternierungshierarchie des μ-Kalküls über unendlichen Transitionssystemen strikt ist. Da der μ-Kalkül jedoch die Endliche-Modell-Eigenschaft hat, d. h. jede erfüllbare Formel $\varphi \in \mathcal{L}_\mu^\infty$ wird bereits von einem endlichen Transitionssystem erfüllt, folgt die Striktheit sofort auch für endliche Transitionssysteme. $\square$

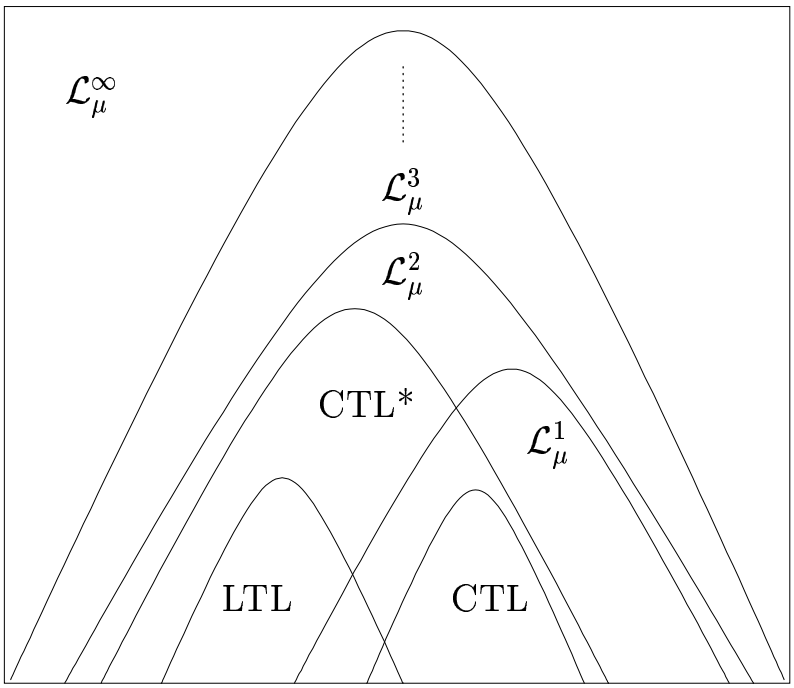

Abbildung 1.4: Ausdrucksstärke von temporalen Logiken.

1.4.2 Andere temporale Logiken

Genauso wie in Abschn. 1.2 läßt sich der Begriff der **Gültigkeit** einer Formel in einem Transitionsystem auch für andere Logiken wie z. B. **CTL**, **CTL*** oder **LTL** ([DRCS97, Eme90]) definieren.[14] Der modale μ-Kalkül ist eine sehr ausdrucksstarke Logik im Vergleich zu diesen anderen temporalen Logiken ([JW96]). Es existieren effektive Verfahren, wie die Logiken **CTL**, **CTL*** und **LTL** in den μ-Kalkül übersetzt werden können ([Dam94]). In allen Fällen ist sogar die Alternierungstiefe der resultierenden Formel beschränkt, d.h. es existieren Funktionen

$$\begin{array}{rcccc} \beta_{CTL} & : & CTL & \rightarrow & \mathcal{L}_\mu^1 \\ \beta_{CTL^*} & : & CTL^* & \rightarrow & \mathcal{L}_\mu^2 \\ \beta_{LTL} & : & LTL & \rightarrow & \mathcal{L}_\mu^2 \end{array}$$

so daß für alle Transitionssystem $\mathcal{T} = (S, T, Acts, \lambda)$ mit $s \in S$ und für alle Formeln $\varphi_1 \in CTL$, $\varphi_2 \in CTL^*$, $\varphi_3 \in LTL$, gilt:

$$\begin{array}{lllll} (\mathcal{T}, s) & \models_{CTL} \varphi_1 & \Leftrightarrow & (\mathcal{T}, s) & \models \beta_{CTL}(\varphi_1) \\ (\mathcal{T}, s) & \models_{CTL^*} \varphi_2 & \Leftrightarrow & (\mathcal{T}, s) & \models \beta_{CTL^*}(\varphi_2) \\ (\mathcal{T}, s) & \models_{LTL} \varphi_3 & \Leftrightarrow & (\mathcal{T}, s) & \models \beta_{LTL}(\varphi_3) \end{array}$$

[13] Da der Beweis nicht einfach und für die vorliegende Arbeit nicht wesentlich ist, wollen wir uns mit der Referenz darauf begnügen.

[14] Wir verzichten hier auf die explizite Angabe von Syntax und Semantik der angesprochenen temporalen Logiken und verweisen dabei auf die angegeben Literatur.

Zur genaueren Betrachtung dieser Zusammenhänge bedarf es einer Definition der einzelnen $\models_L$ Beziehungen. Da wir die Logiken nicht explizit definiert haben, wollen wir uns an dieser Stelle damit begnügen, daß zu einer Logik immer ein Modellbegriff gehört, der lediglich besagt, ob eine Interpretation eine Formel erfüllt oder nicht.

Abb. 1.4 zeigt, wie sich die Fragmente $\mathcal{L}_\mu^n$ des modalen μ-Kalküls und die anderen temporalen Logiken bezüglich ihrer Ausdrucksstärke zueinander verhalten.

1.5 Model Checking Algorithmen für $\mathcal{L}_\mu^\infty$

Im folgenden stellen wir zwei naive Model Checking Algorithmen vor, die nicht effizient, jedoch fundamental für den μ-Kalkül sind. Der erste zeigt, daß man mit exponentiellem Aufwand die Fixpunktformeln, die das Model Checking kompliziert machen, dafür jedoch die Ausdrucksstärke erhöhen, elimieren kann. Der zweite berechnet letztendlich die Semantik einer Formel in Bezug auf ein gegebenes Transitionssystem, wie sie in Def. 1.2.7 eingeführt wurde.

1.5.1 Fixpunktelimination

Ist die Größe des zugrundeliegenden Transitionssystems, d.h. die Anzahl der Zustände, bekannt, z. B. $|S|$, dann kann man jede Formel $\varphi \in \mathcal{L}_\mu^\infty$ in eine äquivalente Formel $\psi \in \mathbf{HML}$ übersetzen. Die Idee, die hinter der Transformation steht, ist die, daß die Berechnung der Semantik einer μ- oder ν-Formel einer monotonen Fixpunktberechnung auf dem Potenzmengenverband der Zustände des Transitionssystems entspricht. Diese ist spätestens nach h Schritten stabil, wobei h die Höhe des Verbandes, in unserem Fall $|S|$ ist.

Def. 1.5.1 Die Transformation ist definiert durch

$$\begin{array}{lccc} \zeta: & \mathcal{L}_\mu^\infty & \to & \mathbf{HML} \\ \\ & \zeta(\mathtt{tt}) & = & \mathtt{tt} \\ & \zeta(\mathtt{ff}) & = & \mathtt{ff} \\ & \zeta(\varphi \wedge \psi) & = & \zeta(\varphi) \wedge \zeta(\psi) \\ & \zeta(\varphi \vee \psi) & = & \zeta(\varphi) \vee \zeta(\psi) \\ & \zeta([K]\varphi) & = & [K]\zeta(\varphi) \\ & \zeta(\langle K\rangle\varphi) & = & \langle K\rangle\zeta(\varphi) \\ & \zeta(\mu X.\varphi) & = & unfold(\zeta(\varphi), \mu, X, |S|) \\ & \zeta(\nu X.\varphi) & = & unfold(\zeta(\varphi), \nu, X, |S|) \end{array}$$

$$\begin{array}{lcl} unfold(\psi, \mu, X, 0) & = & \mathtt{ff} \\ unfold(\psi, \nu, X, 0) & = & \mathtt{tt} \\ unfold(\psi, \sigma, X, n+1) & = & \psi[X \mapsto unfold(\psi, \sigma, X, n)] \end{array}$$

Dabei ist $\varphi[X \mapsto \psi]$ diejenige Formel, die entsteht, wenn jedes Auftreten der Variablen X in φ durch die Formel ψ ersetzt wird. $\diamond$

Eingabe: $\mathcal{T} = (S, T, Acts, \lambda)$, $s \in S$, $\varphi \in \mathbf{HML}$

```
case φ of
     tt          →  return yes
     ff          →  return no
     φ1 ∨ φ2     →  if MCHML(s, φ1) = yes
                    then return yes
                    else return MCHML(s, φ2)
     φ1 ∧ φ2     →  if MCHML(s, φ1) = no
                    then return no
                    else return MCHML(s, φ2)
     ⟨a⟩ψ        →  for t in T(s, a) do
                        if MCHML(t, ψ) = yes
                        then return yes
                    return no
     [a]ψ        →  for t in T(s, a) do
                        if MCHML(t, ψ) = no
                        then return no
                    return yes
```

Abbildung 1.5: Algorithmus $\mathbf{MCHML}(\mathcal{T}, s, \varphi)$.

Somit bleibt eine fixpunkt- und variablenfreie Formel φ übrig, deren Gültigkeit z. B. mit dem deterministischen Algorithmus **MCHML** aus Abb. 1.5 in Zeit $O(|\varphi| \cdot |S|)$ getestet werden kann.
Der Algorithmus ist nicht lokal, weil zu Beginn der Formeltransformation die Größe des Transitionssystems bekannt sein muß.
Die Laufzeit des Algorithmus **MCHML** ist zwar linear in der Größe der Formel, die Formel kann jedoch bei der Transformation exponentiell größer werden, falls Variablen mehrfach auftreten.

1.5.2 Fixpunktberechnung

Die Idee der **Fixpunktberechnung** ist, für jede Unterformel $\psi \in Sub(\varphi)$ sich zu merken, in welchen Zuständen sie gilt. Dabei beginnt man mit atomaren Formeln `tt` und `ff`, die immer in jedem bzw. keinem Zustand gelten. Die Variablen X erhalten den **Initialwert** $\emptyset$ bzw. S, falls sie vom Typ μ bzw. ν sind. Alle anderen Unterformeln können dann anhand der Semantik (s. Def. 1.2.7) behandelt werden. Dieses Verfahren wird iteriert, indem im nächsten Schleifendurchlauf jede Variable X auf den Wert gesetzt wird, der bei der letzten Iteration für den Rumpf der entsprechenden Fixpunktformel $\sigma X.\varphi$ errechnet wurde. Dabei ist zu beachten, daß weiter innen stehende Fixpunktformeln bei jeder Iteration einer äußeren neu berechnet werden müssen. Außerdem kann man jede Formel $\sigma X.\varphi$ bei der Berechnung mit der Variablen X selbst identifizieren. Das Verfahren ist beendet, wenn der Wert der zu testenden Formel selbst stabil ist. Die Korrektheit und Terminierung folgt

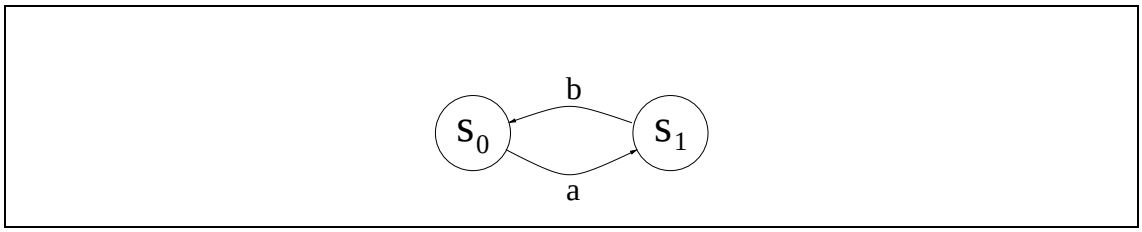

Abbildung 1.6: Ein weiteres Transitionssystem $\mathcal{T}$.

wegen Monotonie[15] aus dem Fixpunktsatz von Tarski ([Tar55]).
Wir wollen uns damit begnügen, den Algorithmus an einem Beispiel vorzuführen.

Bsp. 1.5.2 Sei $\varphi = \mu X.\nu Y.([a]X \wedge [-a]Y)$. [16] Die Fixpunktberechnung zu $\mathcal{T}$ aus Abb.1.6 und φ sieht dann folgendermaßen aus:

X	Y	$[a]X$	$[-a]Y$	$[a]X \wedge [-a]Y$
$\emptyset$	$\{s_0, s_1\}$	$\emptyset$	$\emptyset$	$\emptyset$
$\emptyset$	$\emptyset$	$\emptyset$	$\{s_0\}$	$\emptyset$
$\emptyset$	$\emptyset$	$\emptyset$	$\{s_0\}$	$\emptyset$

Der letzlich stabile Wert $\emptyset$, der der Formel $[a]X \wedge [-a]Y$ und somit auch der gesamten Formel φ zugeordnet wird, besagt, daß φ in keinem Zustand des Transitionssystems gilt.

Man sieht, daß die Fixpunktberechnung „zuviel“ macht, weil sie die Gültigkeit der zu testenden Formel in *jedem* Zustand des Transitionssystems berechnet. Normalerweise reicht es zu wissen, ob die Formle in einem einzelnen, bestimmten Zustand gilt.

[15] s. Bem. 1.2.9
[16] s. Bsp. 1.3.1

2 Spielbasiertes Model Checking

Ziel dieses Kapitels ist es, einen Model Checking Algorithmus für den alternierungsfreien μ-Kalkül vorzustellen, der die drei Kriterien erfüllt, die bereits in der Einleitung angedeutet wurden:

- **Komplexität**: Auch im Hinblick auf eine Implementierung sollte die Laufzeit des Algorithmus nicht nur polynomiell, sondern linear, höchstens jedoch quadratisch in der Größe der Eingabe Transitionssystem und Formel sein. Die Speicherplatzkomplexität sollte ebenfalls linearen Aufwand nicht überschreiten. Unser Algorithmus wird ein Laufzeitkomplexität von $O(n \cdot log\, n)$ und eine Speicherplatzkomplexität von $O(n)$ haben, wobei n die Größe des Transitionssystems und der Formel ist.

- **Lokalität**: Der Algorithmus wird lokal sein, d. h. er wird nicht unbedingt das gesamte Transitionssystem brauchen. Dies erklärt auch den logarithmischen Faktor in der Laufzeitkomplexität. Ein globale Variante, die leicht aus der lokalen zu gewinnen ist, würde lineare Laufzeitkomplexität haben.

- **Gegenbeispiel generierend**: Der Algorithmus bietet im Gegensatz zu den anderen Algorithmen, die in Kap. 3 vorgestellt werden, dem Benutzer die Möglichkeit festzustellen, warum eine gegebene Formel nicht erfüllt ist, falls sie nicht erfüllt ist.

Dabei werden die in [Sti97] vorgestellten Spiele benutzt.

2.1 Spiele für den μ-Kalkül

Ein spielbasierter Algorithmus simuliert zwei **Spieler**, $\forall$***belard*** (I) und $\exists$***loise*** (II). $\forall belard$ hat dabei die Aufgabe, die vorliegende Formel zu widerlegen, während $\exists loise$ sie beweisen muß. Dabei ist das **Spielbrett** das kartesische Produkt aus der Zustandsmenge des Transitionssystems und der Menge der Unterformeln.

Def. 2.1.1 Eine **Partie** besteht aus einer (möglicherweise unendlichen) Folge von Zustand-Formel-Paaren $\mathcal{C}_i$, **Konfigurationen** genannt. Mit $\mathcal{C}_{\mathcal{T}}(s, \varphi)$ bezeichnen wir die Menge aller im **Spiel** $\Gamma_{\varphi,\mathcal{T}}(s, \varphi)$ möglichen Konfigurationen. $\diamond$

Ein Zug verändert die **Formelkomponente** einer Konfiguration und eventuell auch die **Zustandskomponente**. Die **Regeln** dazu sind:

Def. 2.1.2 Sei $\mathcal{T} = (S, T, Acts, \lambda)$ mit Startzustand s_0 und $\varphi \in \mathcal{L}_\mu^\infty$. Dann gilt:

1. Die Partie $\Gamma_{\varphi,\mathcal{T}}(s_0,\varphi)$ ist eine (möglicherweise) unendliche Folge von Konfigurationen $\mathcal{C}_i$ mit $\mathcal{C}_0 = (s_0,\varphi)$.

2. Ist die aktuelle Konfiguration

   - $\mathcal{C}_i = (s, \mathtt{tt})$, dann endet die Partie.
   - $\mathcal{C}_i = (s, \mathtt{ff})$, dann endet die Partie.
   - $\mathcal{C}_i = (s, X)$ und $\psi = \sigma X.\chi \in Sub(\varphi)$, dann ist $\mathcal{C}_{i+1} = (s,\psi)$
   - $\mathcal{C}_i = (s, \varphi_1 \wedge \varphi_2)$, dann wählt $\forall belard$ ein $i \in \{1,2\}$ und $\mathcal{C}_{i+1} = (s,\varphi_i)$
   - $\mathcal{C}_i = (s, \varphi_1 \vee \varphi_2)$, dann wählt $\exists loise$ ein $i \in \{1,2\}$ und $\mathcal{C}_{i+1} = (s,\varphi_i)$
   - $\mathcal{C}_i = (s, [K]\psi)$, dann wählt $\forall belard$ ein $t \in S$ und ein $a \in K$ mit $s \xrightarrow{a} t$, und $\mathcal{C}_{i+1} = (t,\psi)$.
   - $\mathcal{C}_i = (s, \langle K\rangle\psi)$, dann wählt $\exists loise$ ein $t \in S$ und ein $a \in K$ mit $s \xrightarrow{a} t$, und $\mathcal{C}_{i+1} = (t,\psi)$.
   - $\mathcal{C}_i = (s, \nu X.\psi)$, dann ist $\mathcal{C}_{i+1} = (s,\psi)$.
   - $\mathcal{C}_i = (s, \mu X.\psi)$, dann ist $\mathcal{C}_{i+1} = (s,\psi)$.

Wir schreiben $\mathcal{C}_i \to \mathcal{C}_j$, falls ein **Zug** von $\mathcal{C}_i$ nach $\mathcal{C}_j$ möglich ist, bzw. $\mathcal{C}_i \to_p \mathcal{C}_j$ falls Spieler p diesen Zug machen kann. ◇

Def. 2.1.3 Wir definieren nun die **Gewinnbedingungen**, die angeben, welcher Spieler welche Partien gewinnt. Sei $\mathcal{T}$= (S, T, $Acts$, λ) mit Startzustand s_0, $\varphi \in \mathcal{L}_\mu^\infty$ und $s \in S$. $\forall belard$ gewinnt die Partie $\Gamma_{\varphi,\mathcal{T}}(s_0,\varphi)$, falls

- $\Gamma_{\varphi,\mathcal{T}}(s_0,\varphi) = \mathcal{C}_0 \to \mathcal{C}_1 \to \ldots \to \mathcal{C}_n$ und $\mathcal{C}_n = (s, \mathtt{ff})$.
- $\Gamma_{\varphi,\mathcal{T}}(s_0,\varphi) = \mathcal{C}_0 \to \mathcal{C}_1 \to \ldots \to \mathcal{C}_n$ und $\mathcal{C}_n = (s, \langle K\rangle\psi)$ und $\nexists t \in S\ s \xrightarrow{a} t$ für ein $a \in K$.
- $\Gamma_{\varphi,\mathcal{T}}(s_0,\varphi) = \mathcal{C}_0 \to \mathcal{C}_1 \to \ldots$ unendlich ist und eine größte Fixpunktformel, die unendlich oft in den Konfigurationen auftritt, existiert und vom Typ μ ist.[1]

Dagegen gewinnt $\exists loise$ die Partie $\Gamma_{s_0,\mathcal{T}}(\varphi,)$, falls

- $\Gamma_{\varphi,\mathcal{T}}(s_0,\varphi) = \mathcal{C}_0 \to \mathcal{C}_1 \to \ldots \to \mathcal{C}_n$ und $\mathcal{C}_n = (s, \mathtt{tt})$.
- $\Gamma_{\varphi,\mathcal{T}}(s_0,\varphi) = \mathcal{C}_0 \to \mathcal{C}_1 \to \ldots \to \mathcal{C}_n$ und $\mathcal{C}_n = (s, [K]\psi)$ und $\nexists t \in S\ s \xrightarrow{a} t$ für ein $a \in K$.
- $\Gamma_{\varphi,\mathcal{T}}(s_0,\varphi) = \mathcal{C}_0 \to \mathcal{C}_1 \to \ldots$ unendlich ist, und eine größte Fixpunktformel, die unendlich oft in den Konfigurationen auftritt, existiert und vom Typ ν ist. ◇

Die Idee, die hinter dieser Definition steckt, ist die folgende: $\exists loise$ soll nach endlicher Zeit gewonnen haben, wenn sie es schafft, die Partie über eine Reihe von Zügen in **existentiellen Unterformeln**[2] in eine Konfiguration zu bringen, die entweder $\mathtt{tt}$ enthält oder in der

[1] Die Existenz ist gewährleistet, wie Lemma 2.1.8 zeigt.

[2] $\mathtt{tt}$, $\varphi \vee \psi$, $\langle a\rangle\varphi$ sind existentielle, $\mathtt{ff}$, $\varphi \wedge \psi$, $[a]\varphi$ dagegen universelle Formeln.

$\forall belard$ nichts mehr widerlegen kann. Genauso soll $\forall belard$ nach endlicher Zeit gewonnen haben, wenn $\exists loise$ nichts mehr beweisen kann. Eine unendliche Partie soll von $\forall belard$ gewonnen werden, wenn sie durch die Abwicklung eines μ-Fixpunktes zustande kommt, weil kleinste Fixpunkte bewiesen, größte jedoch widerlegt werden müssen. Gerät ein Spiel also in eine Schleife, die immer wieder eine Konfiguration mit einer μ-Formel durchläuft, dann hat $\exists loise$ es nicht geschafft, die Formel zu beweisen.

Bsp. 2.1.4 Sei $\mathcal{T}$ das Transitionssystem aus Abb. 1.1 und $\varphi = \nu X.\langle a\rangle \mathtt{tt} \ \wedge \ \langle -\rangle X$.[3] Dann sind z. B. folgende Partien möglich:

1. $(s_0, \varphi) \to (s_0, \langle a\rangle \mathtt{tt} \wedge \langle -\rangle X) \to_I (s_0, \langle a\rangle \mathtt{tt}) \to_{II} (s_1, \mathtt{tt})$
2. $(s_0, \varphi) \to (s_0, \langle a\rangle \mathtt{tt} \wedge \langle -\rangle X) \to_I (s_0, \langle a\rangle \mathtt{tt}) \to_{II} (s_2, \mathtt{tt})$
3. $(s_0, \varphi) \to (s_0, \langle a\rangle \mathtt{tt} \wedge \langle -\rangle X) \to_I (s_0, \langle -\rangle X) \to_{II} (s_2, X) \to (s_2, \varphi) \to (s_2, \langle a\rangle \mathtt{tt} \wedge \langle -\rangle X) \to_I (s_2, \langle a\rangle \mathtt{tt})$
4. $(s_0, \varphi) \to (s_0, \langle a\rangle \mathtt{tt} \wedge \langle -\rangle X) \to_I (s_0, \langle -\rangle X) \to_{II} (s_1, X) \to (s_1, \varphi) \to (s_1, \langle a\rangle \mathtt{tt} \wedge \langle -\rangle X) \to_I (s_1, \langle -\rangle X) \to_{II} (s_0, X) \to (s_0, \varphi) \to \ldots$

Wie man sieht, werden die Partien 1., 2. und 4. von $\exists loise$ gewonnen. Partie 3. geht jedoch an $\forall belard$. Hätte $\exists loise$ ihren ersten Zug anders gewählt, dann hätte sie auch diese Partie gewinnen können.

Lemma 2.1.5 Sei $\mathcal{T} = (S, T, Acts, \lambda)$ mit $s \in S$ und φ alternierungsfrei. Jede Partie des Spiels $\Gamma_{\varphi,\mathcal{T}}(s, \varphi)$ durchläuft dann keine verschiedenen Fixpunktformeln unendlich oft.

Beweis: Seien $\mu X.\psi$ und $\nu Y.\chi$ zwei Fixpunktformeln, die in einer Partie des Spiels unendlich oft auftreten. Sei $\mu X.\psi$ die zuerst auftretende Formel. Offensichtlich muß die Partie dann folgendermaßen aussehen:

$$(s, \varphi) \to \ldots \to (t_1, \mu X.\psi) \to \ldots \to (t_2, \nu Y.\chi) \to \ldots \to (t_1, X) \to \ldots \to (t_1, \mu X.\psi) \to \ldots$$

Dann gilt jedoch: $X \in Free(\chi)$ und damit $ad(\varphi) > 1$. Analog gilt $ad(\varphi) > 1$ auch, falls $\nu Y.\chi$ die zuerst auftretende Formel ist. □

2.1.1 Zusammenhang zum Model Checking

Das Ziel dieses Abschnitts ist, einen Zusammenhang zwischen den Spielen und dem Model Checking für den μ-Kalkül zu schaffen.

Def. 2.1.6 Sei P ein Spieler. Dann bezeichnet $P^{\mathcal{C}}$ den jeweils anderen Spieler.[4] ◇

Def. 2.1.7 Sei

$$\Gamma_{\varphi_0,\mathcal{T}}(s_0, \varphi_0) = (s_0, \varphi_0) \to_{P_0} (s_1, \varphi_1) \to_{P_1} \ldots$$

[3] Die Formel besagt: Es gibt einen unendlichen Pfad, der überall mit a beschriftet ist.

[4] Falls im Zusammenhang mit einem Spielzug P kein Spieler ist, dann soll auch $P^{\mathcal{C}}$ kein Spieler sein.

eine Partie. Dann ist die **duale Partie**[5] dazu

$$\Gamma^{\mathcal{C}}_{\mathcal{T},\varphi_0}(s_0,\varphi^{\mathcal{C}}_0) = (s_0,\varphi^{\mathcal{C}}_0) \rightarrow_{P^{\mathcal{C}}_0} (s_1,\varphi^{\mathcal{C}}_1) \rightarrow_{P^{\mathcal{C}}_1} \ldots \qquad \diamond$$

Lemma 2.1.8 Jede Partie hat genau einen Sieger.

Beweis: **a)** Zuerst zeigen wir, daß jede Partie so ablaufen muß, daß mindestens eine Gewinnbedingung erfüllt ist:
Angenommen, die Partie ist endlich. Das ist nur möglich, wenn sie in $\mathtt{tt}, \mathtt{ff}, \langle a\rangle\psi$ oder $[a]\psi$ endet. Sei die betrachtete Partie mit der Formel φ nun unendlich lang. Wegen $|\varphi| < \infty$ und den Regeln für die Spiele muß es eine Teilformel $\psi \in Sub(\varphi)$ geben, mit $\psi = \sigma X.\chi$ und ψ tritt in der Partie unendlich oft auf. Damit ist sichergestellt, daß auf jeden Fall einer der beiden Spieler diese Partie gewinnt.
b) Jetzt müssen wir noch zeigen, daß jede Partie höchstens von einem Spieler gewonnen wird. Dazu unterscheiden wir zwei Fälle:
1. Fall: Die Partie ist endlich. Dann hängt der Gewinner nur von der letzten Konfiguration ab. Da diese eindeutig ist, ist auch der Gewinner eindeutig.
2. Fall: Die Partie ist unendlich. Sei φ die betrachtete Formel. Angenommen, die Partie durchläuft unendlich oft Konfigurationen mit den Unterformeln $\sigma_1 X_1.\psi_1$, $\sigma_2 X_1.\psi_2 \in Sub(\varphi)$, für $\sigma_1 \neq \sigma_2, X_1 \neq X_2$. Weil beide unendlich oft vorkommen, sieht die Partie also folgendermaßen aus:

$$(s,\varphi) \rightarrow \ldots \rightarrow (t_1,\sigma_1 X_1.\psi_1) \rightarrow .^{\star}. \rightarrow (t_2,\sigma_2 X_2.\psi_2) \rightarrow .^{\dagger}. \rightarrow (t_1,\sigma_1 X_1.\psi_1) \rightarrow \ldots$$

O.B.d.A. gelte: $\sigma_2 X_2.\psi_2$ kommt nicht vor dem ersten Auftreten von $\sigma_1 X_1.\psi_1$ vor, und zwischen den betrachteten Konfigurationen treten keine weiteren auf, die Variablen in der Formelkomponente enthalten. Weil die Spielregeln immer ein Abstieg in echte Teilformeln oder ein Abwickeln von Fixpunktformeln fordern, muß folgendes gelten: In $\star$ oder in $\dagger$ wird mindestens einmal eine Regel zum Abwickeln der Fixpunkte aus Def. 2.1.2 angewandt, ansonsten wäre $\sigma_1 X_1.\psi_1$ eine echte Teilformel von sich selbst. Dann sind drei Fälle zu unterscheiden:

1. In $\dagger$ wird abgewickelt, in $\star$ jedoch nicht. Dann ist $\sigma_2 X_2.\psi_2 \in Sub(\sigma_1 X_1.\psi_1)$ und damit ist $\sigma_1 X_1.\psi_1$ größer als $\sigma_2 X_2.\psi_2$.

2. In $\star$ wird abgewickelt, in $\dagger$ jedoch nicht. Dann ist $\sigma_1 X_1.\psi_1 \in Sub(\sigma_2 X_2.\psi_2)$ und damit ist $\sigma_2 X_2.\psi_2$ größer als $\sigma_1 X_1.\psi_1$.

3. In $\star$ und $\dagger$ wird abgewickelt. Dann existiert wegen der Halbverbandseigenschaft[6] eine Formel ψ_3 mit $\sigma_1 X_1.\psi_1 \in Sub(\psi_3)$ und $\sigma_2 X_2.\psi_2 \in Sub(\psi_3)$. Angenommen ψ_3 wird nicht unendlich oft durchlaufen. Dann muß einer der beiden Fälle 1. oder 2. gelten. Falls ψ_3 aber ebenfalls unendlich oft durchlaufen wird, muß ψ_3 auch eine Fixpunktformel oder eine Unterformel einer Fixpunktformel ψ_3' sein, die gleichfalls unendlich oft durchlaufen wird.[7]

[5] Diese ist aufgrund der symmetrischen Spielregeln und Komplementbildungsregeln (s. Def. 1.4.4 und 2.1.2) wohldefiniert.
[6] s. Def. 1.2.5
[7] Es kann auch gelten: $\psi_3 = \sigma_1 X_1.\psi_1$, bzw. $\psi_3 = \sigma_2 X_2.\psi_2$

Damit ist gezeigt, daß die Menge der unendlich oft auftretenden Fixpunktformeln in einer Partie immer ein Maximum besitzt, wodurch der Sieger eindeutig bestimmt ist. □

Def. 2.1.9 Eine **Gewinnstrategie** für P ist eine partielle Abbildung

$$\varsigma : \ \mathcal{C}_{\mathcal{T}}(s,\varphi) \to \mathcal{C}_{\mathcal{T}}(s,\varphi)$$

mit

- $dom(\varsigma) = \{ \ \mathcal{C}_i \in \mathcal{C}_{\mathcal{T}}(s,\varphi) \ \mid \ P \text{ ist in } \mathcal{C}_i \text{ am Zug } \}$
- Wenn P in jeder Konfiguration $\mathcal{C}_i$ die Nachfolgekonfiguration $\varsigma(\mathcal{C}_i)$ wählt, dann gewinnt P jede mögliche Partie.

Ein Spieler P gewinnt das **Spiel** $\Gamma_{\varphi,\mathcal{T}}(s_0,\varphi)$, falls es eine Gewinnstrategie für P gibt. ◇

Lemma 2.1.10 P gewinnt das Spiel $\Gamma_{\varphi,\mathcal{T}}(s,\varphi) \ \Rightarrow \ P^{\mathcal{C}}$ gewinnt das Spiel $\Gamma_{\varphi,\mathcal{T}}(s,\varphi)$ nicht.

Beweis: Wenn P eine Gewinnstrategie hat, dann kann er unabhängig von den Zügen seines Gegners das Spiel in eine Partie zwingen, die er gewinnt. Damit kann der Gegner $P^{\mathcal{C}}$ keine Gewinnstrategie haben, weil die Gewinnbedingungen eindeutig sind. □

Def. 2.1.11 Sei ς eine Gewinnstrategie für Spieler P auf dem Spiel $\Gamma_{\varphi,\mathcal{T}}(s,\varphi)$. Dann ist die **duale Strategie** $\varsigma^{\mathcal{C}}$ für $P^{\mathcal{C}}$ auf $\Gamma_{\varphi^{\mathcal{C}},\mathcal{T}}(s,\varphi^{\mathcal{C}})$ definiert durch:

$$\varsigma^{\mathcal{C}}(s,\psi) := (t,\chi) \ \text{ falls } \ \varsigma(s,\psi^{\mathcal{C}}) = (t,\chi^{\mathcal{C}})$$ ◇

Lemma 2.1.12 $\exists loise$ gewinnt das Spiel $\Gamma_{\varphi,\mathcal{T}}(s,\varphi) \ \Leftrightarrow \ \forall belard$ gewinnt das Spiel $\Gamma_{\varphi^{\mathcal{C}},\mathcal{T}}(s,\varphi^{\mathcal{C}})$.

Beweis: $\underline{\Rightarrow}$: Angenommen $\exists loise$ gewinnt das Spiel $\Gamma_{s,\mathcal{T}}(\varphi,)$. Dann existiert eine Partie, für die es drei Möglichkeiten gibt:

1. Die Partie endet in der Konfiguartion $(s,\mathtt{tt})$. Dann endet die duale Partie von $\Gamma_{\varphi^{\mathcal{C}},\mathcal{T}}(s,\varphi^{\mathcal{C}})$ in $(s,\mathtt{ff})$.
2. Die Partie endet in einer Konfiguration $(s,[a]\psi)$. Dann endet die duale Partie von $\Gamma_{\varphi^{\mathcal{C}},\mathcal{T}}(s,\varphi^{\mathcal{C}})$ in $(s,\langle a\rangle\psi)$.
3. Die Partie ist unendlich, und die größte Variable, die unendlich oft durchlaufen wird, ist vom Typ ν. Dann durchläuft die duale Partie in $\Gamma_{\varphi^{\mathcal{C}},\mathcal{T}}(s,\varphi^{\mathcal{C}})$ eine Variable vom Typ μ.

$\exists loise$ gewinnt also keine duale Partie in $\Gamma_{\varphi^c,\mathcal{T}}(s,\varphi^{\mathcal{C}})$ zu einer Gewinnpartie in $\Gamma_{\varphi,\mathcal{T}}(s,\varphi)$. Stattdessen kann $\forall belard$ mithilfe der dualen Strategie zu $\exists loises$ Gewinnstrategie eine Gewinnpartie für sich erzwingen. Damit hat er exakt dieselben Züge auf den Komplementformeln gemacht wie $\exists loise$ sie auf den ursprünglichen Formeln gemacht hätte.
$\underline{\Leftarrow}$: genauso, wegen $\varphi^{\mathcal{CC}} = \varphi$. □

Mit Lemma 2.1.12 ist zudem noch gezeigt, daß die duale Strategie auch eine Gewinnstrategie ist.

Lemma 2.1.13 Sei $P \in \{\forall belard, \exists loise\}$ und ζ die Übersetzungsfunktion von $\mathcal{L}_\mu^\infty$ nach **HML** aus Def. 1.5.1. Dann gilt:

$$P \text{ gewinnt das Spiel } \Gamma_{\varphi,\mathcal{T}}(s,\varphi) \quad \Leftrightarrow \quad P \text{ gewinnt das Spiel } \Gamma_{\zeta(\varphi),\mathcal{T}}(s,\zeta(\varphi))$$

Beweis: Wir zeigen das Lemma durch Induktion über den Formelaufbau.

- $\psi = \mathtt{tt}$: Klar, wegen $\zeta(\psi) = \mathtt{tt}$.
- $\psi = \mathtt{ff}$: Klar, wegen $\zeta(\psi) = \mathtt{ff}$.
- $\psi = \varphi_1 \vee \varphi_2$: $\forall belard$, bzw. $\exists loise$, gewinnt $\Gamma_{\varphi,\mathcal{T}}(s,\psi) \Leftrightarrow \forall i \in \{1,2\}$, bzw. $\exists i \in \{1,2\}$, $\forall belard$, bzw. $\exists loise$, gewinnt das Spiel $\Gamma_{\varphi,\mathcal{T}}(s,\varphi_i) \overset{\text{I.V.}}{\Leftrightarrow} \forall i \in \{1,2\}$, bzw. $\exists i \in \{1,2\}$, $\forall belard$, bzw. $\exists loise$, gewinnt das Spiel $\Gamma_{\zeta(\varphi),\mathcal{T}}(s,\zeta(\varphi_i)) \Leftrightarrow \forall belard$, bzw. $\exists loise$ gewinnt $\Gamma_{\zeta(\varphi),\mathcal{T}}(s,\zeta(\psi))$
- $\psi = \varphi_1 \wedge \varphi_2$: $\forall belard$, bzw. $\exists loise$, gewinnt $\Gamma_{\varphi,\mathcal{T}}(s,\psi) \Leftrightarrow \exists i \in \{1,2\}$, bzw. $\forall i \in \{1,2\}$, $\forall belard$, bzw. $\exists loise$, gewinnt das Spiel $\Gamma_{\varphi,\mathcal{T}}(s,\varphi_i) \overset{\text{I.V.}}{\Leftrightarrow} \exists i \in \{1,2\}$, bzw. $\forall i \in \{1,2\}$, $\forall belard$, bzw. $\exists loise$, gewinnt das Spiel $\Gamma_{\zeta(\varphi),\mathcal{T}}(s,\zeta(\varphi_i)) \Leftrightarrow \forall belard$, bzw. $\exists loise$ gewinnt $\Gamma_{\zeta(\varphi),\mathcal{T}}(s,\zeta(\psi))$
- $\psi = \langle a\rangle\chi$: $\forall belard$, bzw. $\exists loise$, gewinnt $\Gamma_{\varphi,\mathcal{T}}(s,\psi) \Leftrightarrow \forall t$, bzw. $\exists t$, mit $s \xrightarrow{a} t$, so daß $\forall belard$, bzw. $\exists loise$, das Spiel $\Gamma_{\varphi,\mathcal{T}}(s,\chi)$ gewinnt $\overset{\text{I.V.}}{\Leftrightarrow} \forall t$, bzw. $\exists t$, mit $s \xrightarrow{a} t$, so daß $\forall belard$, bzw. $\exists loise$, das Spiel $\Gamma_{\zeta(\varphi),\mathcal{T}}(s,\zeta(\chi))$ gewinnt $\Leftrightarrow \forall belard$, bzw. $\exists loise$ gewinnt $\Gamma_{\zeta(\varphi),\mathcal{T}}(s,\zeta(\psi))$
- $\psi = [a]\chi$: $\forall belard$, bzw. $\exists loise$, gewinnt $\Gamma_{\varphi,\mathcal{T}}(s,\psi) \Leftrightarrow \exists t$, bzw. $\forall t$, mit $s \xrightarrow{a} t$, so daß $\forall belard$, bzw. $\exists loise$, das Spiel $\Gamma_{\varphi,\mathcal{T}}(s,\chi)$ gewinnt $\overset{\text{I.V.}}{\Leftrightarrow} \exists t$, bzw. $\forall t$, mit $s \xrightarrow{a} t$, so daß $\forall belard$, bzw. $\exists loise$, das Spiel $\Gamma_{\zeta(\varphi),\mathcal{T}}(s,\zeta(\chi))$ gewinnt $\Leftrightarrow \forall belard$, bzw. $\exists loise$ gewinnt $\Gamma_{\zeta(\varphi),\mathcal{T}}(s,\zeta(\psi))$
- $\psi = \sigma X.\chi$: Für diese Fälle unterscheiden wir, auf welche Art und Weise P das Spiel $\Gamma_{\varphi,\mathcal{T}}(s,\psi)$ gewinnt. Ist die Gewinnpartie endlich oder unendlich, ohne ψ unendlich oft zu durchlaufen, dann gilt: P gewinnt $\Gamma_{\varphi,\mathcal{T}}(s,\sigma X.\chi) \Leftrightarrow P$ gewinnt $\Gamma_{\varphi,\mathcal{T}}(s,\chi[X/\sigma X.\chi])$. Somit gilt das Lemma auch in diesen Fällen per Induktion.

  Sei nun die Gewinnpartie unendlich und o.B.d.A. damit φ die größte, unendlich oft auftretende Fixpunktformel.

- $\psi = \mu X.\chi$: Sei $P = \forall belard$: Wir zeigen diesen Fall durch Induktion über den Parameter n der $unfold$-Funktion.
  - ▷ $n = 0$: $\zeta(\psi) = \mathtt{ff}$ und $\forall belard$ gewinnt somit $\Gamma_{\zeta(\varphi),\mathcal{T}}(s, \zeta(\psi))$.
  - ▷ $n > 0$: $\forall belard$ gewinnt $\Gamma_{\varphi,\mathcal{T}}(s,\psi) \Leftrightarrow \forall belard$ kann die Partie zu einer Konfiguration (t, X), mit $t \in S$, lenken.[8] $\overset{\text{I.V.}}{\Leftrightarrow}$ $\forall belard$ kann die Partie in die Konfiguration $(s, unfold(\chi, \mu, X, n-1)$ lenken. $\Leftrightarrow$ $\forall belard$ gewinnt $\Gamma_{\zeta(\varphi),\mathcal{T}}(s, unfold(\chi, \mu, X, n)) \Leftrightarrow \forall belard$ gewinnt $\Gamma_{\zeta(\varphi),\mathcal{T}}(s, \zeta(\psi))$.

  Sei nun $P = \exists loise$: $\exists loise$ gewinnt $\Gamma_{\varphi,\mathcal{T}}(s,\psi)$ nicht. Genauso wie im Fall $P = \forall belard$ gewinnt $\exists loise$ dann auch nicht das Spiel $\Gamma_{\zeta(\varphi),\mathcal{T}}(s, \zeta(\psi))$.
- $\psi = \nu X.\chi$: Sei $P = \exists loise$: Wir zeigen diesen Fall durch Induktion über den Parameter n der $unfold$-Funktion.
  - ▷ $n = 0$: $\zeta(\psi) = \mathtt{tt}$ und $\exists loise$ gewinnt somit $\Gamma_{\zeta(\varphi),\mathcal{T}}(s, \zeta(\psi))$.
  - ▷ $n > 0$: $\exists loise$ gewinnt $\Gamma_{\varphi,\mathcal{T}}(s,\psi) \Leftrightarrow \exists loise$ kann die Partie zu einer Konfiguration (t, X), mit $t \in S$, lenken. $\overset{\text{I.V.}}{\Leftrightarrow}$ $\exists loise$ kann die Partie in die Konfiguration $(s, unfold(\chi, \nu, X, n-1)$ lenken. $\Leftrightarrow$ $\exists loise$ gewinnt $\Gamma_{\zeta(\varphi),\mathcal{T}}(s, unfold(\chi, \nu, X, n)) \Leftrightarrow \exists loise$ gewinnt $\Gamma_{\zeta(\varphi),\mathcal{T}}(s, \zeta(\psi))$.

  Sei nun $P = \forall belard$: $\forall belard$ gewinnt $\Gamma_{\varphi,\mathcal{T}}(s,\psi)$ nicht. Genauso wie im Fall $P = \exists loise$ gewinnt $\forall belard$ dann auch nicht das Spiel $\Gamma_{\zeta(\varphi),\mathcal{T}}(s, \zeta(\psi))$. □

Satz 2.1.14 Sei $\mathcal{T} = (S, T, Acts, \lambda)$ mit $s \in S$ und $\varphi \in \mathcal{L}_\mu^\infty$. Dann gilt:

$$(\mathcal{T}, s) \models \varphi \Leftrightarrow \exists loise \text{ gewinnt das Spiel } \Gamma_{\varphi,\mathcal{T}}(s, \varphi)$$

Beweis: $\underline{\Rightarrow}$: Wir zeigen die Korrektheit der Aussage durch Induktion über den Formelaufbau. Wegen Lemma 2.1.13 brauchen wir nur **HML**-Formeln zu betrachten.

- $\psi = \mathtt{tt}$: Für alle $(\mathcal{T}, s)$ gilt: $(\mathcal{T}, s) \models \mathtt{tt}$ und $\Gamma_{\varphi,\mathcal{T}}(s,\psi)$=(s,$\mathtt{tt}$) wird von $\exists loise$ gewonnen.
- $\psi = \mathtt{ff}$: Für kein Paar $(\mathcal{T}, s)$ gilt $(\mathcal{T}, s) \models \mathtt{ff}$. Somit ist die Aussage auch richtig, obwohl $\forall belard$ $\Gamma_{\varphi,\mathcal{T}}(s,\psi)$ gewinnt.
- $\psi = \psi_1 \vee \psi_2$: $(\mathcal{T}, s) \models \psi \Leftrightarrow (\mathcal{T}, s) \models \psi_1$ oder $(\mathcal{T}, s) \models \psi_2 \overset{\text{I.V.}}{\Rightarrow} \exists loise$ gewinnt das Spiel $\Gamma_{\varphi,\mathcal{T}}(s,\psi_1)$ oder das Spiel $\Gamma_{\varphi,\mathcal{T}}(s,\psi_2) \Leftrightarrow \exists loise$ gewinnt auch $\Gamma_{\varphi,\mathcal{T}}(s,\psi) = \ldots \to (s,\psi) \to_{II} (s,\psi_i) \to \ldots$, weil er die Teilformel ψ_i wählen kann.
- $\psi = \psi_1 \wedge \psi_2$: $(\mathcal{T}, s) \models \psi \Leftrightarrow (\mathcal{T}, s) \models \psi_1$ und $(\mathcal{T}, s) \models \psi_2 \overset{\text{I.V.}}{\Rightarrow} \exists loise$ gewinnt das Spiel $\Gamma_{\varphi,\mathcal{T}}(s,\psi_1)$ und das Spiel $\Gamma_{\varphi,\mathcal{T}}(s,\psi_2) \Leftrightarrow \exists loise$ gewinnt auch $\Gamma_{\varphi,\mathcal{T}}(s,\psi) = \ldots \to (s,\psi) \to_I (s,\psi_i) \to \ldots$, obwohl $\forall belard$ die Teilformel ψ_i wählt.
- $\psi = \langle a \rangle \chi$: $(\mathcal{T}, s) \models \psi \Leftrightarrow \exists t \in S$ mit $s \overset{a}{\to} t$ und $(\mathcal{T}, t) \models \chi \overset{\text{I.V.}}{\Rightarrow} \exists t \in S$ mit $s \overset{a}{\to} t$ und $\exists loise$ gewinnt das Spiel $\Gamma_{\varphi,\mathcal{T}}(t,\chi) \Leftrightarrow \exists loise$ gewinnt auch $\Gamma_{\varphi,\mathcal{T}}(s,\psi) = \ldots \to (s,\psi) \to_{II} (t,\chi) \to \ldots$, indem er den Folgezustand t wählt.

[8] Ansonsten würde die Partie nicht unendlich oft ψ durchlaufen.

- $\psi = [a]\chi$: $(\mathcal{T}, s) \models \psi \Leftrightarrow \forall t \in S$ gilt, wenn $s \xrightarrow{a} t$ dann $(\mathcal{T}, t) \models \chi \overset{\text{I.V.}}{\Rightarrow} \forall t \in S$ gilt, wenn $s \xrightarrow{a} t$ dann gewinnt $\exists loise$ das Spiel $\Gamma_{\varphi,\mathcal{T}}(t, \chi) \Leftrightarrow \exists loise$ gewinnt auch $\Gamma_{\varphi,\mathcal{T}}(s, \psi) = \ldots \rightarrow (s, \psi) \rightarrow_I (t, \chi) \rightarrow \ldots$, unabhängig davon, welchen Folgezustand t $\forall belard$ wählt.

$\Leftarrow$: Wir zeigen die Vollständigkeit der Aussage auf indirekte Weise: $(\mathcal{T}, s) \not\models \varphi \overset{1.4.5}{\Leftrightarrow} (\mathcal{T}, s) \models \varphi^{\mathcal{C}} \overset{2.1.14}{\Rightarrow} \exists loise$ gewinnt $\Gamma_{\varphi^{\mathcal{C}},\mathcal{T}}(s, \varphi^{\mathcal{C}}) \overset{2.1.12}{\Leftrightarrow} \forall belard$ gewinnt $\Gamma_{\varphi,\mathcal{T}}(s, \varphi) \overset{2.1.10}{\Rightarrow} \exists loise$ gewinnt $\Gamma_{\varphi,\mathcal{T}}(s, \varphi)$ nicht. □

Korollar 2.1.15 Sei $\mathcal{T} = (S, T, Acts, \lambda)$ mit $s \in S$ und $\varphi \in \mathcal{L}_\mu^\infty$. Dann hat entweder $\forall belard$ oder $\exists loise$ eine Gewinnstrategie für das Spiel $\Gamma_{\varphi,\mathcal{T}}(s, \varphi)$.

Beweis: Angenommen, $\exists loise$ hat keine Gewinnstrategie für $\Gamma_{\varphi,\mathcal{T}}(s, \varphi) \overset{2.1.14}{\Rightarrow} (\mathcal{T}, s) \not\models \varphi \overset{1.4.5}{\Leftrightarrow} (\mathcal{T}, s) \models \varphi^{\mathcal{C}} \overset{2.1.14}{\Rightarrow} \exists loise$ hat eine Gewinnstrategie für $\Gamma_{\varphi^{\mathcal{C}},\mathcal{T}}(s, \varphi^{\mathcal{C}}) \overset{2.1.12}{\Rightarrow} \forall belard$ hat eine Gewinnstrategie für $\Gamma_{\varphi,\mathcal{T}}(s, \varphi)$. Genauso umgekehrt. □

2.2 Gamegraphen

Wir wenden uns nun der Frage zu, wie man die Spiele des vorigen Abschnitts benutzen kann, um automatisch Model Checking für den μ-Kalkül durchzuführen.

Def. 2.2.1 Sei $\mathcal{T} = (S, T, Acts, \lambda)$, $s \in S$ und $\varphi \in \mathcal{L}_\mu^\infty$. Der **Gamegraph** (oder **Spielgraph**) zu s und φ ist:

$$\begin{array}{lcll} \mathcal{G}_\mathcal{T}(s, \varphi) & := & (V, E) \text{ mit} & \\ V & \subseteq & S \times Sub(\varphi) & \\ E & = & \{\ ((t, \psi), (u, \chi)) & \mid\ t, u \in S,\ \psi, \chi \in Sub(\varphi), \\ & & & (s, \psi) \rightarrow (t, \chi) \text{ ist ein Zug in } \Gamma_{\varphi,\mathcal{T}}(s, \varphi)\ \} \end{array}$$

Die Knotenmenge V des Graphen soll dabei stets nur alle von der Startkonfiguration aus durch die Kantenrelation E **erreichbaren** Konfigurationen enthalten. ◇

Ein Gamegraph ist somit die „Vereinigung“ aller möglichen Partien für ein Transitionssystem und eine Formel, bzw. eine Partie ist ein Pfad in dem entsprechenden Gamegraphen.

Bsp. 2.2.2 Abb. 2.1 zeigt den Gamegraphen zu dem Transitionssystem aus Abb. 1.1 und der Formel $\varphi = \nu X.\langle a \rangle \mathtt{tt} \wedge \langle - \rangle X$.

Def. 2.2.3 Sei $\mathcal{G}_\mathcal{T}(s, \varphi) = (V, E)$ ein Gamegraph und $v = (s, \psi) \in V$. Dann ist der **Fixpunkttyp von v** der Fixpunkttyp der größten Fixpunktformel χ, so daß gilt:

$$\chi \in Sub(\varphi),\ \psi \in Sub(\chi)$$ ◇

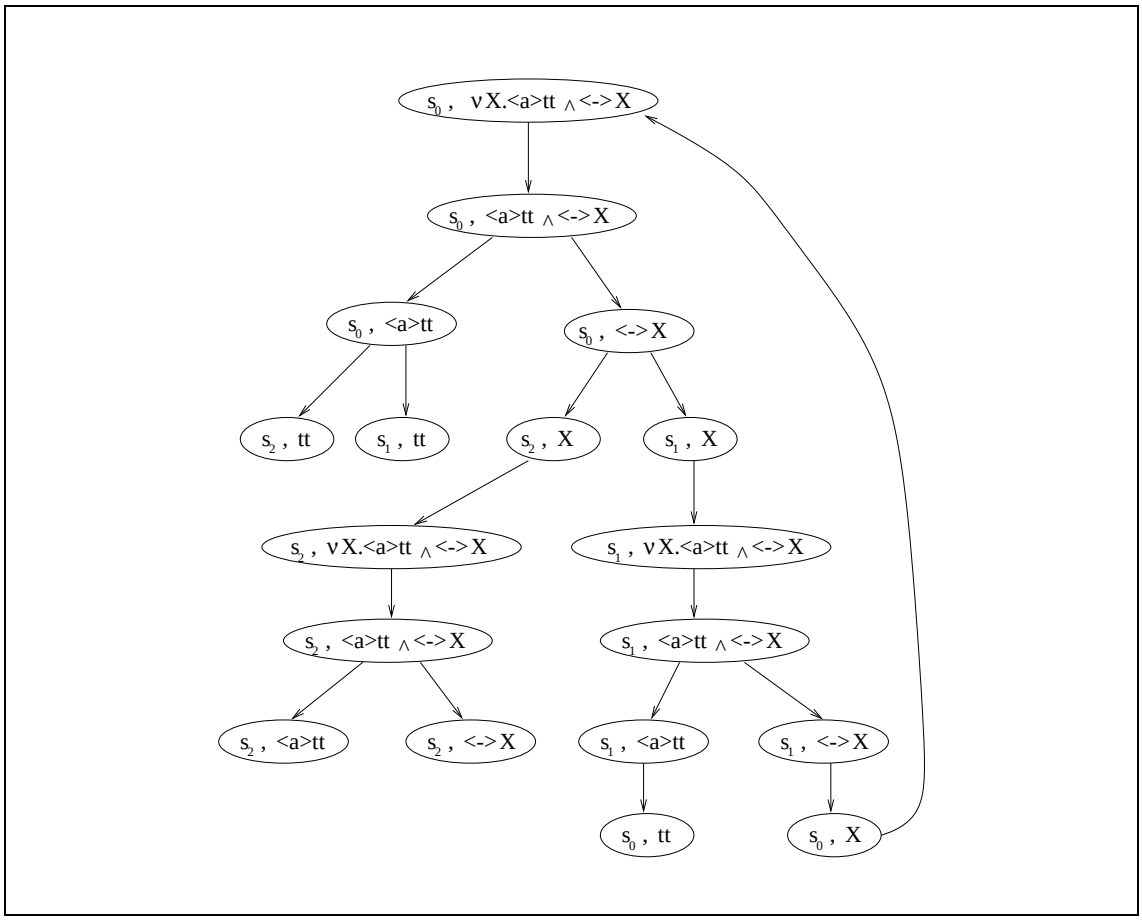

Abbildung 2.1: Ein Gamegraph.

Def. 2.2.4 Sei $\mathcal{T} = (S, T, Acts, \lambda)$, $s \in S$ und $\varphi \in \mathcal{L}_\mu^\infty$. Der **gefärbte Gamegraph** zu s und φ ist:

$$\begin{array}{rcl} \mathcal{G}_\mathcal{T}(s,\varphi) & := & (V,E,\kappa) \quad \text{mit} \\ (V,E) & & \text{ist der Gamegraph zu } s \text{ und } \varphi. \\ \kappa : V \to \{Red, Green\} & & \text{ist eine Färbungsfunktion der Knotenmenge.} \end{array}$$

Wenn noch

$$\kappa(t,\psi) = Green \;\Leftrightarrow\; \exists loise \text{ gewinnt das Spiel } \Gamma_{\varphi,\mathcal{T}}(t,\psi)$$

gilt, dann nennen wir den Gamegraph **korrekt gefärbt**.[9]
Unter dem Ausdruck **Färbeproblem** für Gamegraphen verstehen wir dann die Aufgabe, zu einem gegebenen Gamegraphen eine korrekte Färbung zu finden. ◇

Bsp. 2.2.5 Abb. 2.2 zeigt den gefärbten Gamegraphen zu dem vorherigen Bsp. 2.2.2.

[9] Da wir uns nur für korrekt gefärbte Gamegraphen interessieren, nehmen wir uns die Freiheit, im folgenden einen Gamegraph nur gefärbt zu nennen, wenn wir meinen, daß er korrekt gefärbt ist. Sollte ein Gamegraph nicht korrekt gefärbt sein, so werden wir dies explizit anmerken.

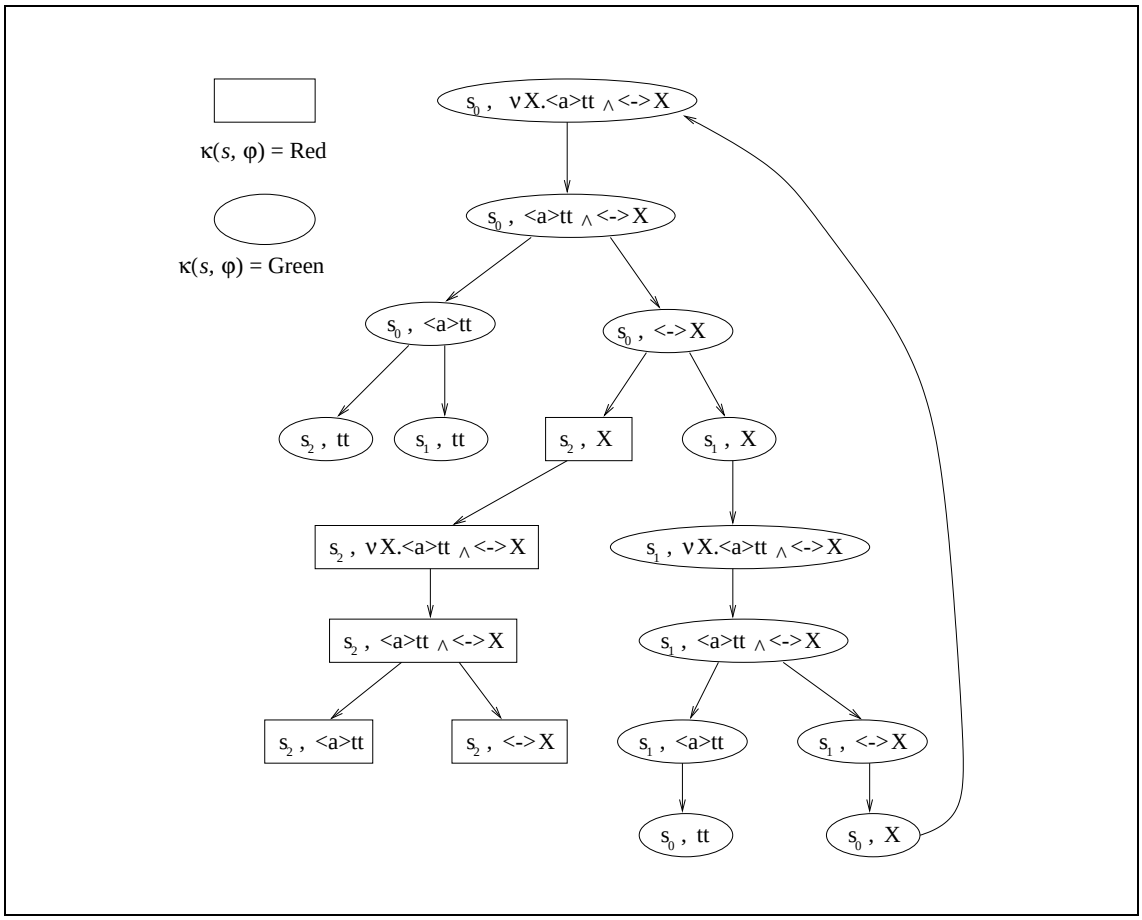

Abbildung 2.2: Ein korrekt gefärbter Gamegraph.

Lemma 2.2.6 Sei $\mathcal{G}_{\mathcal{T}}(s,\varphi) = (V, E, \kappa)$ ein gefärbter Gamegraph. Dann gilt:

$$(\mathcal{T}, s) \models \varphi \quad \Leftrightarrow \quad \kappa(s,\varphi) = Green$$

Beweis: $\kappa(s,\varphi) = Green \overset{2.2.4}{\Leftrightarrow} \exists loise$ gewinnt das Spiel $\Gamma_{\varphi,\mathcal{T}}(s,\varphi) \overset{2.1.14}{\Leftrightarrow} (\mathcal{T}, s) \models \varphi$ □

2.2.1 Boolesche Gleichungssysteme

Jeder Gamegraph läßt sich in linearer Zeit in ein **boolesches Gleichungssystem** umwandeln, so daß die Färbung des Graphen der Lösung des Gleichungssystems entspricht. Die Idee, die hinter dieser Transformation steht, ist, jeden Knoten als Variable aufzufassen. Die bestimmenden, rechten Seiten der Gleichungen sind entweder eine Konjunktion oder eine Disjunktion von Variablen und den **atomaren, booleschen Konstanten** *true* und $false$. Dabei werden die Konfigurationen, in denen $\forall belard$ wählen kann, zu Konjunktionen, weil er widerlegen muß, während $\exists loise's$ Konfigurationen auf Disjunktionen abgebildet werden, weil sie die Formeln beweisen muß.
Jede Gleichung des Systems erhält noch einen Index μ oder ν, der besagt, ob der kleinste oder der größte Fixpunkt der entsprechenden Gleichung berechnet werden soll. Dabei sind

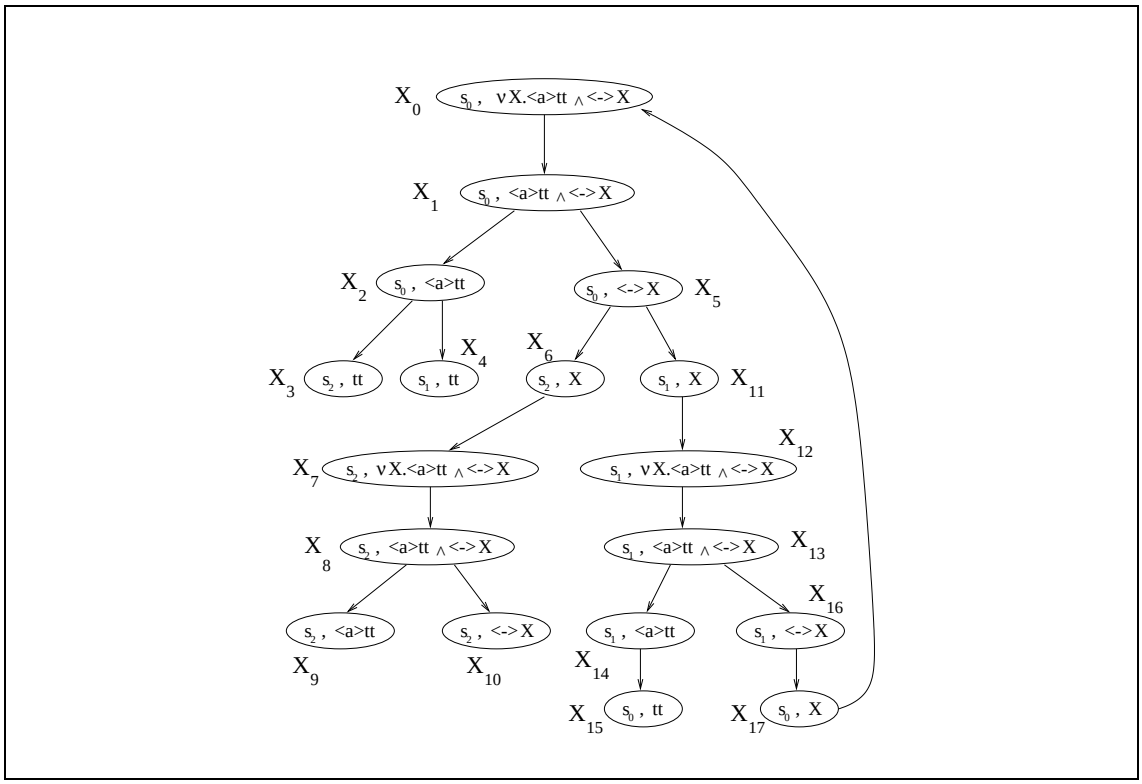

Abbildung 2.3: Gamegraph mit Variablenzuordnung.

diese Indizes nur interessant für Gleichungen, die wirklich eine Rekursion enthalten, d.h. deren entsprechende Knoten im zugehörigen Gamegraphen auf einem **Zykel** liegen.
Wir zeigen diese Transformation anhand eines Beispiels:

Bsp. 2.2.7 Das boolesche Gleichungssystem zum Gamegraph aus Bsp. 2.2.2 ist:

$$
\begin{array}{llll}
X_0 & =_\nu & X_1 & X_1 =_\nu X_2 \wedge X_5 \\
X_2 & =_\nu & X_3 \vee X_4 & X_3 =_\nu true \\
X_4 & =_\nu & true & X_5 =_\nu X_6 \vee X_{11} \\
X_6 & =_\nu & X_7 & X_7 =_\nu X_8 \\
X_8 & =_\nu & X_9 \wedge X_{10} & X_9 =_\nu false \\
X_{10} & =_\nu & false & X_{11} =_\nu X_{12} \\
X_{12} & =_\nu & X_{13} & X_{13} =_\nu X_{14} \wedge X_{16} \\
X_{14} & =_\nu & X_{15} & X_{15} =_\nu true \\
X_{16} & =_\nu & X_{17} & X_{17} =_\nu X_0
\end{array}
$$

Die Zuordnung zwischen Knoten und Variablen ist dabei Abb. 2.3 zu entnehmen.

Bem. 2.2.8 Ist die Formel aus dem Gamegraphen alternierungsfrei, dann hat das zugehörigen Gleichungssystem eine eindeutige Lösung.

Verschiedene Verfahren zum Lösen boolescher Gleichungssysteme, vor allem im Zusammenhang zum Model Checking für den μ-Kalkül, werden in [Mad97] vorgestellt. Einen Überblick über boolesche Gleichungssysteme im Generellen gibt [Fec96].

2.2.2 Alternierende Automaten

In [Eme96] findet sich die Idee, daß sich das Färben von Gamegraphen auf das **Leerheitsproblem** einer bestimmten Klasse von **alternierenden Automaten** ([CKS81, MH84, MS87, HU80]) zurückführen läßt. Im Falle des vollen μ-Kalküls sind dies **Rabin-Automaten**, die auf unendlichen Wörtern arbeiten. Diese unterscheiden sich von normalen **Büchiautomaten** lediglich in der **Akzeptanzbedingung**. Während zur Erkennung eines Wortes durch einen Büchi-Automaten ein Zustand der Endzustandsmenge unendlich oft durchlaufen werden muß, hat ein Rabin-Automat mit Zustandsmenge Q eine Enzustandskomponente

$$\mathfrak{F} = \{(F_0, G_0), \ldots, (F_n, G_n)\}, \text{ mit } F_i, G_i \subseteq Q \text{ für } \forall i = 1, \ldots, n$$

für ein $n \in \mathbb{N}$. Damit ein Rabin-Automat ein Wort akzeptiert, muß es ein Paar $(F_i, G_i) \in \mathfrak{F}$ geben, so daß ein Zustand aus F_i unendlich oft durchlaufen wird, ohne daß ein Zustand aus G_i unendlich oft durchlaufen wird.
Um die Übergangsfunktion eines alternierenden Automaten angeben zu können, brauchen wir noch eine Definition:

Def. 2.2.9 Sei M eine Menge. Dann ist $B^-(M)$, die Menge der **einfachen booleschen Ausdrücke** über M, definiert durch:

$$\begin{aligned} B^-(M) = \; & \{ \; m_1 \; \wedge \; \ldots \; \wedge \; m_n \mid n \in \mathbb{N}, \; m \in M \text{ für } \forall i = 1, \ldots, n \} \cup \\ & \{ \; m_1 \; \vee \; \ldots \; \vee \; m_n \mid n \in \mathbb{N}, \; m \in M \text{ für } \forall i = 1, \ldots, n \} \end{aligned} \qquad \diamond$$

Die allgemeine Übersetzung eines Gamegraphen $\mathcal{G}_\mathcal{T}(s, \varphi) = (V, E)$ in einen alternierenden Automaten $\mathcal{A} = (Q, \Sigma, q_0, \delta, \mathcal{F})$ sieht folgendermaßen aus:

- $Q = V$
- $\Sigma = \{\circ\}$, d. h. das Alphabet ist einelementig, da die Kanten eines Gamegraphen unbeschriftet sind. Damit ist das einzig mögliche Wort, das der Automat erkennen kann $\circ^\omega$.
- $q_0 = (s, \varphi)$, d. h. der Anfangszustand des Automaten ist die Wurzel des Gamegraphen.
- $\delta : Q \to \mathcal{B}^-(Q)$ ist gegeben durch

$$\delta(s, \psi) = \begin{cases} \bigwedge \{ \; (t, \chi) \mid ((s, \psi), (t, \chi)) \in E \text{ und } \forall \mathit{belard} \text{ wählt in } (s, \varphi) \; \} \\ \bigvee \{ \; (t, \chi) \mid ((s, \psi), (t, \chi)) \in E \text{ und } \exists \mathit{loise} \text{ wählt in } (s, \varphi) \; \} \\ (s, \psi), \text{ falls } (s, \psi) \text{ im Gamegraph keinen Nachfolger hat.} \end{cases}$$

- $\mathfrak{F} = \{ \; (F_i, G_i) \mid i = 1, \ldots, ad(\varphi) \; \} \cup \{F_0, \emptyset\}$, wobei

$$\begin{aligned} F_0 &= \{ \; (s, \mathtt{tt}) \mid s \in S \; \} \cup \{ \; (s, [a]\psi) \mid \nexists a \; \nexists t \in S \; s \xrightarrow{a} t \; \} \\ F_i &= \{ \; (s, \nu X.\psi) \mid s \in S, \; ad(\nu X.\psi) = i \; \} \\ G_i &= \{ \; (s, \mu Y.\chi) \mid s \in S, \; \nu X.\psi \in Sub(\chi) \text{ für ein } (s, \nu X.\psi) \in F_i \; \} \end{aligned}$$

Man sieht, daß sich mithilfe der Akzeptanzbedingung von alternierenden Rabin-Automaten und dieser Konstruktion folgender Zusammenhang ergibt:

Lemma 2.2.10 Sei $\Gamma_{\varphi,\mathcal{T}}(s,\varphi)$ ein Spiel mit zugehörigem Gamegraphen $\mathcal{G}$ und $\mathcal{A}_\mathcal{G}$ der entsprechende alternierende Rabin-Automat. Dann gilt:

$$\exists \mathit{loise} \text{ gewinnt das Spiel } \Gamma_{\varphi,\mathcal{T}}(s,\varphi) \;\Leftrightarrow\; \mathcal{A}_\mathcal{G} \text{ erkennt das Wort } \circ^\omega$$

Beweis: Angenommen, $\exists \mathit{loise}$ gewinnt das Spiel $\Gamma_{\varphi,\mathcal{T}}(s,\varphi)$. D. h. eine Partie des Spiels endet entweder in einer Konfiguration $(s,\mathtt{tt})$ oder $(s,[a]\psi)$ oder sie ist unendlich, und die größte Fixpunktformel, die unendlich oft durchlaufen wird, ist vom Typ ν. Aufgrund der Konstruktion der Übergangsfunktion δ durchläuft der Automat in beiden Fällen eine Schleife. Da $\exists \mathit{loise}$ gewinnt, kann es keine größere Fixpunktformel vom Typ μ geben, die ebenfalls unendlich oft durchlaufen wird. Damit ist genau die Erkennungsbedingung für alternierende Rabin-Automaten erfüllt. □

Damit ist das Model Checking für den μ-Kalkül auf das Leerheitsproblem für alternierende Rabin-Automaten zurückgeführt. Dies hat im allgemeinen Fall jedoch exponentielle Komplexität, was speziell für den alternierungsfreien Fall zu hoch ist. Es stellt sich die Frage, ob sich alternierende Rabin-Automaten in geeigneter Weise einschränken lassen, so daß

- das Leerheitsproblem für solche Automaten in linearer Zeit entscheidbar ist.
- sich diese Automaten genau eignen, um Model Checking für den alternierungsfreien μ-Kalkül durchzuführen.

In [BVW94] wird u.a. gezeigt, daß sich dazu im alternierungsfreien Fall sogenannte *1-letter simple weak alternating automata* ([MSS92]) eignen. Dies sind alternierende Automaten über unendlichen Wörtern mit folgenden Einschränkungen:

- **1-letter**: Das Alphabet, über dem der Automat arbeitet, enthält, wie bereits erwähnt, nur einen Buchstaben. Damit ist die erkannte Sprache des Automaten $\mathcal{A}$ entweder $L(\mathcal{A}) = \emptyset$ oder $L(\mathcal{A}) = \{\circ^\omega\}$.
- **simple**: Im allgemeinen läßt man bei alternierenden Automaten in der Transitionsfunktion eine beliebige boolesche Kombination von Nachfolgezuständen zu. Ein alternierender Automat ist dann *simple*, wenn seine Transitionsfunktion vom eingeschränkten Typ $\delta : Q \to \mathcal{B}^-(Q)$ ist. Dies ist im allgemeinen schon ausreichend für jede Klasse von alternierenden Automaten, falls die Größe des Automaten keine Rolle spielt.
- **weak**: Ein alternierenden Automat ist *weak*, falls sich seine Zustandsmenge Q in disjunkte Mengen Q_i zerlegen läßt, so daß für alle i entweder $Q_i \subseteq F$ oder $Q_i \cap F = \emptyset$ gilt, wobei F die Endzustandsmenge des Automaten (mit Büchi-Akzeptanzbedingung ist). Im ersten Fall bezeichnen wir Q_i als akzeptierende, im zweiten Fall als verwerfende Menge. Zusätzlich muß eine partielle Ordnung $\leq$ auf der Menge der Q_i existieren, so daß alle Transitionen des Automaten in dem gleichen Q_i bleiben, oder in eine andere Teilmenge Q_j führen, für die gilt: $Q_j \leq Q_i$. Wegen der Endlichkeit der Zustandsmenge folgt, daß jeder Pfad in dem Automaten schließlich

nur noch Zustände einer einzelnen Menge Q_i durchläuft. Dies ist bei alternierenden Automaten, die aus der Übersetzung von Gamegraphen mit alternierungsfreier μ-Kalkül-Formel entstehen, der Fall, weil die Partition der Zustandsmenge den Zusammenhangskomponenten des Gamegraphen entspricht, die zu einem Fixpunkttyp gehören.[10]

- Außerdem kann man bei *1-letter simple weak alternating automata* auf die komplexe Struktur der Endzustandskomponente verzichten und stattdessen eine Endzustandsmenge $F \subseteq Q$ mit herkömmlicher Büchi-Abzeptanzbedingung wählen.

Folgender Satz beschreibt die Idee des Model Checking durch Reduktion auf alternierende Automaten.

Satz 2.2.11 Sei $\Gamma_{\varphi,\mathcal{T}}(s,\varphi)$ ein Spiel mit zugehörigem Gamegraphen $\mathcal{G}$ und $\mathcal{A}_\mathcal{G}$ der entsprechende alternierende Automat. Dann gilt:

$$L(\mathcal{A}) = \{\circ^\omega\} \quad \Leftrightarrow \quad \kappa(s,\varphi) = Green,$$

wobei (s,φ) die Wurzel des Gamegraphen ist, der dem Automaten $\mathcal{A}$ entspricht.

Beweis: Folgt sofort aus Lemma 2.2.10. □

Bsp. 2.2.12 Der entsprechende Automat zu dem Gamegraph aus Bsp. 2.2.2 ist $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$, wobei

- $Q = \{q_0, q_1, \ldots, q_{17}\}$
- $\Sigma = \{a\}$
- $F = \{q_0, \ldots, q_5, q_{11}, \ldots, q_{17}\}$
- $\delta : Q \to \mathcal{B}^-(Q)$, mit

$$\begin{array}{lcllcl}
\delta(q_0,a) & = & q_1 & \delta(q_1,a) & = & q_2 \wedge q_5 \\
\delta(q_2,a) & = & q_3 \vee q_4 & \delta(q_3,a) & = & q_3 \\
\delta(q_4,a) & = & q_4 & \delta(q_5,a) & = & q_6 \vee q_{11} \\
\delta(q_6,a) & = & q_7 & \delta(q_7,a) & = & q_8 \\
\delta(q_8,a) & = & q_9 \wedge q_{10} & \delta(q_9,a) & = & q_9 \\
\delta(q_{10},a) & = & q_{10} & \delta(q_{11},a) & = & q_{12} \\
\delta(q_{12},a) & = & q_{13} & \delta(q_{13},a) & = & q_{14} \wedge q_{16} \\
\delta(q_{14},a) & = & q_{15} & \delta(q_{15},a) & = & q_{15} \\
\delta(q_{16},a) & = & q_{17} & \delta(q_{17},a) & = & q_0
\end{array}$$

Damit läßt sich Q partitionieren in $Q_1 = Q \backslash F$ und $Q_2 = F$ mit der partiellen Ordnung $Q_1 \leq Q_2$.

Ein Algorithmus, der das Leerheitsproblem für *1-letter simple weak alternating automata* löst, wird in [KWV98] angegeben.

[10] s. Abschn. 2.2.3

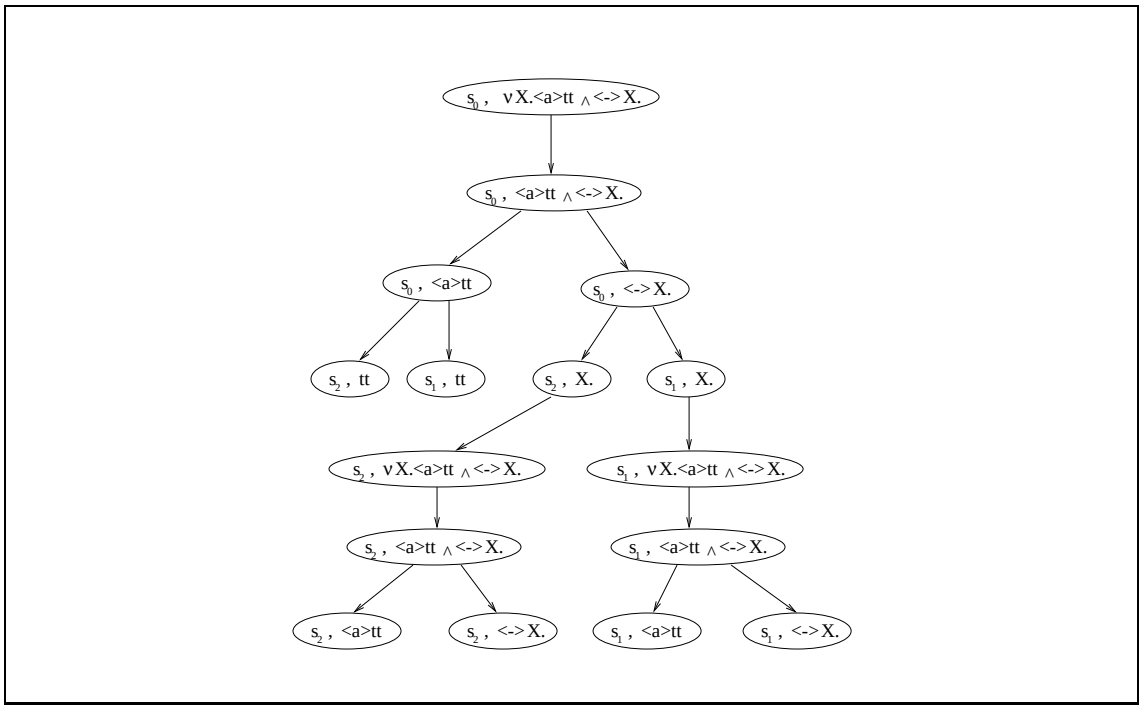

Abbildung 2.4: Ein leicht zu färbender Gamegraph.

2.2.3 Eigenschaften von Gamegraphen

Gamegraphen, die gleichzeitig Bäume sind, lassen sich einfach mit einer Variante des **Tiefensuche-Algorithmus** färben. Dabei durchläuft man den Baum bis zu seinen Blättern, die Konfigurationen der Form $(s, \mathtt{tt})$, $(s, \mathtt{ff})$, $(s, \langle a \rangle \psi)$ oder $(s, [a]\psi)$ sein müssen. Die Farbe dieser Knoten wird dann mithilfe der Rekursion an die darüberliegenden Knoten weitergegeben. Dort berechnet sich die neue Farbe aus der Kombination der Farben der Sohnknoten.

Bsp. 2.2.13 Gäbe es in dem Transitionssystem aus Abb. 1.1 die Kante von s_1 nach s_0 nicht, dann wäre der Gamegraph aus Bsp. 2.1 ein Baum und würde so aussehen, wie er in Abb. 2.4 dargestellt ist.

In einem Gamegraph kann es jedoch auch Knoten geben, die den Graphen von einem Baum unterscheiden. Offensichtlich müssen dies Knoten u sein, für die gilt:

$$ingrad(u) > 1 \tag{2.1}$$

oder

$$ingrad(w) \geq 1$$

falls w die Wurzel des Gamegraphen ist.

Def. 2.2.14 Wir betrachten einen Graphen (V, E) mit $u, v, w \in V$ paarweise verschieden und $(v, u) \in E, (w, u) \in E$. Damit gilt Gl. 2.1 für u. Wir unterscheiden zwei Fälle:

- Es existiert ein Pfad der Länge ≥ 1 von u zu sich selbst und genau einer der beiden Knoten v und w tritt auf dem Pfad auf. Wir nennen u dann einen **Zykelknoten**.

- Es existiert ein weiterer Knoten x, so daß u von x über zwei verschiedene Pfade zu erreichen ist und v und w nicht auf demselben Pfad auftreten. In diesem Fall nennen wir v einen **Joinknoten**. ◇

Da vor allem die Zykelknoten beim Färben des Gamegraphen eine besondere Rolle spielen, beweisen wir noch eine allgemeine Eigenschaft, die insbesondere auch auf sie zutrifft:

Lemma 2.2.15 Sei φ alternierungsfrei. Dann läßt sich an jedem Knoten (t, ψ) eines Gamegraphen $\mathcal{G}_{\mathcal{T}}(s, \varphi)$, der auf einem Zykel im Graphen liegt, in Zeit $O(|\varphi|)$ entscheiden, welches der Typ der größten auftretenden Fixpunktformel ist.

Beweis: Der Zykel des Graphen bildet das Ende einer unendlichen Partie des Spiels $\Gamma_{\varphi,\mathcal{T}}(s, \varphi)$. Nach Lemma 2.1.8 hat diese Partie aufgrund der Eindeutigkeit der größten auftretenden Fixpunktformel einen eindeutigen Gewinner. Um herauszufinden, in welchem Kontext einer Fixpunktformel die Formel ψ innerhalb von φ auftritt, muß man φ höchstens einmal durchlaufen. □

Der naive Tiefensuche-Algorithmus, der auf Bäumen einwandfrei arbeitet, muß aufgrund der Zykelknoten leicht abgeändert werden, um echte Gamegraphen korrekt färben zu können. Damit die Terminierung weiterhin gewährleistet ist, müssen Knoten, die einen Zykelknoten als Nachfolger haben, wie Blätter behandelt werden. Nach Lemma 2.2.15 kann man ihnen die Farbe *Red* (bzw. *Green*) zuweisen, falls sie auf einem Zykel liegen, der vom Typ μ (bzw. ν) ist. Dies entspricht im Algorithmus Fixpunktberechnung der Initialisierung von Variablen mit den Zustandsmengen $\emptyset$ (bzw. S).

Eine weitere Eigenschaft, die bereits im Zusammenhang mit alternierenden Automaten im Abschn. 2.2.2 angesprochen wurde, ist, daß sich ein Gamegraph mit zugrundeliegender alternierungsfreier Formel sinnvoll in **Zusammenhangskomponenten** zerlegen läßt.

Lemma 2.2.16 Sei φ alternierungsfrei und $\mathcal{G}_{\mathcal{T}}(s, \varphi) = (V, E)$ der Gamegraph zu φ und einem Transitionssystem $\mathcal{T} = (S, T, Acts, \lambda)$ mit $s \in S$. Dann existiert eine Partition $\Pi = V_1, \ldots, V_n$ von V, so daß für $v, w \in V$ gilt:
a): $v \in V_i$ und $w \in V_i$ für ein $i \in \{1, \ldots, n\} \;\Rightarrow\; v$ und w haben den gleichen Fixpunkttyp.
b): $\exists$ ein Weg von v nach w und $\exists$ ein Weg von w nach $v \;\Rightarrow\; \exists i$ mit $v \in V_i$ und $w \in V_i$

Beweis: Der Algorithmus **PARTITION** aus Abb. 2.5 findet eine Partition $\Pi = V_1, \ldots, V_n$ der Knotenmenge V eines Gamegraphen. Er ist eine Instanz des Tiefensuchealgorithmus auf Graphen, der die besuchten Knoten der Reihe nach nummeriert. Der Unterschied besteht darin, daß der Algorithmus **PARTITION** nur dann einen Knoten mit einer neuen Nummer markiert, wenn er beim Zurücklaufen eine Fixpunktformel gefunden hat, deren Typ verschieden von dem der zuletzt besuchten Fixpunktformel ist. Dadurch gilt für die Größe n der Partition im alternierungsfreien Fall:

$$n \leq fd(\varphi)$$

Eingabe: Gamegraph $\mathcal{G}_{\mathcal{T}}(s,\varphi) = (V,E)$, $v \in V$
global: $i := 0$, $t :=$ `undefined`, $\forall v \in V : m(v) := 0$, $visited(v) := false$
Ausgabe: Partition $\Pi = V_1, \ldots, V_n$

```
PARTITION(v) =
if visited(v) then
  return ;
visited(v) := true;
S := { w ∈ V | (v,w) ∈ E };
forall w ∈ S do
  PARTITION(w);
if v = (s, σX.ψ) und t ≠ σ then
    t := σ;
    i := i + 1;
m(v) := i
```

Abbildung 2.5: Der Partitionierungsalgorithmus für Gamegraphen

falls der Gamegraph $\mathcal{G}_{\mathcal{T}}(s,\varphi)$ ist. Die Partition Π erhält man letztendlich über folgende Gleichungen:

$$V_i \;=\; \{\; v \in V \;\mid\; m(v) = i \;\}$$

a): Seien v und w Knoten aus V, die nicht denselben Fixpunkttyp haben und sei v der zuerst besuchte Knoten. Weil der Algorithmus dann zwischen w und v mindestens einmal die Variable i erhöht haben muß,[11] gilt:

$$m(v) \geq m(w) + 1$$

Damit können v und w nicht in demselben V_i liegen.
b): Angenommen, es gilt $v \in V_i$ und $w \in V_j$ für $i < j$, und v und w seien stark zusammenhängend. Dann bilden die beiden Wege, die v und w miteinander verbinden, den sich unendlich oft wiederholenden Teil einer Partie des Spiels $\Gamma_{\varphi,\mathcal{T}}(s,\varphi)$. Da $i < j$ gilt, müssen in dieser Partie zwei Fixpunktformeln von verschiedenem Typ auftreten. Nach Lemma 2.1.5 ist dies jedoch ein Widerspruch zur Alternierungsfreiheit von φ. □

Def. 2.2.17 Wir nennen ein Element einer Partition aus Lemma 2.2.16 **Gamegraphkomponente**. ◇

Lemma 2.2.18 Sei $\mathcal{T} = (S,T,Acts,\lambda)$ mit $s \in S$. Dann sind die Gamegraphen $\mathcal{G}_{\mathcal{T}}(s,\varphi) = (V,E)$ und $\mathcal{G}_{\mathcal{T}}(s,\varphi^{\mathcal{C}}) = (V',E')$ zu einer μ-Kalkül-Formel φ und zu deren Komplement $\varphi^{\mathcal{C}}$ isomorph:

$$\mathcal{G}_{\mathcal{T}}(s,\varphi) \;\cong\; \mathcal{G}_{\mathcal{T}}(s,\varphi^{\mathcal{C}})$$

[11] Beachte, daß sich aufgrund der Tiefensuche die Reihenfolge zum Markieren vertauscht hat.

Beweis: Dazu definieren wir einen Isomorphismus $\gamma : V \to V'$ folgendermaßen:

$$\gamma(s, \psi) := (s, \psi^{\mathcal{C}})$$

Wegen Def. 1.4.4 ist γ injektiv und surjektiv, und es gilt außerdem:

$$(v, w) \in E \;\Leftrightarrow\; (\gamma(v), \gamma(w)) \in E'$$ □

Bem. 2.2.19 Betrachtet man einen gefärbten Gamegraphen $\mathcal{G}_{\mathcal{T}}(s, \varphi) = (V, E, \kappa)$ und den entsprechenden Gamegraphen $\mathcal{G}_{\mathcal{T}}(s, \varphi^{\mathcal{C}}) = (V', E', \kappa')$ zu der Komplementformel, dann gilt:

$$\kappa(v) = \mathit{Green} \;\Leftrightarrow\; \kappa'(\gamma(v)) = \mathit{Red}$$

Um das Model Checking Problem zumindest im alternierungsfreien Fall effizient lösen zu können, brauchen wir Kriterien, die besagen, wie man **lokal**, also an einzelnen Knoten, eine korrekte Färbung erzeugen kann. Das nächste Lemma zeigt, welche Bedingungen dazu vorliegen müssen.

Def. 2.2.20 Sei $C_i \to \ldots$ ein Pfad, der in einem Gamegraphen aus einem Zykel vom Typ σ herausführt,[12] heißt **bestimmend**, falls gilt:

- $\sigma = \mu$ und $\exists \mathit{loise}$ wählt in C_i, oder
- $\sigma = \nu$ und $\forall \mathit{belard}$ wählt in C_i. ◇

Ein Pfad heißt somit bestimmend für einen Zykel, wenn der Spieler, der durch den Zykel verlieren würde, das Spiel in diesen Pfad lenken kann. Der endliche Pfad in Abb. 2.7 ist z. B. bestimmend.

Lemma 2.2.21 Sei $\mathcal{G}_{\mathcal{T}}(s, \varphi) = (V, E, \kappa)$ ein gefärbter Gamegraph mit alternierungsfreier Formel $\varphi \in \mathcal{L}^1_\mu$ und $s, t, u \in V$. $\mathcal{G}_{\mathcal{T}}(s, \varphi)$ ist genau dann korrekt gefärbt, wenn gilt:

a) $\kappa(t, \sigma X.\psi) = \kappa(t, \psi)$

b) $\kappa(t, X) = \kappa(t, \sigma X.\psi)$

c) $\kappa(t, \mathtt{tt}) = \mathit{Green}$

d) $\kappa(t, \mathtt{ff}) = \mathit{Red}$

e) $\kappa(t, \varphi_1 \vee \varphi_2) = \mathit{Green} \;\Leftrightarrow\; \exists i \in \{1, 2\}$ mit $\kappa(t, \varphi_i) = \mathit{Green}$

f) $\kappa(t, \varphi_1 \wedge \varphi_2) = \mathit{Red} \;\Leftrightarrow\; \exists i \in \{1, 2\}$ mit $\kappa(t, \varphi_i) = \mathit{Red}$

g) $\kappa(t, \langle a \rangle \psi) = \mathit{Green} \;\Leftrightarrow\; \exists u \in V$ mit $t \xrightarrow{a} u$ und $\kappa(u, \psi) = \mathit{Green}$

h) $\kappa(t, [a]\psi) = \mathit{Red} \;\Leftrightarrow\; \exists u \in V$ mit $t \xrightarrow{a} u$ und $\kappa(u, \psi) = \mathit{Red}$

[12] Der Knoten C_i liegt somit auf dem Zykel.

i) Ein Zykel, aus dem kein bestimmender Pfad herausführt, ist vollständig mit *Green* gefärbt, falls der Zykel durch Abwickeln einer ν-Variable entstanden ist. Im μ-Fall ist er vollständig mit *Red* gefärbt.

Beweis: Einige Fälle lassen sich analog behandeln.

a), b) Diese Konfigurationen haben jeweils genau eine Nachfolgekonfiguration, und kein Spieler hat in ihnen eine Wahlmöglichkeit. Daher muß der Gewinner immer derselbe sein.

c), d) folgt sofort aus den Gewinnbedingungen für endliche Spiele in Def. 2.1.3.

e), g) In diesen Konfigurationen hat $\exists loise$ die Wahl. Wenn sie also eine Nachfolgekonfiguration findet, von der aus sie gewinnt, dann gewinnt sie auch aus der aktuellen Konfiguration heraus.

f), h) In diesen Konfigurationen hat $\forall belard$ die Wahl. Wenn er also eine Nachfolgekonfiguration findet, von der aus er gewinnt, dann gewinnt er auch aus der aktuellen Konfiguration heraus.

i) Diese nicht-lokale Bedingung ist notwendig, weil Zykel, aus denen kein bestimmender Pfad herausführt, nach den Bedingungen **a)-h)** sowohl mit *Green* als auch mit *Red* gefärbt sein könnten. Da aber auf solchen Zykeln offensichtlich nur eine einzige, unendliche Partie möglich ist, und $\exists loise$ diese gewinnt, wenn es sich um einen ν-Zykel handelt, muß in diesem Fall der gesamte Zykel mit *Green* gefärbt sein. Genauso muß er vollständig mit *Red* gefärbt sein, wenn es sich um einen μ-Zykel handelt, weil $\forall belard$ die einzige Partie auf diesem Zykel gewinnt. □

Bem. 2.2.22 Das Model Checking Problem für $\mathcal{L}_\mu^1$ liegt in **NP**.

Obwohl diese Aussage schon in Satz 1.4.7 gezeigt wurde und darüberhinaus eine viel zu große obere Schranke für die Komplexität des Model Checking Problems für den alternierungsfreien μ-Kalkül liefert, erwähnen wir sie hier dennoch, weil sich aus Lemma 2.2.21 leicht der nichtdeterministische Algorithmus **VIALC** (s. Abb. 2.6) für dieses Problem konstruieren läßt. Er hat polynomielle Laufzeit und beinhaltet bereits grundlegende Ideen für den spielbasierten Algorithmus, der in Abschn. 2.8 vorgestellt wird.

2.3 Der Algorithmus CGG

Wir werden nun einen spielbasierten Algorithmus, der das Model Checking Problem für den alternierungsfreien μ-Kalkül löst, vorstellen, komplexitätstheoretisch betrachten sowie seine Korrektheit und Terminierung beweisen.

Eingabe: $\mathcal{T} = (S, T, Acts, \lambda)$, $s \in S$, $\varphi \in \mathcal{L}_\mu^1$

1. konstruiere $\mathcal{G}_\mathcal{T}(s, \varphi) = (V, E)$
2. partitioniere $\mathcal{G}_\mathcal{T}(s, \varphi)$ gemäß Lemma 2.2.16 in $V = V_1 \dot\cup \ldots \dot\cup V_n$
3. for $i = 1, \ldots, n$ do
   färbe Komponente V_i folgendermaßen:
4. suche Zykel, aus denen kein bestimmender Pfad herausführt und färbe diese gemäß Lemma 2.2.21 i)
5. for all $v \in V_i$ do
   falls v noch nicht gefärbt wurde, dann errate $\kappa(v) \in \{Red, Green\}$
6. for all $v \in V_i$ do
   prüfe, ob $\kappa(v)$ nach Lemma 2.2.21 a)-h) korrekt ist

Abbildung 2.6: Algorithmus **VIALC**$(\mathcal{T}, s, \varphi)$.

2.3.1 Intuitive Beschreibung

Die allgemeine Vorgehensweise des Algorithmus ist, wie bereits oben gesagt, eine modifizierte Tiefensuche durch den Gamegraphen. Dabei sucht man nach Blättern, deren Farbe man sofort anhand der Formelkomponente des Blattes feststellen kann, und Zykeln. Die Zykelknoten werden zuerst wie Blätter behandelt, damit der Algorithmus nicht unendlich lang durch den Graphen läuft. Außerdem werden diesen Zykelknoten je nach dem Fixpunkttyp des Zykels vorläufig eine Farbe zugewiesen. Dabei bedeutet „vorläufig“, daß diese später bei Bedarf verändert werden kann.
Daß dies notwendig ist, sieht man in Abb. 2.7, die einen nicht korrekt gefärbten Zykel zeigt. Die Tiefensuche-Strategie ließ den Algorithmus zuerst zum linken Sohn an dem durch $\vee$ angedeuteten *Oder*-Knoten gehen. Auf diesem Pfad fand er dann einen Zykel, der durch eine μ-Formel entstanden ist. Aufgrund dessen färbte der Algorithmus zuerst den Zykelknoten und schließlich den gesamten Zykel zurück bis an den *Oder*-Knoten mit *Red*.[13] Ein *Oder*-Knoten wird jedoch selbst nur *Red*, wenn alle seine Nachfolger mit *Red* gefärbt sind. Daher mußte der Algorithmus auch den rechten Sohn des *Oder*-Knoten untersuchen. Dieser ist ein tt-Knoten, wird also mit *Green* gefärbt. Dadurch erhält auch der *Oder*-Knoten selbst die Farbe *Green*, welche dann weiter nach oben gereicht wird.
Man sieht, daß dies nicht zu einer korrekten Färbung des Gamegraphstücks führte, weil der Vorgänger des Zykelknoten auf dem Zykel letztendlich mit *Red* gefärbt ist, obwohl $\exists$*loise* von diesem Knoten aus eine Gewinnstrategie hat, denn sie gewinnt von dem Zykelknoten aus.

[13] Die Färbung der einzelnen Knoten ist genauso wie in Abb. 2.2 angezeigt.

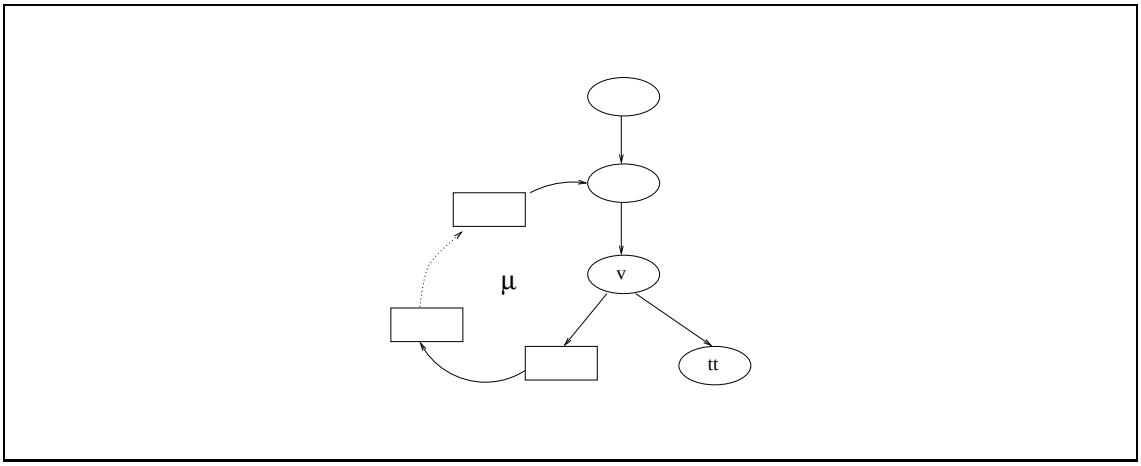

Abbildung 2.7: Ein nicht korrekt gefärbter Zykel.

Dies allein sorgt noch nicht dafür, daß der Algorithmus eine nicht korrekte Antwort berechnet. Führt jedoch aus einem anderen Teil des Gamegraphen eine Kante in den Zykel, enthält der Zykel also einen Joinknoten, und übernimmt der Algorithmus dort einfach die vorhandene Farbe, dann können sich Fehler sehr wohl fortsetzen und zu einer nicht korrekten Antwort des Algorithmus führen.
An Joinknoten die Farbe nicht zu übernehmen, ist noch viel weniger sinnvoll, weil sich der Algorithmus dann an jedem Knoten merken müßte, ob die Färbung bereits gesichert ist oder nicht, was nicht einfach zu entscheiden ist. Um weder solche zusätzliche Information vermerken zu müssen, noch die Terminierung des Algorithmus zu gefährden, verlangen wir, daß eine Färbung eines Knoten korrekt sein muß, bevor dieser sich als Joinknoten entpuppen kann. D.h. der Algorithmus muß die Fehlfärbung eines Zykels, wie z. B. in Abb. 2.7 gegeben, beheben, bevor er diese Gamegraphkomponente wieder verläßt.

Abb. 2.8 zeigt den Algorithmus. Er erhält als Eingabe ein Transitionssystem $\mathcal{T} = (S, T, Acts, \lambda)$ mit ausgezeichnetem Anfangszustand s und eine alternierungsfreie μ-Kalkül-Formel φ. Folgende globale Datenstrukturen müssen zur Verfügung stehen:

- $\forall t \in S, \forall \psi \in Sub(\varphi): \ \pi(t,\psi) \subseteq S \times Sub(\varphi)$
  Für jeden Knoten (t,ψ) merkt sich der Algorithmus diejenigen Zykelknoten, die unmittelbarer Vorgänger des Knotens selbst sind.
- $\forall t \in S, \forall \psi \in Sub(\varphi): \ \kappa(t,\psi) \in \{White, Green, Red\}$
  Jeder Knoten erhält zu Anfang die Farbe $White$. Dadurch lassen sich Zykel erkennen. Zum Schluß wird jeder besuchte Knoten entweder mit $Green$ oder Red markiert sein.
- `pred` ist dazu da, sich den Knoten zu merken, der vor dem aktuellen besucht wurde.

Die Aufgabe der lokalen Variablen `color` ist lediglich, sich die soeben berechnete Farbe Red bzw. $Green$ des Knotens zu merken.

Zuerst prüft der Algorithmus, ob der besuchte Knoten bereits gefärbt wurde. Ist dies der Fall, dann kann die gefundene Farbe zurückgegeben werden, da es sich um einen Joinknoten handelte.

Eingabe: $\mathcal{T} = (S, T, Acts, \lambda)$, $s \in S$, $\varphi \in \mathcal{L}_\mu^1$
global: $pred :=$ `undefined`
$\forall t \in S \ \forall \psi \in Sub(\varphi) : \ \pi(t,\psi) := \emptyset, \ \kappa(t,\psi) :=$ `undefined`
Ausgabe: $\mathcal{G}_\mathcal{T}(s,\varphi) = (V,E)$, $\kappa : V \to \{Green, Red\}$

```
CGG(s, φ) =
local color;
case κ(s,φ) of
 Green       →  return Green
 Red         →  return Red
 White       →  π(s,φ) := π(s,φ) ∪ {pred}
                return CYCLETYPE-COLOR(φ)
 otherwise   →  pred := (s,φ);
                κ(s,φ) := White;
                case φ of
                  tt          →  color := Green
                  ff          →  color := Red
                  φ1 ∨ φ2     →  if CGG(s,φ1) = Green
                                 then color := Green
                                 else color := CGG(s,φ2)
                  φ1 ∧ φ2     →  if CGG(s,φ1) = Red
                                 then color := Red
                                 else color := CGG(s,φ2)
                  X           →  color := CGG(s, σX.ψ))
                  μX.ψ        →  color := CGG(s,ψ))
                  νX.ψ        →  color := CGG(s,ψ))
                  ⟨a⟩ψ        →  succslist := { t | s -a-> t };
                                 for t in succslist do
                                   if CGG(t,ψ) = Green then
                                     color := Green ; casebreak
                                 color := Red
                  [a]ψ        →  succslist = { t | s -a-> t };
                                 for t in succslist do
                                   if CGG(t,ψ) = Red then
                                     color := Red ; casebreak
                                 color := Green
                κ(s,φ) := color ;
                COLOR-BACKWARDS(π(s,φ), color);
                return color
```

Abbildung 2.8: Der Algorithmus **CGG**

```
COLOR-BACKWARDS(P, color) =
if      P = ∅
then    return
else    choose (t, ψ) ∈ P;
        P  :=  P \ {(t, ψ)};
        if      κ(t, ψ) ≠ color
        then    κ(t, ψ) := undefined ;
                CGG(t, ψ) ;
                COLOR-BACKWARDS(P ∪ π(t, ψ) , color)
        else    COLOR-BACKWARDS(P, color)
```

Abbildung 2.9: Die Unterfunktion **COLOR-BACKWARDS**

Ist die Farbe des besuchten Knoten *White*, so hat der Algorithmus einen Zykel gefunden. Er trägt den Knoten, der in der Variable `pred` verzeichnet wurde, als unmittelbaren Vorgänger des aktuellen Knotens ein und gibt die Farbe zurück, die der Initialisierung der Fixpunktberechnung in diesem Zykel entspricht, also *Red* im Falle eines μ- und *Green* im Falle eines ν-Zykels.[14]
In allen anderen Fällen ist der aktuelle Knoten noch nicht besucht worden und wir unterscheiden das weitere Verhalten aufgrund der Formelkomponente φ des Knotens:

- $\varphi = \mathtt{tt}$ oder $\varphi = \mathtt{ff}$: Offensichtlich müssen diese Knoten *Green* bzw. *Red* sein.
- $\varphi = \varphi_1 \vee \varphi_2$ oder $\varphi = \varphi_1 \wedge \varphi_2$: Zuerst ruft sich der Algorithmus rekursiv mit dem „linken“ Sohn, also mit der Formel φ_1 auf. Ist diese Farbe bestimmend, also *Green* im $\vee$-Fall, bzw. *Red* im $\wedge$-Fall, dann kann der aktuelle Knoten selbst genauso gefärbt werden. Ansonsten erhält er dieselbe Farbe wie der „rechte Sohn“.
- $\varphi = X$ oder $\varphi = \sigma X.\psi$: In solchen Fällen ist die Farbe des aktuellen Knotens gleich der Farbe des einzigen Nachfolgers.
- $\varphi = \langle a \rangle \psi$ oder $\varphi = [a]\psi$: Diese Fälle werden ähnlich den $\vee$- und $\wedge$-Fällen behandelt. Sobald ein bestimmender Nachfolger gefunden wurde, liegt die Farbe des aktuellen Knotens fest. Wurde z. B. im $\langle a \rangle \psi$-Fall kein Nachfolger mit der Farbe *Green* gefunden, so muß der Knoten offensichtlich selbst auch mit *Red* markiert werden.

Bevor die ermittelte Farbe in der Rekursion nach oben gereicht werden kann, muß noch dafür gesorgt werden, daß eine Fehlfärbung, wie sie in Abb. 2.7 dargestellt ist, nicht bestehen bleibt. Dies wird dadurch gewährleistet, daß die Funktion **COLOR-BACKWARDS** aus Abb. 2.9 mit der Menge der unmittelbaren Vorgänger des aktuellen Knotens und dessen Farbe aufgerufen wird.

COLOR-BACKWARDS bearbeitet Teilgraphen, in denen entweder nur grüne Knoten rot oder nur rote Knoten grün gefärbt werden müssen. Dazu erhält die Funktion eine Menge von Knoten, deren korrekte Färbung zu testen ist. Ist die Menge leer, so wird die Funktion beendet. Ansonsten werden Knoten, deren Färbung bereits korrekt ist, d.h. die

[14] Dies wird durch die Funktion `CYCLETYPE-COLOR` gewährleistet.

dieselbe Farbe haben wie der Parameter der Funktion angibt, aus der Menge entfernt. Um zu ermittlen, ob die anderen Knoten korrekt sind, oder umgefärbt werden müssen, ruft **COLOR-BACKWARDS** wiederum die Funktion **CGG** an diesen Knoten auf.

2.3.2 Terminierung

Satz 2.3.1 Sei $\mathcal{T} = (S, T, Acts, \lambda)$ und $\mathcal{G}_{\mathcal{T}}(s, \varphi) = (V, E)$ ein Gamegraph mit $s \in S$ und φ alternierungsfrei. Dann terminiert der Algorithmus **CGG** auf der Eingabe (s, φ).

Beweis: Die Terminierung ist gewährleistet, da jeder Knoten höchstens zweimal gefärbt wird. Offensichtlich werden Knoten, die nicht auf einem Zykel liegen, nur ein einziges Mal gefärbt, weil sie niemals von der Funktion **COLOR-BACKWARDS** besucht werden. Wir betrachten also einen Knoten, der auf einem Zykel liegt. Im schlimmsten Fall ergibt sich die Situation, die in Abb. 2.10 dargestellt ist. Es gilt z. B.:[15]

$$\kappa(v_1) = Red, \;\; \kappa(v_2) = Green$$

Beide Knoten v_1 und v_2 sind also bereits einmal gefärbt worden. **COLOR-BACKWARDS** färbt v_1 dann noch einmal durch Aufruf von **CGG**, dieses Mal mit der Farbe *Green*. Dabei wird v_2 nicht noch einmal gefärbt, weil **CGG** gefärbte Knoten als korrekt gefärbt annimmt. Dies läßt sich fortsetzen mit dem Vorgänger von v_1 auf dem Zykel usw., bricht jedoch spätestens bei v_2 ab, wenn der gesamte Zykel „rückwärts“ gefärbt wurde, weil v_2 bereits mit *Green* gefärbt ist.
Der Zykel, auf dem alle Knoten höchstens zweimal gefärbt wurden, zusammen mit den Knoten auf dem Weg zu diesem Zykel, die genau einmal gefärbt wurden, bilden eine Gamegraphkomponente nach Lemma 2.2.16.
Die Knoten dieser Gamegraphkomponente werden später nicht noch einmal gefärbt, weil sie per Induktion über den Index der Gamegraphkomponente von **CGG** als korrekt gefärbt angesehen werden, falls ein weiterer Weg in diese Komponente existiert.

Zuletzt bleibt noch zu zeigen, daß kein Knoten unendlich oft nur besucht wird. Ein bereits gefärbter Knoten kann nicht unendlich oft besucht werden, weil seine Färbung die Färbung seiner Vorgänger bedingt. Auf jedem Pfad, der zu einem gefärbten Knoten führt, gibt es also nach endlicher Zeit gefärbte Vorgänger. Da der Algorithmus im wesentlichen wie eine Tiefensuche arbeitet, wird der Knoten nicht mehr über diesen Pfad besucht. Da in einem endlichen Graphen aber nur endlich viele Pfade zu jedem Knoten existieren können, kann kein Knoten unendlich oft besucht werden. Ein noch nicht gefärbter Knoten kann auch nicht unendlich oft besucht werden, weil er beim ersten Besuch durch die Funktion **CGG** gefärbt wird. □

2.3.3 Korrektheit

Satz 2.3.2 Sei $\mathcal{T} = (S, T, Acts, \lambda)$ und $\mathcal{G}_{\mathcal{T}}(s, \varphi) = (V, E)$ ein Gamegraph mit $s \in S$ und φ alternierungsfrei. Dann berechnet der Algorithmus **CGG** eine korrekte Färbung auf einem Teilgraph von $\mathcal{G}_{\mathcal{T}}(s, \varphi)$.

[15] oder auch mit umgekehrter Färbung

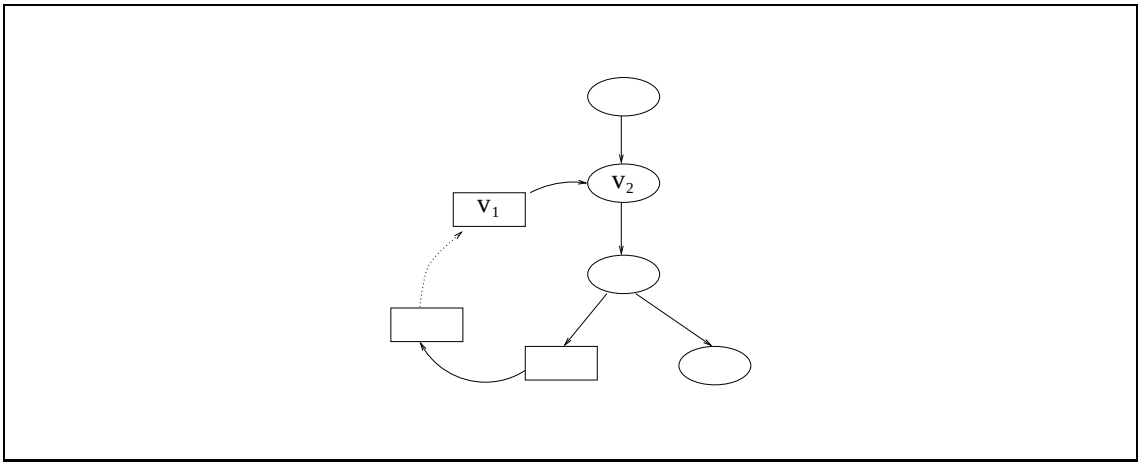

Abbildung 2.10: Die Terminierung des Algorithmus.

Beweis: Wir zeigen die Korrektheit des Algorithmus **CGG** mithilfe von Lemma 2.2.21, welches besagt, daß es fast ausreicht, lokal korrekte Färbungen zu erzeugen, um den gesamten Gamegraphen korrekt zu färben. Dabei bedeutet „lokal korrekt“ korrekt in Bezug auf die unmittelbaren Nachfolger eines Knotens.
Wir betrachten zuerst die Fälle, in denen der Algorithmus einen Knoten v noch nicht besucht hat, und unterscheiden diese bezüglich der Formelkomponente eines Knotens. Dies entspricht in Abb.2.8 dem `otherwise`-Fall der ersten `case`-Verzweigung. In diesen Fällen ist der Gamegraph ein Baum, und somit läßt sich der Beweis durch Induktion über den Aufbau des Gamegraphen führen.
Sei $t \in S$. $\mathbf{CGG}(\mathrm{v})$ sei im folgenden die Färbung, die der Algorithmus dem Knoten v zuweist, während $\kappa(v)$ die Färbung sei, die der Knoten laut Lemma 2.2.21 erhalten muß. Zu zeigen ist also:

$$\forall v \in V : \ \mathbf{CGG}(v) = \kappa(v)$$

- $v = (t, \mathtt{tt}) : \mathbf{CGG}(v) = \mathit{Green} = \kappa(v)$.
- $v = (t, \mathtt{ff}) : \mathbf{CGG}(v) = \mathit{Red} = \kappa(v)$.
- $v = (t, \sigma X.\psi) : \mathbf{CGG}(v) = \mathbf{CGG}(t, \psi) \stackrel{\text{I.V.}}{=} \kappa(t, \psi) = \kappa(v)$.
- $v = (t, X)$: Sei $\psi = \sigma X.\chi \in \mathit{Sub}(\varphi)$. $\mathbf{CGG}(v) = \mathbf{CGG}(t, \psi) \stackrel{\text{I.V.}}{=} \kappa(t, \psi) = \kappa(v)$.
- $v = (t, \psi_1 \vee \psi_2)$:
  - $\mathbf{CGG}(v) = \mathit{Green} \ \Leftrightarrow \ \mathbf{CGG}(t, \psi_1) = \mathit{Green}$ oder $\mathbf{CGG}(t, \psi_1) = \mathit{Red}$ und $\mathbf{CGG}(t, \psi_2) = \mathit{Green} \stackrel{\text{I.V.}}{\Leftrightarrow} \kappa(t, \psi_1) = \mathit{Green}$ oder $\kappa(t, \psi_1) = \mathit{Red}$ und $\kappa(t, \psi_2) = \mathit{Green} \ \Rightarrow \ \exists i \in \{1, 2\} \ \kappa(t, \psi_i) = \mathit{Green} \ \Leftrightarrow \ \kappa(v) = \mathit{Green}$
  - $\mathbf{CGG}(v) = \mathit{Red} \ \Leftrightarrow \ \mathbf{CGG}(t, \psi_1) = \mathit{Red}$ und $\mathbf{CGG}(t, \psi_2) = \mathit{Red} \stackrel{\text{I.V.}}{\Leftrightarrow} \kappa(t, \psi_1) = \mathit{Red}$ und $\kappa(t, \psi_2) = \mathit{Red} \ \Leftrightarrow \ \kappa(v) = \mathit{Red}$

- $v = (t, \psi_1 \wedge \psi_2)$:
  - $\mathbf{CGG}(v) = \mathit{Red} \Leftrightarrow \mathbf{CGG}(t, \psi_1) = \mathit{Red}$ oder $\mathbf{CGG}(t, \psi_1) = \mathit{Green}$ und $\mathbf{CGG}(t, \psi_2) = \mathit{Red} \overset{\text{I.V.}}{\Leftrightarrow} \kappa(t, \psi_1) = \mathit{Red}$ oder $\kappa(t, \psi_1) = \mathit{Green}$ und $\kappa(t, \psi_2) = \mathit{Red} \Rightarrow \exists i \in \{1, 2\}\ \kappa(t, \psi_i) = \mathit{Red} \Leftrightarrow \kappa(v) = \mathit{Red}$
  - $\mathbf{CGG}(v) = \mathit{Green} \Leftrightarrow \mathbf{CGG}(t, \psi_1) = \mathit{Green}$ und $\mathbf{CGG}(t, \psi_2) = \mathit{Green} \overset{\text{I.V.}}{\Leftrightarrow} \kappa(t, \psi_1) = \mathit{Green}$ und $\kappa(t, \psi_2) = \mathit{Green} \Leftrightarrow \kappa(v) = \mathit{Green}$
- $v = (t, \langle a \rangle \psi)$: $\mathbf{CGG}(v) = \mathit{Green} \Leftrightarrow \exists t' \in S\ \mathbf{CGG}(t', \psi) = \mathit{Green} \overset{\text{I.V.}}{\Leftrightarrow} \exists t' \in S\ \kappa(t', \psi) = \mathit{Green} \Leftrightarrow \kappa(v) = \mathit{Green}$
- $v = (t, [a] \psi)$: $\mathbf{CGG}(v) = \mathit{Red} \Leftrightarrow \exists t' \in S\ \mathbf{CGG}(t', \psi) = \mathit{Red} \overset{\text{I.V.}}{\Leftrightarrow} \exists t' \in S\ \kappa(t', \psi) = \mathit{Red} \Leftrightarrow \kappa(v) = \mathit{Red}$

Jetzt bleibt noch zu zeigen, daß der Algorithmus auch Gamegraphen mit Joinknoten richtig färbt. Dies gilt, weil die Funktion **COLOR-BACKWARDS** aufgerufen wird, *bevor* eine Komponente des Gamegraphen von der Funktion **CGG** verlassen wird.

Nehmen wir also an, der Algorithmus erreicht einen bereits gefärbten Knoten v, dessen Farbe nicht korrekt ist. O.B.d.A. sei $\mathbf{CGG}(v) = \mathit{Red}$, aber $\kappa(v) = \mathit{Green}$. Dann hat $\exists \mathit{loise}$ von v aus eine Gewinnstrategie. Somit existiert ein Weg von v zu einem Knoten w, für den gilt:

$$\kappa(w) = \mathit{Green}$$

Dann sind zwei Fälle zu unterscheiden:
1. Fall: w liegt nicht auf einem Zykel. Dann gilt offensichtlich auch $\mathbf{CGG}(w) = \mathit{Green}$.
2. Fall: w liegt auf einem Zykel. Entweder der Zykel ist geschlossen, dann muß er vom Typ ν sein, weil ansonsten kein Knoten darauf mit *Green* gefärbt worden wäre. Da v aber bereits gefärbt ist, muß v auch auf diesem Zykel liegen. Damit kann $\mathbf{CGG}(v) = \mathit{Red}$ aber nicht gelten. Oder der Zykel hat einen Ausweg, dann muß dort wegen der Endlichkeit aller Pfade in einem Gamegraphen ein Knoten existieren, auf den der 1. Fall zutrifft.
Wir können also annehmen, daß für den Knoten w ebenfalls gilt:

$$\mathbf{CGG}(w) = \mathit{Green}$$

Damit ist zumindest der Knoten w korrekt gefärbt. Wir zeigen nun, daß dann auch alle Knoten auf dem Pfad von v bis w korrekt gefärbt sind. Dazu betrachten wir den vorletzten Knoten u auf dem Pfad

$$v \rightarrow \ldots \rightarrow u \rightarrow w$$

1. Fall: u ist kein Zykelknoten. Dann ist wegen der korrekten Färbung von w aber auch nach dem ersten Teil des Beweises die Färbung von u korrekt.
2. Fall: u ist ein Zykelknoten. Dann ist u ebenfalls korrekt gefärbt worden, weil u als Vorgänger von w markiert wurde ($u \in \pi(w)$), und **COLOR-BACKWARDS** *nach* dem korrekten Färben von w mit dem Argument $\pi(w)$ aufgerufen wurde. **COLOR-BACKWARDS** hat dann wiederum $\mathbf{CGG}(u)$ aufgerufen, was aufgrund der korrekten Färbung von w die Färbung von u korrigiert hat, falls sie nicht korrekt war.
Mit der Iteration auf diesem Pfad folgt dann, daß alle Knoten einschließlich v korrekt gefärbt waren. □

2.3.4 Komplexität

Satz 2.3.3 Sei $\mathcal{T} = (S, T, Acts, \lambda)$ mit $s \in S$ und $\varphi \in \mathcal{L}^1_\mu$. Dann ist die **Zeitkomplexität** des Algorithmus mit Eingabe $(\mathcal{T}, s, \varphi)$

$$t_{CGG}(|\mathcal{T}|, |\varphi|) \;=\; O(|\mathcal{T}| \cdot |\varphi| \cdot log(|\mathcal{T}| \cdot |\varphi|))$$

Beweis: Wie in dem Beweis von Satz 2.3.1 bemerkt, wird jeder Knoten des Gamegraphen höchstens zweimal gefärbt. Aufgrund der Lokalität des Algorithmus müssen die Knoten des Gamegraphen in einer Datenstruktur beliebiger Größe verwaltet werden, in die sie eingefügt und wieder ausgelesen werden können.

$$\begin{aligned} t_{CGG}(|\mathcal{T}|, |\varphi|) \;\leq\; & |S| \cdot |\varphi| \cdot 2 \cdot t_{Färbung} \;+ \\ & |S| \cdot |\varphi| \cdot t_{einfügen} \;+ \\ & |S| \cdot |\varphi| \cdot t_{auslesen} \end{aligned}$$

Man kann die Zeit, die das Färben eines einzelnen Knoten in Anspruch nimmt, als konstant ansehen, weil die Färbung als Flag an dem entsprechenden Knoten gespeichert werden kann.

$$t_{Färbung} \;=\; const$$

Wird als angesprochene Datenstruktur z. B. ein höhenbalancierter Baum ([CLR92]) benutzt, dann ergeben sich folgende Zeitschranken für das Einfügen und Auslesen eines Knotens:

$$\begin{aligned} t_{einfügen}(n) &= O(log\, n) \\ t_{auslesen}(n) &= O(log\, n) \end{aligned}$$

Und mit der Abschätzung

$$|\mathcal{T}| \leq |S| + |T|$$

ergibt sich insgesamt folgende Zeitkomplexität für den Algorithmus **CGG**:

$$t_{CGG}(|\mathcal{T}|, |\varphi|) \;=\; O(|\mathcal{T}| \cdot |\varphi| \cdot log(|\mathcal{T}| \cdot |\varphi|))$$

□

Satz 2.3.4 Sei $\mathcal{T} = (S, T, Acts, \lambda)$ mit $s \in S$ und $\varphi \in \mathcal{L}^1_\mu$. Dann ist die **Platzkomplexität** des Algorithmus mit Eingabe $(\mathcal{T}, s, \varphi)$

$$s_{CGG}(|\mathcal{T}|, |\varphi|) \;=\; O(|\mathcal{T}| \cdot |\varphi|)$$

Beweis: Im schlimmsten Fall muß der gesamte Gamegraph, d.h. $\leq |S| \cdot |\varphi|$ in der verwendeten Datenstruktur gespeichert werden. Zusätzlich muß der Algorithmus sich für jeden Knoten die Färbung und die Menge der Vorgänger merken. Es gilt also:

$$s_{CGG}(|\mathcal{T}|, |\varphi|) \;\leq\; |S| \cdot |\varphi| \cdot (s_{Färbung} + max\{\; |\pi(s, \varphi)| \;\mid\; s \in S \;\})$$

Da aber die Menge der Vorgänger eines Knotens dadurch beschränkt ist, daß nur eine bestimmte Unterformel von φ die Formelkomponente des Vorgängers sein kann, und zwei Knoten im Gamegraph nur dann verbunden sein können, wenn die Zustandskomponenten im Transitionssytem verbunden sind, gilt für jedes $s \in S$:

$$|\pi(s, \varphi)| \;\leq\; ingrad(s)$$

Da sich die Färbung mit konstantem Aufwand abspeichern läßt, gilt insgesamt:

$$s_{CGG}(|\mathcal{T}|, |\varphi|) \leq |S| \cdot |\varphi| \cdot (const + max\{ingrad(s) | s \in S\}) = O(|\mathcal{T}| \cdot |\varphi|)$$

□

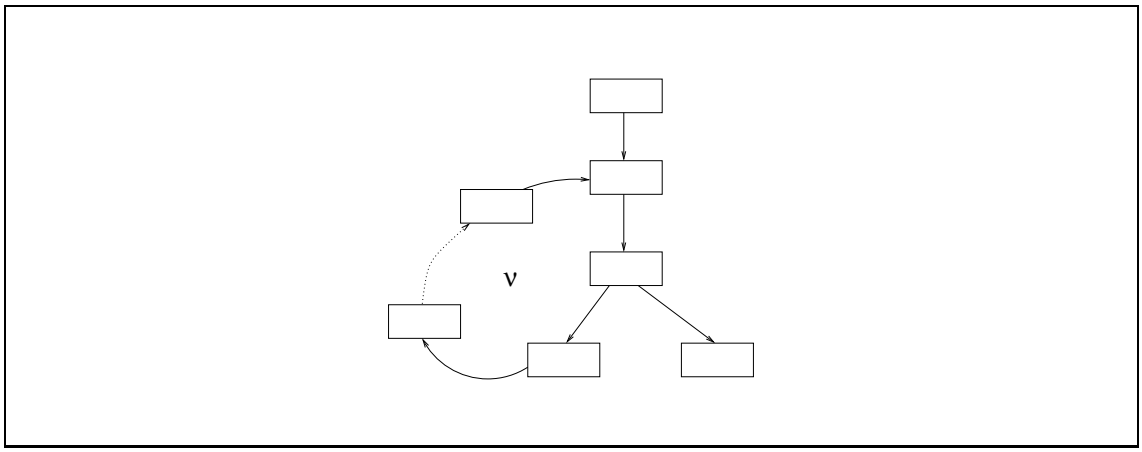

Abbildung 2.11: Einmaliger Zykeldurchlauf.

2.3.5 Ausbau zu $\mathcal{L}_\mu^\infty$

In [And94] zeigt Andersen, wie aus einem Algorithmus, der das Model Checking Problem für den alternierungsfreien μ-Kalkül in Zeit $O((|S| + |T|) \cdot |\varphi|)$ löst, ein Algorithmus für den vollen μ-Kalkül gewonnen werden kann, der in Zeit $O(|S|^{ad(\varphi)-1} \cdot |T| \cdot |\varphi|^{ad(\varphi)})$ läuft. Da der Algorithmus **CGG** aufgrund seiner Lokalität eine höhere Laufzeit als die geforderte hat, ist dieses Resultat nicht unbedingt interessant für die vorliegende Arbeit. Andererseits hätte eine globale Variante des Algorithmus **CGG** die geforderte Laufzeit und könnte somit zu einem Algorithmus für $\mathcal{L}_\mu^\infty$ mit Andersens Beweis erweitert werden.

2.4 Interaktive Model Checking Spiele

Der spielbasierte Ansatz erlaubt anzuzeigen, warum eine Formel durch ein Transitionssystem nicht erfüllt wird, falls sie nicht erfüllt wird. Dazu zeigen wir, wie sich aus einem Gamegraphen leicht ein Baum gewinnen läßt, dessen endliche Pfade die möglicherweise unendlichen Gewinnpartien von $\forall$*belard* repräsentieren. Es reicht nicht, im Zuge eines Model Checking Spiels den Gamegraphen zu durchlaufen und an Knoten, in denen $\forall$*belard* wählt, einen beliebigen Nachfolger zu suchen, der mit *Red* gefärbt wurde. Wie Abb. 2.11 zeigt, kann es z. B. ν-Zykel geben, die aufgrund eines Pfades, der aus dem Zykel hinausführt, mit *Red* gefärbt sind. In einem **interaktiven Model Checking Spiel** wollen wir diesen Zykel jedoch nicht beschreiten, weil das einmalige Durchlaufen eines Zykels das unendliche Durchlaufen und damit die Gewinnbedingung repräsentieren soll.

Aufgrund der völligen Dualität der Model Checking Spiele läßt sich die folgende Argumentation genauso verwenden, um zu zeigen, warum ein Transitionssystem eine Formel erfüllt statt nicht erfüllt, falls es sie erfüllt. Dies wurde bereits in Lemma 2.1.12 gezeigt. Im folgenden wollen wir ohne Einschränkung nur den Fall betrachten, daß $\forall$*belard* eine Gewinnstrategie für das betrachtete Spiel hat.

Def. 2.4.1 Sei $\mathcal{T} = (S, T, Acts, \lambda)$ mit $s, t \in S$ und $\varphi \in \mathcal{L}_\mu^\infty$. Eine **Pseudopartie** ist eine endliche Folge P von Konfigurationen

$$\mathcal{C}_0 \quad \rightarrow \quad \ldots \quad \rightarrow \quad \mathcal{C}_k$$

mit $\mathcal{C}_i \in \mathcal{C}_\mathcal{T}(s, \varphi)$ für alle $i \in \{0, \ldots, k\}$ und entweder

1. P ist eine endliche Partie des Spiels $\Gamma_{s,\mathcal{T}}(\varphi,)$, oder
2. es existiert ein $n < k$, so daß $\mathcal{C}_k = (t, \sigma X.\psi) = \mathcal{C}_n$ und für alle $j \in \{0, \ldots, n-1, n+1, \ldots, k-1\}$ gilt $\mathcal{C}_j \neq \mathcal{C}_k$.

Im zweiten Fall sagen wir, daß P die unendliche Partie

$$\mathcal{C}_0 \rightarrow \ldots \rightarrow \mathcal{C}_n \rightarrow \ldots \rightarrow \mathcal{C}_k \rightarrow \mathcal{C}_{n+1} \rightarrow \ldots \mathcal{C}_k \rightarrow \mathcal{C}_{n+1} \rightarrow \ldots$$

repräsentiert. ◇

Satz 2.4.2 Sei $\mathcal{G}_\mathcal{T}(s, \varphi) = (V, E, \kappa)$ ein durch den Algorithmus **CGG** korrekt gefärbter Gamegraph mit

$$\kappa(s, \varphi) \quad = \quad Red$$

Dann läßt sich eine Pseudopartie finden, die eine Gewinnpartie für $\forall belard$ präsentiert, indem $\forall belard$ in seinen Konfigurationen folgendermaßen zieht: Falls $\mathcal{C}_j$ derjenige Knoten unter den Nachfolgern von $\mathcal{C}_i$ ist, der von **CGG** zuletzt gefärbt wurde, dann macht $\forall belard$ den Zug $\mathcal{C}_i \rightarrow_I \mathcal{C}_j$.

Beweis: Zuerst bemerken wir, daß ein Nachfolger von $\mathcal{C}_i$ mit *Red* gefärbt sein muß, da $\kappa(\mathcal{C}_i) = Red$ nach Voraussetzung gilt und ein Knoten nur mit *Red* gefärbt sein kann, wenn

1. er keine Nachfolger hat, oder
2. mindestens einen Nachfolger hat, der ebenfalls mit *Red* gefärbt ist.

Im ersten Fall haben wir eine endliche Gewinnpartie für $\forall belard$ gefunden.
Gelte also Fall 2. Der zuletzt besuchte Nachfolgeknoten $\mathcal{C}_j$ von $\mathcal{C}_i$ ist sicherlich mit *Red* gefärbt, weil $\forall belard$ in $\mathcal{C}_i$ wählen durfte und deswegen *Red* die bestimmende Farbe dieses Knotens ist. **CGG** besucht Nachfolgeknoten jedoch nur solange, wie noch keine bestimmende Farbe gefunden wurde.
Falls $\mathcal{C}_j$ zwischendurch durch **COLOR-BACKWARDS** besucht wurde und letztlich mit *Green* gefärbt wurde, dann wird später **CGG** an $\mathcal{C}_i$ erneut aufgerufen, und es wird nochmals ein roter Nachfolger gesucht. Sollte letztendlich keiner existieren, dann kann $\mathcal{C}_i$ auch nicht mit *Red* gefärbt worden sein.
Es muß nun noch gezeigt werden, daß der Pfad über $\mathcal{C}_j$ nicht in einen ν-Zykel führt, um die Repräsentationsbedingung für die Pseudopartie zu erfüllen. Angenommen, dies ist der Fall. Dann können wir uns darauf beschränken, daß sowohl $\mathcal{C}_i$ als auch $\mathcal{C}_j$ auf diesem Zykel liegen. Sollte aus diesem Zykel kein Pfad herausführen, der die Farbe *Red* bestimmend zurückliefert, dann wurde $\mathcal{C}_j$ zuerst mit *Green* gefärbt, weil es sich um einen ν-Zykel handelt. Diese kann später in *Red* umgewandelt werden. Da $\mathcal{C}_i$ jedoch nach Voraussetzung ebenfalls auf dem Zykel liegt, wird dieser Knoten gefärbt, bevor **COLOR-BACKWARDS** den Zykel umfärben kann. Und somit kann $\mathcal{C}_j$ nicht der zuletzt besuchte Nachfolger von $\mathcal{C}_i$ gewesen sein. □

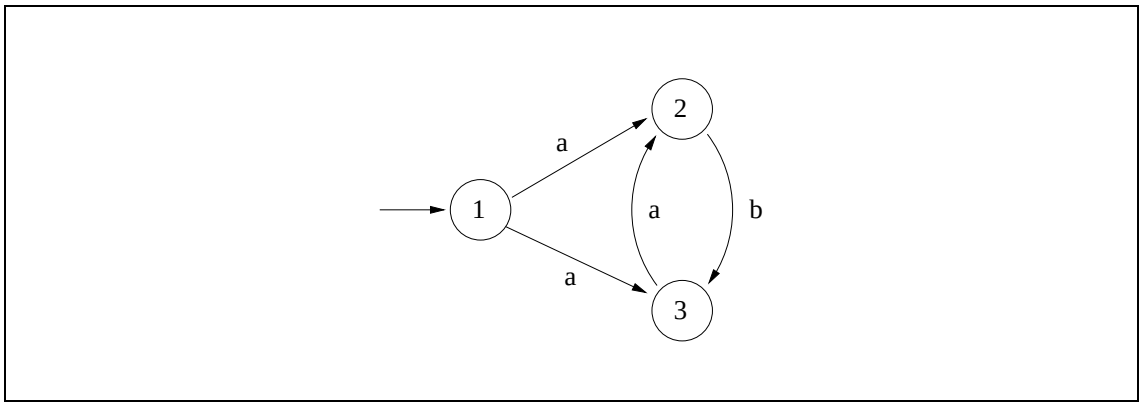

Abbildung 2.12: Ein weiteres Transitionssystem.

Def. 2.4.3 Sei $\mathcal{G}_{\mathcal{T}}(s,\varphi)$ ein Gamegraph. Der zugehörige **Spielbaum** $T_{s,\varphi}$ besteht dann aus dem einmaligen Abwickeln des Gamegraphen zu einem Baum.
Ein **Spielbaum für** P $T^P_{s,\varphi}$ besteht aus dem einmaligen Abwickeln des Gamegraphen $\mathcal{G}_{\mathcal{T}}(s,\varphi)$, wobei in Konfigurationen, in denen P wählt, nur der nach Satz 2.4.2 bestimmte, zuletzt besuchte Nachfolgerknoten, als Nachfolger im Spielbaum aufgenommen wird.

Bem. 2.4.4 Offensichtlich gilt dann: Die Pfade im Spielbaum (für P) sind genau die Pseudopartien, die die Partien repräsentieren, die durch Pfade im entsprechenden Gamegraphen dargestellt sind.

2.5 Ein Beispiel

Wir führen die Funktionsweise des Algorithmus **CGG** an einem übersichtlichen Beispiel vor.[16]

Bsp. 2.5.1 Sei $\varphi = \mu X.\langle b\rangle \mathtt{tt} \vee [-]X$ [17] und $\mathcal{T} = (S, T, Acts, \lambda)$ wie in Abb. 2.12 dargestellt. Der Startzustand des Transitionssystems sei 1. Dann sieht der zu φ und $\mathcal{T}$ gehörige Gamegraph $\mathcal{G}_{\mathcal{T}}(\varphi, s)$ aus, wie es in Abb. 2.13 dargestellt ist.

Bem. 2.5.2 Es gilt: $(\mathcal{T}, 1) \models \varphi$.

Da eine Tiefensuche-Strategie nicht genau festlegt, welche Pfade eines Graphen zuerst durchlaufen werden, kann die Arbeitsweise des Algorithmus verschieden ausfallen.[18]

[16] In Kap. 4 wird ebenfalls dieses Beispiel verwendet.

[17] φ besagt: Auf allen Pfaden tritt irgendwann einmal eine b-Aktion auf.

[18] Beachte, daß Abb. 2.13 lediglich eine Repräsentation des Gamegraphen darstellt. Der Model Checking Algorithmus kann die jeweiligen Nachfolger eines Knoten in einer anderen Reihenfolge verwalten, ohne daß sich festlegen läßt, welcher Sohnknoten in einer Tiefensuche der zuerst besuchte ist.

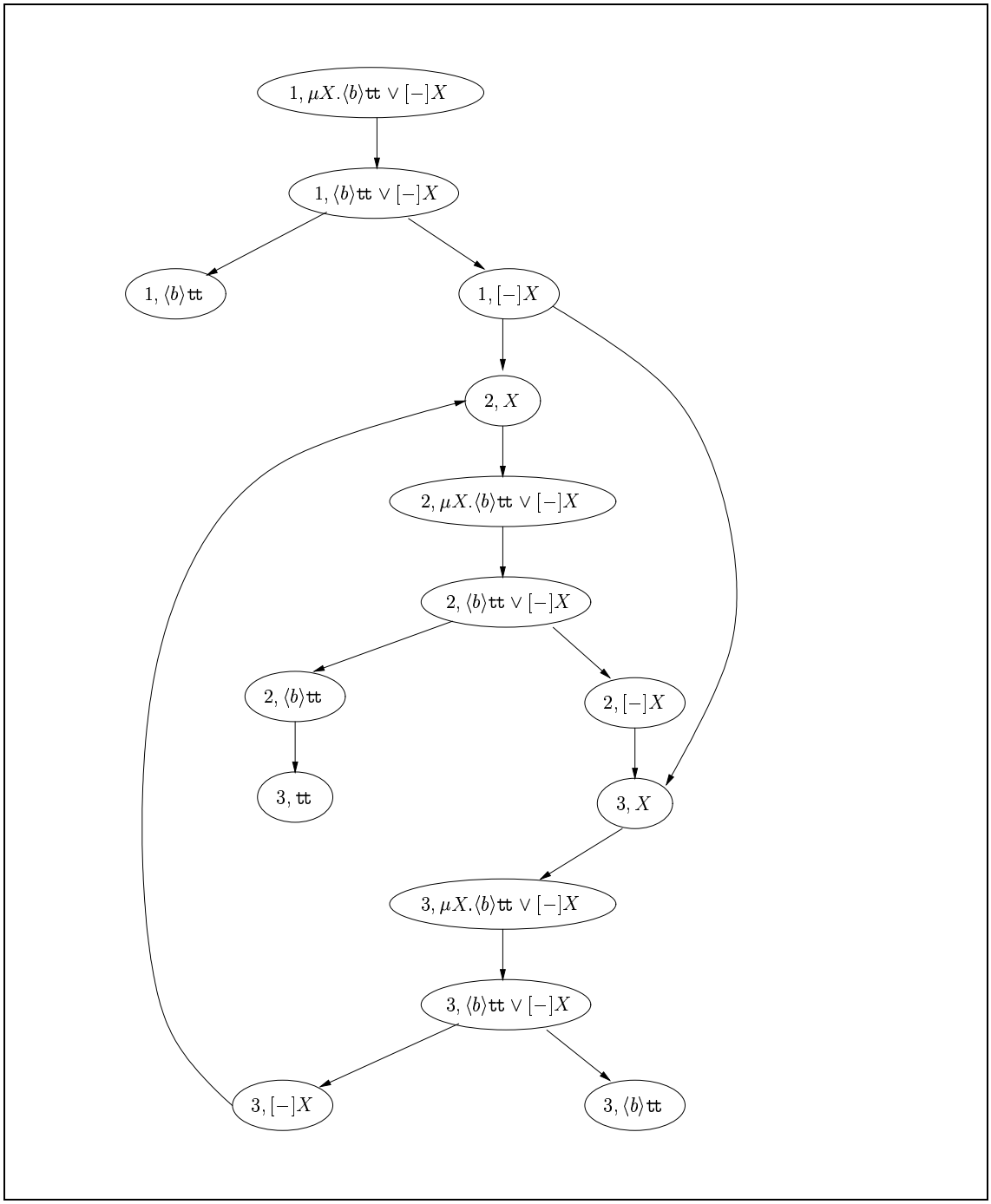

Abbildung 2.13: Der Gamegraph zu Bsp. 2.5.1.

Läuft der Algorithmus z. B. zuerst in den endlichen Pfad, der mit $(3, \mathtt{tt})$ endet, so wird die Funktion **COLOR-BACKWARDS** niemals aufgerufen, weil es nie zu einer Fehlfärbung kommt.
Damit das Beispiel nicht trivial ist, zeigen wir die Funktionsweise des Algorithmus **CGG** im „schlimmsten" und somit interessantesten Fall in diesem Beispiel. Um gefundene und gefärbte Knoten zu unterscheiden, verwenden wir ab jetzt folgende Konvention:

- Ein besuchter, aber noch nicht gefärbter Knoten erhält **keine** Umrandung.[19]
- Ein gefärbter Knoten erhält eine elliptische Umrandung, falls seine Farbe *Green* ist, ansonsten ist er rechteckig dargestellt.
- Nicht besuchte Knoten werden im Gamegraphen überhaupt nicht dargestellt.

CGG startet bei $(1, \varphi)$ und durchläuft dann die Knoten

$$(1, \langle b\rangle\mathtt{tt} \vee [-]X),\ (1, [-]X),\ (2, X),\ (2, \varphi),\ (2, \langle b\rangle\mathtt{tt} \vee [-]X),\ (2, [-]X),$$
$$(3, X),\ (3, \varphi),\ (3, \langle b\rangle\mathtt{tt} \vee [-]X),\ (3, [-]X),\ (2, X)$$

Hier erkennt **CGG**, daß der Knoten $(2, X)$ bereits besucht wurde und prüft, auf welcher Art von Zykel er sich befindet. Da es sich um einen Zykel vom Typ μ handelt, erhält der letzte Knoten $(3, [-]X)$ auf dem Zykel die Farbe *Red*. Außerdem wird dieser als Vorgänger von $(2, X)$ vermerkt.
Da nun eine Farbe festgelegt wurde, kann **CGG** diese in der Rekursion nach oben weiterreichen. Der zuvorletzt besuchte Knoten $(3, \langle b\rangle\mathtt{tt} \vee [-]X)$ ist eine Konfiguration, in der $\exists$*loise* die Wahl hat. Daher muß auch der andere Sohn besucht werden.
Dieser wird offensichtlich ebenfalls mit *Red* gefärbt, und dadurch kann die Farbe *Red* weiter nach oben gereicht werden, bis der Knoten $(2, \langle b\rangle\mathtt{tt} \vee [-]X)$ erreicht ist. Hier muß ebenfalls der andere Sohn besucht werden.

Dies endet jedoch in einem Pfad, der mit *Green* gefärbt werden muß, weil die letzte auftretende Formel $\mathtt{tt}$ ist. Die Farbe reicht sich über $(2, \langle b\rangle\mathtt{tt})$ nach oben zu $(2, \langle b\rangle\mathtt{tt} \vee [-]X)$. Da dieser Knoten eine Konfiguration ist, in der $\exists$*loise* wählt, hat sich hier *Green* gegenüber *Red* „durchgesetzt" und die Knoten $(2, \langle b\rangle\mathtt{tt} \vee [-]X)$, $(2, \varphi)$ und $(2, X)$ werden alle mit *Green* gefärbt.

Die aktuelle Situation ist in Abb. 2.14 dargestellt.

Würde diese Farbe jetzt nach zu $(1, [-]X)$ weitergereicht, so würde **CGG**, weil dies ein $\forall$*belard*-Knoten ist, den anderen Sohn $(3, X)$ besuchen und von dort die Farbe *Red* übernehmen. Man sieht leicht, daß diese sich bis zur Wurzel des Gamegraphen durchsetzen würde, woraus folgen würde, daß die Formel vom Transitionssystem nicht erfüllt wird.
Außerdem erkennt man eine Fehlfärbung durch *Red* am Knoten $(3, [-]X)$, dessen einziger Nachfolger $(2, X)$ mit *Green* gefärbt ist.

Um solche Fehler zu vermeiden, wird nun **zuerst** die Funktion **COLOR-BACKWARDS** mit der Farbe *Green* am Vorgänger $(3, [-]X)$ des Knoten $(2, X)$ aufgerufen. **COLOR-BACKWARDS** sorgt dafür, daß folgende Knoten umgefärbt werden:

[19] Im Algorithmus **CGG** wird solch ein Knoten mit der „Farbe" *White* markiert.

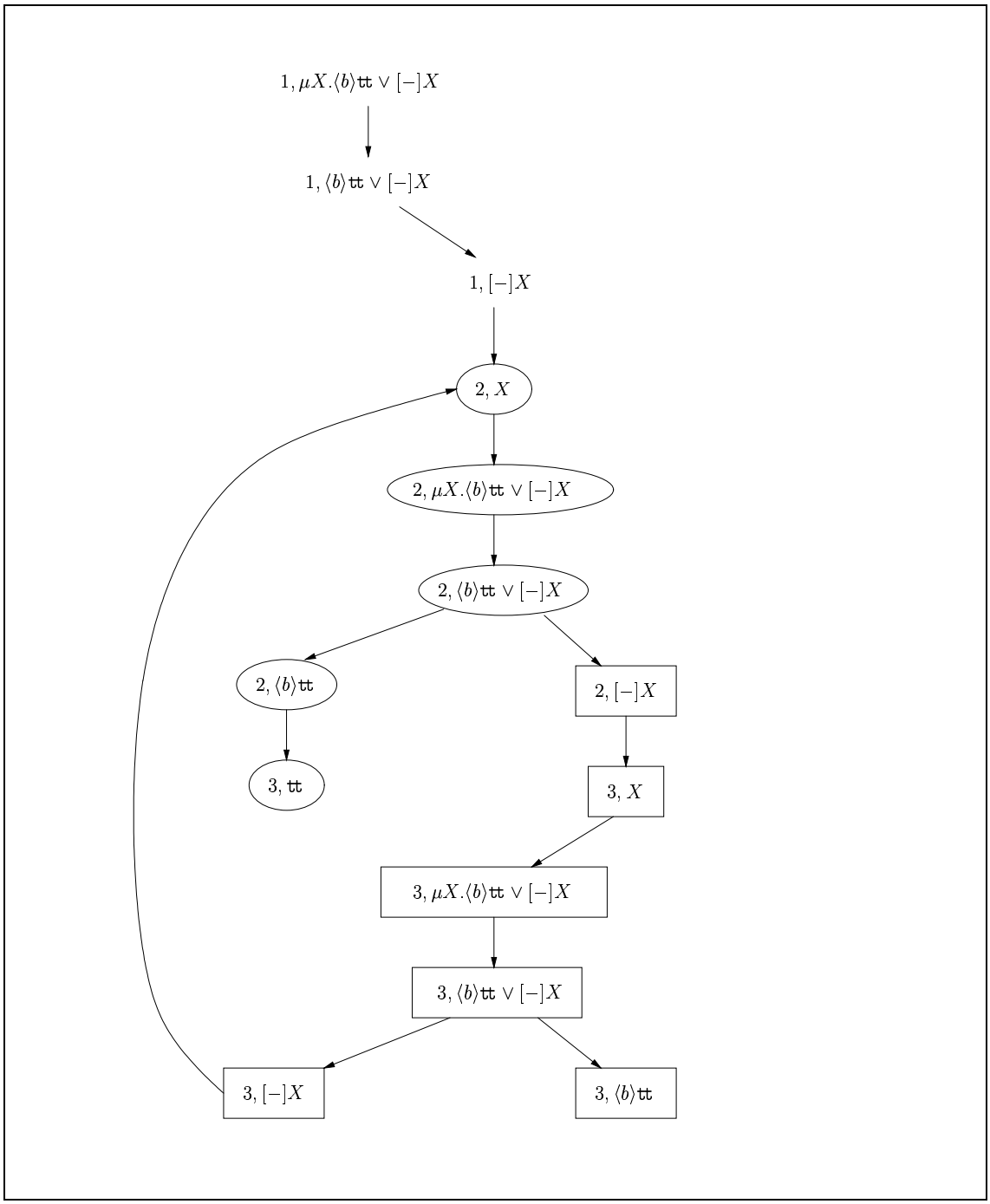

Abbildung 2.14: Die ersten Schritte im Färben des Gamegraphen.

$$(3, [-]X),\ \ (3, \langle b\rangle \mathtt{tt} \vee [-]X),\ \ (3, \varphi),\ \ (3, X),\ \ (2, [-]X)$$

Danach trifft **COLOR-BACKWARDS** auf $(2, \langle b\rangle \mathtt{tt} \vee [-]X)$, der bereits mit *Green* gefärbt ist, und somit ist das Rückwärtsfärben in diesem Zykel beendet.

Erst jetzt wird die Farbe *Green* vom Knoten $(2, X)$ weiter nach oben gereicht. Dort muß, weil $(1, [-]X)$ ein $\forall$*belard*-Knoten ist, auch der andere Sohn betrachtet werden. Dieser ist mittlerweile ebenfalls mit *Green* gefärbt worden, so daß sich die Farbe *Green* bis zur Wurzel durchsetzt, was wegen Bem. 2.5.2 zu erwarten war.

2.5.1 Die Gewinnpartien

Offensichtlich enthält der gefärbte Gamegraph aus Bsp. 2.5.1 unendlich viele Partien, die Gewinnpartien für $\exists$*loise* sind, weil jedes n-malige Durchlaufen des μ-Zykels, das letztendlich in der Konfiguration $(3, \mathtt{tt})$ endet, $\exists$*loise* gewinnen läßt. Aber nur zwei dieser Partien sind Pseudopartien nach Def. 2.4.1. Diese sind in Abb. 2.15 dargestellt.

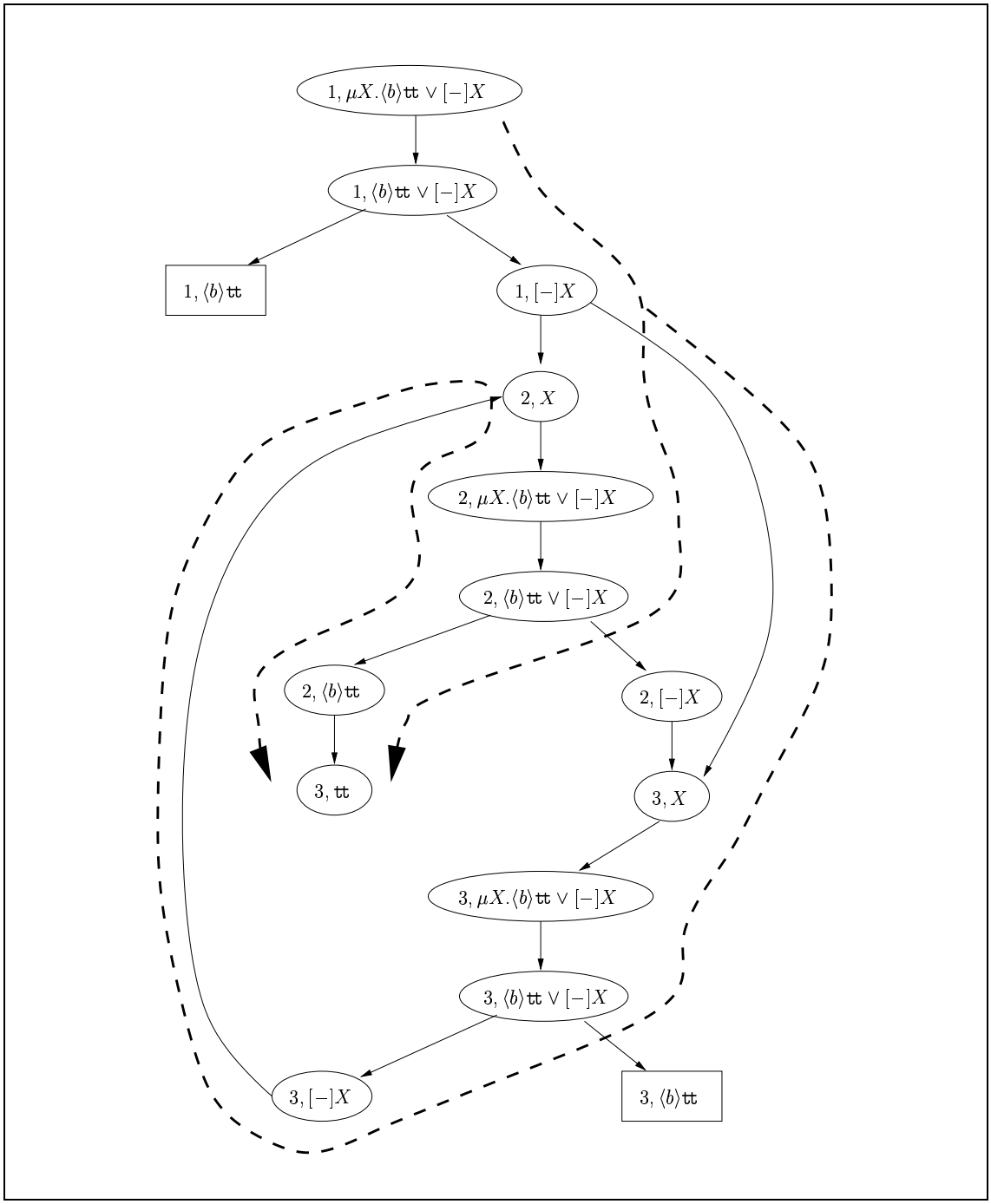

Abbildung 2.15: Die Gewinnpartien im gefärbten Gamegraphen.

3 Andere Model Checking Algorithmen für $\mathcal{L}_\mu^1$

Im folgenden werden wir einige andere deterministische Algorithmen, die das Model Checking Problem für den alternierungsfreien μ-Kalkül lösen, vorstellen und mit dem spielbasierten Algorithmus aus Kap. 2 vergleichen.

3.1 Der Algorithmus von Cleaveland und Steffen

Cleaveland und Steffen geben in [CS92] einen Model Checking Algorithmus für den alternierungsfreien μ-Kalkül an, der global ist, aber lineare Laufzeit hat. Das Verfahren ist im wesentlichen eine effiziente Implementierung des Algorithmus Fixpunktberechnung, den wir in Abschn. 1.5.2 vorgestellt haben.
Cleaveland und Steffen wandeln die zu untersuchende Formel in ein rekursives Gleichungssystem über μ-Kalkül-Variablen um. Wegen der Alternierungsfreiheit läßt sich dieses Gleichungssystem in sogenannte Blöcke partitionieren, in denen die Variablen voneinander abhängen und denselben Fixpunkttyp haben.

Bsp. 3.1.1 [1] Sei $\varphi = (\nu X.\mathtt{tt} \wedge [a]X) \vee (\mu Y.\mathtt{ff} \vee ([a]Y \wedge \langle a \rangle \mathtt{tt}))$. Dann besteht das zugehörige System von Gleichungen aus zwei Blöcken:

Block 2

$$\begin{array}{lll} X_1 & =_\nu & X_2 \vee X_3 \\ X_2 & =_\nu & X_4 \wedge X_5 \\ X_4 & =_\nu & \mathtt{tt} \\ X_5 & =_\nu & [a]X_2 \end{array}$$

Block 1

$$\begin{array}{lll} X_3 & =_\mu & X_6 \vee X_7 \\ X_6 & =_\mu & \mathtt{ff} \\ X_7 & =_\mu & X_8 \wedge X_9 \\ X_8 & =_\mu & [a]X_3 \\ X_9 & =_\mu & \langle a \rangle X_{10} \\ X_{10} & =_\mu & \mathtt{tt} \end{array}$$

Wir ersparen uns, die Semantik solcher Gleichungssysteme formal anzugeben, da die Gleichungen dadurch entstehen, daß man jeder Unterformel einer μ-Kalkül-Formel eine eigene Variable zuordnet, so daß die rechten Seiten jeder Gleichung entweder atomar oder eine einfache Disjunktion oder Konjunktion von zwei Variablen oder eine durch eine Modalität

[1] Aus [CS92]

bewachte Variable sind. Offensichtlich sollte eine Variable in einem Zustand eines Transitionssytems gültig sein, genau dann wenn die entsprechende Unterformel in diesem Zustand gilt.
In diesem Beispiel hängt der ν-Block von dem μ-Block wegen der Gleichung $X_1 = X_2 \vee X_3$ ab. Aufgrund der Alternierungsfreiheit ist es immer möglich, eine wechselseitige Abhängigkeit von Blöcken zu vermeiden. Und wegen der endlichen Größe der Formel und somit auch des Gleichungssystems ist gewährleistet, daß sich die Blöcke topologisch sortieren lassen. Seien B_i und B_j Blöcke eines Gleichungssystems. Dann gilt:

$$B_i \text{ hängt von } B_j \text{ ab } \Rightarrow \; i \geq j$$

Der erste Schritt im Algorithmus von Cleaveland und Steffen ist, die Blöcke topologisch zu sortieren, was sich in linearer Zeit bewerkstelligen läßt. Danach werden die einzelnen Blöcke in aufsteigender Reihenfolge ihrer Nummerierung nach bearbeitet. Dabei ist es unwesentlich, ob es sich um einen μ- oder einen ν-Block handelt, da diese gewissermaßen dual zueinander sind. Daher beschränken wir uns hier darauf, vorzustellen, wie ein μ-Block behandelt wird.

Cleaveland und Steffen schlagen vor, für jeden Zustand eines Transitionssystems ein Bitarray der Größe n anzulegen, wenn das zugrundeliegende Gleichungssystem die Variablen $X_1, \ldots, X_n$ enthält. Das i-te Bit des Arrays am Zustand s ist 1, genau dann wenn die Variable X_i im Zustand s gilt. Am Anfang werden alle Array-Einträge auf 0 gesetzt.[2] Der zweite Schritt besteht darin, alle Einträge, die zu Variablen gehören, deren rechte Regelseiten $\mathtt{tt}$ sind, auf 1 zu setzen.
Desweiteren merkt sich der Algorithmus in verschiedenen Listen diejenigen Paare (s, X_i), für die der Arraywert der Variablen X_i im zu s gehörenden Array noch nicht stabil im Sinne der Fixpunktberechnung ist. Am Anfang sind dies alle Paare, deren Werte bei der Initialisierung nicht explizit auf 1 gesetzt wurden. Sorgt man dafür, daß diese Paare in einer günstigen Reihenfolge in die Listen eingetragen werden, dann ist garantiert, daß jedes Paar höchstens einmal in bearbeitet wird, woraus die lineare Laufzeit des Algorithmus folgt.
Das Bearbeiten der Paare besteht darin, sich die entsprechende Gleichung anzusehen, um dann entweder zwei Bits aus den Arrays auszulesen und boolesch zu verknüpfen, oder im Transitionssystem die Nachfolger eines Zustandes zu berechnen und die Bits aus den Arrays der Nachfolgezustände auszulesen.

3.1.1 Vergleich zum Algorithmus CGG

Wie bereits oben erwähnt, entsprechen die Variablen, die Cleaveland und Steffen für die Gleichungssysteme einführen, den Unterformeln einer betrachteten μ-Kalkül-Formel. Die von ihnen vorgeschlagene Datenstruktur Transitionssystem mit Array an den einzelnen Knoten ist somit lediglich eine Repräsentation eines Gamegraphen.[3] Die Biteinträge der Arrays entsprechen der Färbung des Gamegraphen.

[2] Im Falle eines ν-Blocks werden die Einträge mit 1 initialisiert.

[3] Aufgrund der Art, wie das Verifikationstool TRUTH Transitionssysteme behandelt, haben wir exakt diese Repräsentation für die Implementierung des Algorithmus **CGG** gewählt. Siehe auch Kap. 4.

Bem. 3.1.2 Sei $\mathcal{T}= (S, T, Acts, \lambda)$mit $s \in S$ und ψ eine Unterformel der zu checkenden μ-Kalkül-Formel mit zugehöriger Variable X_i. Dann gilt:

$$s.X_i = 1 \quad \Leftrightarrow \quad \kappa(s, \psi) = Green$$

Die angesprochene geschickte Verwaltung der Listen im Algorithmus von Cleaveland und Steffen entspricht der Tiefensuche beim Färben eines Gamegraphen, die dafür sorgt, daß immer die Farben der Knoten als nächstes berechnet werden, deren Nachfolger schon sicher gefärbt sind. Und die Einteilung in Blöcke entspricht der Einteilung eines Gamegraphen in Zusammenhangskomponenten.
Der wesentliche Unterschied zwischen beiden Algorithmen liegt darin, daß der Algorithmus von Cleaveland und Steffen global ist. Da er eine Repräsentation eines gefärbten Gamegraphen berechnet, kann er höchstens bedingt Gegenbeispiel generierend genannt werden. Ersten fehlt eine theoretische Fundierung, wie ein Gegenbeispiel auszusehen hat, wie sie z. B. durch die Model Checking Spiele aus Kap. 2 gegeben ist. Zweitens ist die von Cleaveland und Steffen vorgeschlagene Logik mit Gleichungssystemen noch weniger geeignet, eine Eigenschaft eines Transitionssystems verständlich zu spezifizieren, als es der modale μ-Kalkül, wie wir ihn in Kap. 1 vorgestellt haben, ohnehin schon ist. Beide Nachteile sind jedoch nicht gravierend, sondern lediglich ein formaler Unterschied zu dem spielbasierten Ansatz.

3.2 Der Algorithmus von Andersen

In [And94] beschreibt Andersen einen globalen Algorithmus, der einen booleschen Graphen in Linearzeit gültig markiert. Er zeigt auch, daß sich dieser Algorithmus dazu eignet, das Model Checking Problem für den alternierungsfreien μ-Kalkül zu lösen. Genauso wie im Algorithmus von Cleaveland und Steffen wollen wir uns darauf beschränken, nur kleinste Fixpunktformeln zu betrachten, da jeder Algorithmus auf einfache Art und Weise verändert werden kann, um die dualen Fixpunktformeln zu verarbeiten. Ebenso lassen sich auch hier beide Varianten des Algorithmus kombinieren, um Model Checking für Formeln in $\mathcal{L}_\mu^1$ wie die aus Bsp. 3.1.1 durchzuführen.

Andersen beschreibt eine Methode, wie aus einem Transitionssystem und einer Formel des modalen μ-Kalküls ein boolescher Graph konstruiert werden kann. Dazu überführt er, ähnlich wie Cleaveland und Steffen in Abschn. 3.1 die Formel in ein Gleichungssystem mit mehreren Variablen, auf deren rechten Regelseiten nur Formeln mit höchstens einem Operator auftreten dürfen.

Def. 3.2.1 Ein **boolescher Graph** ist ein Graph $G = (V, E, L)$, wobei (V, E) ein gerichteter Graph im herkömmlichen Sinne ist und

$$L : V \to \{\vee, \wedge\}$$ ◇

Dadurch läßt sich ein boolescher Graph leicht als Schaltkreis auffassen, dessen Knoten die booleschen Schaltelemente AND bzw. OR sind. Wie man leicht sieht, läßt sich jeder Gamegraph, wie wir sie in Abschn. 2.2 eingeführt haben, als boolescher Graph auffassen. So

gilt z. B. für alle Knoten v eines Gamegraphen, an denen $\forall belard$ die Wahl hat, $L(v) = \wedge$. Umgekehrt gilt, daß die booleschen Graphen, wie Andersen sie aus Transitionssystemen und abgewandelten μ-Kalkül-Formeln konstruiert, Repräsentationen eines Gamegraphen sind.
Die Aufgabe des Algorithmus ist es, für einen gegebenen booleschen Graphen $G = (V, E, L)$ eine Markierung

$$m : V \to \{0, 1\}$$

zu finden, so daß für alle $v \in V$ gilt:

$$m(v) = \begin{cases} 1 & \text{falls } L(v) = \wedge \text{ und } \forall w \in V, \text{ wenn } (v,w) \in E \text{ dann } m(w) = 1 \\ & \text{oder } L(v) = \vee \text{ und } \exists w \in V, (v,w) \in E \text{ und } m(w) = 1 \\ 0 & \text{sonst} \end{cases}$$

Man sieht ebenfalls leicht, daß diese Markierung genau der korrekten Färbung des entsprechenden Gamegraphen entspricht.

Bem. 3.2.2 Sei $\mathcal{G}$ ein Gamegraph mit korrekter Färbung κ und $G = (V, E, L)$ der entsprechende boolesche Graph[4] mit korrekter Markierung m. Dann gilt für alle Knoten $v \in V$:

$$\kappa(v) = Green \;\Leftrightarrow\; m(v) = 1$$

Aufgrund dieses Zusammenhanges erkennt man außerdem, daß die korrekte Markierung eines booleschen Graphen wie im Falle der Gamegraphen schrittweise aufgebaut werden kann. Dazu schlägt Andersen vor, im Initialisierungsschritt seines Algorithmus alle Knoten mit 0 zu markieren.[5] Eine Ausnahme bilden natürlich diejenigen Knoten v, die keinen Nachfolger haben und für die gilt: $L(v) = \wedge$. Diese werden mit 1 markiert. Desweiteren merkt sich der Algorithmus für jeden Knoten die Anzahl der Nachfolger, die noch mit 1 markiert werden müssen, um den Knoten selbst mit 1 markieren zu können. Dies ist notwendig, um effizient entscheiden zu können, welcher Knoten als nächstes zu bearbeiten ist, damit der Algorithmus insgesamt noch in linearer Zeit läuft. Wiederum entspricht dies der Tiefensuchestrategie im Gamegraphen, die der Algorithmus **CGG** aus Abschn. 2.3 anwendet.

3.2.1 Vergleich zum Algorithmus CGG

Wie bereits erwähnt, arbeitet Andersens Algorithmus in sehr ähnlicher Weise wie der Algorithmus **CGG** aus Abschn. 2.3. Der wesentliche Unterschied besteht darin, daß Andersens Algorithmus global ist und, wie der Algorithmus von Cleaveland und Steffen, darauf angewiesen ist, daß die zu untersuchende Formel zuerst umgewandelt wird. Deswegen kann man Andersens Algorithmus auch nur bedingt Gegenbeispiel generierend nennen.

[4] mit gleicher Knoten- sowie Kantenmenge
[5] Beachte, daß wir uns hier auf kleinste Fixpunktformeln beschränken.

3.2.2 Eine lokale Variante

In [And94] gibt Andersen noch eine lokale Variante seines Algorithmus an, die hauptsächlich auf zwei Dingen beruht.

- Man betrachtet die Nachfolger eines Knoten, die in gewisser Reihenfolge gegeben sind, nur solange, bis ein Nachfolgerknoten den Wert des aktuellen Knotens bestimmt. So reicht es z. B. im Falle eines Knotens v mit $L(v) = \vee$ einen Nachfolger zu finden, dessen Wert 1 ist.
- Aufgrund des ersten Punktes folgt Andersens lokaler Algorithmus somit im generellen ebenfalls einer Tiefensuchestrategie. Diese kann jedoch in Kombination mit Joinknoten und Zykelknoten zu Problemen führen, wie wir in Abschn. 2.2.3 gezeigt haben. Daher merkt der Algorithmus sich bestimmte Vorgängerknoten, deren Wert eventuell nachträglich abgeändert werden muß.

Der Unterschied zwischen Andersens lokalem Algorithmus und dem Algorithmus **CGG** liegt darin, daß **CGG** die Tatsache ausnutzt, daß es ausreicht, sich an jedem Knoten lediglich die direkten Vorgänger zu merken, ohne eine Liste aufbauen zu müssen, deren Länge linear in der Größe des Gamegraphen sein kann.

3.3 Der Algorithmus von Bhat und Cleaveland

Das Verfahren von Bhat und Cleaveland, das sie in [BC96] angeben, unterscheidet sich von den anderen Algorithmen dadurch, daß sie Model Checking für den alternierungsfreien μ-Kalkül auf das Model Checking für die temporale Logik **LTL** [6] zurückführen. Dies ist kein trivialer Schritt, wie z. B. bei der Reduktion von **CTL** auf den μ-Kalkül, wie Abb. 1.4 zeigt. Es gibt Formeln in $\mathcal{L}^1_\mu$, die sich nicht in eine semantisch äquivalente Formel aus **LTL** übertragen lassen.

Die Reduktion von Bhat und Cleaveland benutzt sogenannte **And-Or Kripke-Strukturen**, die mit Andersens booleschen Graphen übereinstimmen. Sie geben Regeln, wie man aus einem Transitionssystem $\mathcal{T}$ mit Anfangszustand s und einer alternierungsfreien Formel φ eine And-Or Kripke-Struktur $\mathfrak{K}_{\mathcal{T},\varphi}$ mit Anfangszustand s', und eine **LTL**-Formel ψ an,[7] so daß gilt:

$$(\mathfrak{K}_{\mathcal{T},\varphi}, s') \models_{LTL} \psi \quad \Leftrightarrow \quad (\mathcal{T}, s) \models \varphi \tag{3.1}$$

Die Konstruktion der And-Or Kripke-Struktur erfolgt analog zu der Konstruktion eines Gamegraphen oder der Konstruktion eines booleschen Graphen aus Abschn. 3.2.
An der Stelle, an der die bisherigen Algorithmen den Graphen markiert oder gefärbt haben, tritt bei Bhat und Cleaveland das **LTL** Model Checking auf. Zur Erinnerung:

[6] s. Abschn. 1.4.2
[7] An dieser Stelle sei bemerkt, daß ψ unabhängig von ϕ ist.

Die Wurzel eines booleschen Graphen wird mit 1 markiert $\Leftrightarrow$ die Wurzel eines Gamegraphen wird mit *Green* gefärbt $\Leftrightarrow$ $\exists$*loise* gewinnt das entsprechende Spiel in diesem Gamegraphen $\Leftrightarrow$ $\exists$*loise* kann jede Partie entweder in eine Konfiguration, in der $\forall$*belard* nicht mehr ziehen kann, oder in eine Schleife, deren größte Fixpunktformel vom Typ ν ist, lenken. Letzteres ist im alternierungsfreien Fall äquivalent dazu, daß keine Fixpunktformel vom Typ μ in der Schleife auftritt.
Bhat und Cleaveland haben genau diese Bedingung in der Logik **LTL** formuliert. Obwohl wir **LTL** nicht formal definiert haben, wollen wir diese Formel dennoch explizit angeben:

$$\psi \quad = \quad \Diamond \mathit{true} \vee \Box \Diamond \nu$$

Interpretiert man diese Formel in geeigneter Weise über And-Or Kripke-Strukturen, wie es in [BC96] gezeigt wird, so besagt sie: Es gibt einen Lauf, der entweder in einer $\mathtt{tt}$- oder einer [ ]-Konfiguration endet oder unendlich oft eine ν-Konfiguration durchläuft. Ein Lauf ist dabei eine Abwicklung der And-Or Kripke-Struktur, die an jedem Or-Knoten nur einen Nachfolger und ansonsten alle Nachfolger durchläuft.

Benutzt man dann schließlich einen lokalen Model Checker für **LTL**, um die in Gl. 3.1 aufgezeigte Modellbeziehung zu überprüfen, so erhält man auf diese Weise einen lokalen Model Checking Algorithmus für den alternierungsfreien μ-Kalkül.
Im allgemeinen Fall ist Model Checking für **LTL** jedoch *PSPACE-vollständig*, d.h. vermutlich existiert kein Algorithmus, der dieses Problem mit weniger Ressourcen als polynomiellem Platzbedarf lößt. Prinzipiell könnte es zwar dennoch Algorithmen geben, die lineare Laufzeit haben, deren Existenz ist jedoch sehr unwahrscheinlich. Alle bisher bekannten Algorithmen, die Model Checking für **LTL** erlauben, haben eine Laufzeit, die exponentiell in der Länge der Formel ist.
Da die konstruierte **LTL**-Formel jedoch weder vom verwendeten Transitionssystem noch von der ursprünglichen μ-Kalkül-Formel abhängt, und somit nicht zur Eingabe an den μ-Kalkül Model Checking Algorithmus zählt, und die And-Or Kripke-Struktur in linearer Zeit aufgebaut werden kann, resultiert aus der gesamten Konstruktion ein linearer Model Checking Algorithmus für den alternierungsfreien μ-Kalkül.
Andererseits verhindert diese Konstruktion jedoch, daß der Algorithmus Gegenbeispiel generierend ist, solange das **LTL** Model Checking dies auch nicht ist.

Bhat und Cleaveland bleiben jedoch die Erklärung schuldig, wie ein lokaler Algorithmus, der einen Graphen *on demand* aufbaut, in linearer Zeit Zykel erkennen kann. Daher unterscheiden wir zwei Algorithmen: einen globalen, der in linearer Zeit läuft, und einen lokalen, dessen Komplexität $O(n \cdot log\, n)$ ist.

3.4 Vergleich der Algorithmen

Für die bisher betrachteten Algorithmen ergibt sich bezüglich der oben genannten Kriterien folgende Gegenüberstellung. Seien $\mathcal{T} = (S, T, Acts, \lambda)$ und φ die Eingaben für die jeweiligen Algorithmen.

Algorithmus	worst-case Laufzeit		lokal	Ggbsp. gener.
	allgemein	alternierungsfreier Fall		
Fixpunktelimination	$O(\|\mathcal{T}\| \cdot \|\varphi\| \cdot k^{\|S\|})$	$O(\|\mathcal{T}\| \cdot \|\varphi\| \cdot k^{\|S\|})$	nein	nein
Fixpunktberechung	$O(\|\mathcal{T}\| \cdot \|\varphi\|^{fd(\varphi)})$	$O(\|\mathcal{T}\| \cdot \|\varphi\|^{fd(\varphi)})$	nein	nein
Cleaveland/Steffen		$O(\|\mathcal{T}\| \cdot \|\varphi\|)$	nein	nein
Andersen 1		$O(\|\mathcal{T}\| \cdot \|\varphi\|)$	nein	nein
Andersen 2		$O(\|\mathcal{T}\| \cdot \|\varphi\| \cdot log(\|\mathcal{T}\| \cdot \|\varphi\|))$	ja	nein
Bhat/Cleaveland 1		$O(\|\mathcal{T}\| \cdot \|\varphi\|)$	nein	nein
Bhat/Cleaveland 2		$O(\|\mathcal{T}\| \cdot \|\varphi\| \cdot log(\|\mathcal{T}\| \cdot \|\varphi\|))$	ja	nein
CGG		$O(\|\mathcal{T}\| \cdot \|\varphi\| \cdot log(\|\mathcal{T}\| \cdot \|\varphi\|))$	ja	ja

4 Implementierung

Der Algorithmus aus Abschn. 2.3 wurde in dem Verifikationstool **Truth** ([Tru98]) implementiert. Eine allgemeine Übersicht über die aktuellen und zukünftigen Möglichkeiten für den Einsatz von TRUTH findet sich in [LLNT99]. TRUTH wurde am Lehrstuhl für Informatik II der RWTH Aachen im Rahmen der Diplomarbeit von Stephan Tobies entwickelt ([Tob98]).

TRUTH kann sowohl μ-Kalkül- als auch **CTL**-Formeln verarbeiten. Die Transitionssysteme ergeben sich als Semantik von Ausdrücken der Spezifikationssprache **CCS** ([Mil80, Mil89]). In der ersten Version von TRUTH konnte man automatisches Model Checking für den μ-Kalkül mithilfe des **Tableau-Algorithmus** ([Cle90]) durchführen. Dieser ist auch weiterhin ein Bestandteil von TRUTH, damit man immer noch Model Checking für Formeln mit echter Alternierung betreiben kann.
TRUTH wurde u.a. im Hinblick auf Wartbarkeit und Austauschbarkeit von Modulen in der nicht-strikten, rein funktionalen Sprache **Haskell** ([HPW+91, Tho96, Dav92, Bir98, HF92]) geschrieben. Dadurch kann man nicht die schnellste Implementierung eines Algorithmus, der auf Graphen arbeitet, erwarten. Da ein Ziel der Arbeiten am **Glasgow Haskell Compiler** ([HHP+92]) jedoch ist, eine Haskell-Implementierung zu entwickeln, die vergleichbar mit denen anderen Sprachen, wie z. B. **C**, ist, erscheint der Effizienznachteil nicht so gravierend, daß er die Vorteile einer Implementierung in einer funktionalen Sprache überwiegen würde ([Hug90]).

TRUTH ist in der Lage, Transitionssysteme auf effiziente Art und Weise zu handhaben. Mithilfe des **Monaden**-Konzepts ([JP93, Lau93, LJ94]) läßt sich in Haskell „imperativ“ programmieren, was die Handhabung von Graphen wesentlich vereinfacht, weil das Färben eines Graphen auf natürliche Weise einer zeitlichen Abfolge von Zuständen des Graphen entspricht.[1] Rein funktionale Programme bieten jedoch nicht die Möglichkeit, Zustände zu modellieren, weil die Semantik eines rein funktionalen Programms nicht von der Auswertungsreihenfolge einer Funktion abhängt, wie dies bei imperativen Sprachen der Fall ist. Das angesprochene Monaden-Konzept läßt dies jedoch auch innerhalb einer funktionalen Sprache zu.

4.1 Module

TRUTH wurde im Rahmen dieser Arbeit um zwei Module erweitert. Dabei handelt es sich um

[1] Vergleiche auch mit dem Algorithmus **CGG**, Abb. 2.8, der im Pseudocode sowohl funktionale als auch imperative Elemente enthält.

```
module MuGamePlay
(
  gamecheck,
  gameplay,
  cgg,

  GFormula(..),
  GameLabel(..)
)
```

Abbildung 4.1: Die exportierten Funktionen aus `MuGamePlay.lhs`.

- `MuGamePlay.lhs`: Dieses Modul beinhaltet alle wichtigen Datenstrukturen und Funktionen, die entweder Teil der Implementierung des Algorithmus **CGG** sind oder dazu dienen, den Benutzer interaktive Model Checking Spiele spielen zu lassen.
- `MuGameGraph.lhs`: Dieses Modul beherbergt die Funktionen, die zur Darstellung von Gamegraphen nötig sind.[2]

4.2 Funktionen und Datenstrukturen

4.2.1 Das Modul `MuGamePlay.lhs`

Die wichtigsten Funktionen des Moduls `MuGamePlay.lhs`, die exportiert werden, sind in Abb. 4.1 gegeben. Dabei werden folgende Datenstrukturen verwendet:

`GFormula`: Dieser Datentyp beschreibt eine μ-Kalkül-Formel φ, die aktuell verwendet wird. Zur einfacheren und schnelleren Handhabung wird die Formel, die ansonsten in TRUTH durch den Datentyp `MuFormula`[3] dargestellt wird, in ein Array umgewandelt. Die einzelnen Arrayeinträge entsprechen den Unterformeln von φ. Zu jeder Formel werden die potentiellen Nachfolgerformeln aus einem Model Checking Spiel über einen Integerwert vermerkt. Dieser ist der Arrayindex der nächsten, zu betrachtenden Formel.

```
data GFormula = GTrue |
                GFalse |
                GAnd Int Int |
                GOr Int Int |
                GBox Bool Bool [Action] Int |
                GDiam Bool Bool [Action] Int |
```

[2] Diese Funktionen sind weniger für den Benutzer von TRUTH als zur Hilfestellung bei der Implementierung des Algorithmus **CGG** gedacht.

[3] aus dem Modul `MuFormula.lhs`, s. [Tob98]

```
GLFP Id Int |
GGFP Id Int |
GAtom Id Int
```

`GameLabel`: Der Gamegraph wird, wie bereits in 3.1.1 erwähnt, folgendermaßen abgespeichert. Die zugrundeliegende Struktur ist das Transitionssystem, dessen effiziente Behandlung durch TRUTH bereits sichergestellt wird. An einem Knoten s werden alle Formeln ψ gespeichert, so daß die Konfiguration (s, ψ) vom Algorithmus **CGG** besucht wurde. Desweiteren werden dort die Färbung dieses Gamegraphknoten vom Typ `Color` sowie die Liste der Nachfolger im Gamegraphen vermerkt. Aus Effizienzgründen einer nichtstrikten, funktionale Sprache wird diese gesamte Information in einem höhenbalancierten Baum, einer sogenannten **FiniteMap** verwaltet.

```
data GameLabel    = GameLabel Bool GameColoring
type GameColoring = FiniteMap Int (Color, [(LTSState,Int)])
```

Die Färbung ist ein einfacher Aufzählungstyp:

```
data Color = None | White | Green | Red deriving (Eq,Show)
```

Dabei bedeutet `None`, daß ein Knoten noch nicht besucht, `White`, daß er bereits besucht, aber noch nicht gefärbt wurde, und `Green` sowie `Red`, daß er bereits eine Farbe erhalten hat.

Die wichtigsten Funktionen, die das Model `MuGamePlay.lhs` zur Verfügung stellt, sind:

`cgg`: Dies ist die Implementierung des Algorithmus **CGG** aus Abschn. 2.8. [4]

`gamecheck`: Diese Funktion (s. Abb. 4.2 bereitet den Aufruf der Funktion `cgg` vor, indem sie z. B. prüft, ob die zu checkende Formel alternierungsfrei ist. Schließlich wird das Ergebnis, das von `cgg` berechnet wurde, in Form eines booleschen Wertes `True` bzw. `False` zurückgegeben.[5]

`gameplay`: Diese Funktion (s. Abb. 4.3) bereitet das interaktive Spielen zwischen Benutzer und Computer vor. Dazu wird, genauso wie in `gamecheck`, zuerst die Alternierungstiefe der eingegebenen Formel berechnet. Sollte diese größer als 1 sein, dann ist ein Spiel nicht möglich. Ansonsten wird zuerst die Funktion `cgg` aufgerufen und anhand deren Rückgabewert entschieden, welchen Spieler TRUTH simuliert.[6]
Die Funktion `game_tree` wandelt schließlich den Gamegraphen in einen Baum um, der alle möglichen Pseudopartien des Spiels enthält. D. h. die Knoten dieses Baums haben als Nachfolger

[4] Da der Quellcode der Haskellimplementierung aufgrund der Länge unübersichtlich ist und eine Abstraktion, die alles wesentliche enthält und in Haskell-ähnlichem Pseudocode aufgeschrieben wurde, bereits in Abb. 2.8 gezeigt wurde, wollen wir auf die explizite Angabe des gesamten Quellcodes verzichten. Stattdessen sei auf den Sourcecode von TRUTH ([Tru98]) verwiesen, der Kommentare bzgl. der Implementierung des Algorithmus enthält.

[5] s. Abschn. 4.2.3.

[6] TRUTH ist immer der Gewinner des Model Checking Spiels, weil nur der Gewinner in der Lage ist, dem Spielpartner zu zeigen, warum eine Formel erfüllt oder nicht erfüllt ist.

```
gamecheck:: (Int,Int,Int) -> ProcEnv -> Process -> MuFormula -> Bool
gamecheck tablesizes penv proc f =
  let ad = alterDepth f in
  if ad > 1
    then error ("...")
    else
      let color = runLTS tablesizes defLabeling
                    ( getFreshStateLTS `thenLTS` \ st ->
                      setLabelLTS st proc `seqLTS`
                      addToStateHTblLTS proc st `seqLTS`
                      cgg penv st phi fpMap fixedPointMap formMap
                    )
      in
      case color of
           Green -> True
           Red   -> False
```

Abbildung 4.2: Die Funktion `gamecheck`.

- alle Nachfolger, die auch im Gamegraphen vorhanden sind, falls der Benutzer in dieser Konfiguration wählen darf, bzw.
- genau einen Nachfolger, falls TRUTH in dieser Konfiguration wählen darf. Dieser ist der nach Satz 2.4.2 eindeutig bestimmte Nachfolgeknoten.[7]

Damit entspricht die Ausgabe der Funktion `game_tree` einem Spielbaum für P, wobei P der Gewinner des betrachteten Spiels ist.

Das eigentliche Spiel wird dann von der Funktion `do_play`[8] durchgeführt.[9] Dazu bedient sie sich der Datenstruktur `GameMoves`, die die Konfigurationen des Spielbaums folgendermaßen unterscheidet:

```
data GameMoves = End |
                 Stucked |
                 Unfold Bool LTSState Int |
                 MyMove Bool LTSState Int |
                 NoChoice Bool LTSState Int |
                 Choose Bool [(LTSState, Int)] deriving Show
```

[7] Beachte, daß Satz 2.4.2 nur exemplarisch für $\forall$*belard* formuliert wurde, aber dual auch für $\exists$*loise* gilt. Es ist jedoch wahrscheinlicher, daß der Benutzer ein interaktives Model Checking Spiel startet, falls die eingegebene Formel nicht erfüllt ist, wodurch TRUTH zu $\forall$*belard* wird.

[8] Wegen der Länge des Sourcecodes verweisen wir auch hier auf [Tru98].

[9] Die Umwandlung des Gamegraphen in einen Spielbaum ist implementierungstechnisch vonnöten, da innerhalb der in TRUTH definierten **LTS-Monade**, die die effiziente Behandlung von Transitionssystemen gewährleistet, keine Ein- und Ausgabe möglich ist. Dazu dient die **IO-Monade**.

```
gameplay:: (Int,Int,Int) -> ProcEnv -> Process -> MuFormula -> IO ()
gameplay tablesizes penv proc f =
  let ad = alterDepth f in
  if ad > 1
    then error ("...")
    else
      let (me, st, gametree) =
            runLTS tablesizes defLabeling
            (
              getFreshStateLTS `thenLTS` \ st ->
              setLabelLTS st proc `seqLTS`
              addToStateHTblLTS proc st `seqLTS`
              cgg penv st phi fpMap fixedPointMap formMap
              `thenLTS` \ color ->
              let me = case color of
                           Red -> I
                           Green -> II
              in
              game_tree penv st phi me color `thenLTS` \ tree ->
              returnLTS (me, st, tree)
            )
          you = if me == I then II else I
      in
      do
       putStrLn ("Computing winning strategy ...")
       putStrLn ("O.K., I'm player "++ show me ++
                 " and you are player "++ show you ++".")
       do_play st phi gametree
```

Abbildung 4.3: Die Funktion `gameplay`.

Um ein komfortables, interaktives Model Checking Spiel durchführen zu können, unterscheidet TRUTH zwischen Konfigurationen,

- die ein normales Ende (durch Erreichen von `tt` bzw. `ff`),
- die ein Ende durch Erreichen einer $\langle\ \rangle$- oder $[\ ]$-Konfiguration, für die kein Nachfolgezustand im Transitionssystem gefunden werden kann,
- die lediglich das Abwickeln einer Fixpunktformel verlangen,
- die einen Zug von TRUTH ermöglichen,
- in denen der Benutzer wählt, jedoch nur eine einzige Möglichkeit hat,[10] und denen,
- in denen der Benutzer wirklich eine Wahl hat.

[10] Aus praktischen Gründen lassen wir diesen Zug automatisch geschehen.

Die Aufgabe der Funktion `do_play` besteht dann lediglich darin, den Spielbaum zu durchlaufen, anhand des `GameMoves`-Typs jeder Konfiguration eine entsprechende Meldung auszugeben und eventuell den Benutzer dazu aufzufordern, seine Wahl zu machen.

4.2.2 Das Modul `MuGameGraph.lhs`

`MuGameGraph.lhs` exportiert nur zwei Funktionen, die beide der Darstellung eines (eventuell gefärbten) Gamegraphen dienen.

```
module MuGameGraph
(
   dottyGameGraph,
   coloredGameGraph,
)
```

Diese Darstellung ist nicht Teil von TRUTH selbst, sondern wird dem Graphvisualisierungsprogramm **Dotty** ([Kou94]) überlassen.
Die Aufgabe der Funktion `dottyGameGraph` besteht darin, einen Gamegraphen vollständig aufzubauen, dabei die Information über Knoten, Knotenbeschriftungen und Kanten zu sammeln, in Dottycode umzuwandeln, in eine Datei zu schreiben und schließlich Dotty mit dieser Datei als Argument aufzurufen. Die Syntax des Dottycodes ist in der Grammatik aus Abb. 4.4 dargestellt.[11]

4.2.3 Anbindung an Truth

Die Einbindung der beiden Module `MuGamePlay.lhs` und `MuGameGraph.lhs` zu TRUTH erfolgt in dem Hauptmodul `Main.lhs`.

```
module Main(main)

where

...
import MuGamePlay
import MuGameGraph
...
```

Der Aufruf der Model Checking Funktion `gamecheck` aus Modul `MuGamePlay.lhs` z. B. ist in Abb. 4.5 dargestellt. Der Aufruf von `gameplay` erfolgt analog.

[11] Dabei handelt es sich lediglich um eine Grammatik zur Beschreibung der Graphen, die auch wirklich von TRUTH erzeugt werden.

Dotty ::= `strict digraph` *Name*
`{`
NodesOrEdges
`}`

NodesOrEdges ::= *Node NodesOrEdges*
| *Edge NodesOrEdges*
| ϵ

Node ::= *Name* `[ label = "` *Name* `" color =` *Color* `] ;`

Color ::= `green`
| `red`

Edge ::= *Name* `->` *Name* `;`

Name ::= *ASCII**

Abbildung 4.4: Die Syntax des Dottycodes.

4.3 Benutzerschnittstelle

TRUTH meldet sich beim Start mit dem in Abb. 4.6 dargestellten Bild und wartet auf Eingaben des Benutzers.

Bsp. 4.3.1 Um die Funktionsweise der vorgestellten Befehle zu verdeutlichen, gehen wir nochmals auf die in Bsp. 2.5.1 eingeführte Formel $\varphi = \mu X.\langle b\rangle \mathtt{tt} \vee [-]X$ und das Transitionssystem $\mathcal{T}$ aus Abb. 2.12 ein. $\mathcal{T}$ [12] läßt sich leicht in **CCS** formulieren und somit in TRUTH eingeben:

```
truth> def A = a.B + a.C
truth> def B = b.C
truth> def C = a.B
truth>
```

Die angegebene Formel gibt man folgendermaßen ein:

```
truth> prop Phi = min X.<b>tt || [-]X
```

[12] TRUTH verwendet Buchstaben für **CCS**-Prozessbezeichner und natürliche Zahlen für die Zustände der daraus resultierenden Transitionssysteme. In diesem Beispiel gilt somit: `A` $= 1$, `B` $= 2$ und `C` $= 3$.

```
gameMain rest menv@(MkMainEnv inh outh sty us penv muenv idenv actenv
                    msize pt showt davinci )
 =   readIds 2 rest menv >>= \ maybe_params ->
     case maybe_params of
          Nothing -> driver_loop menv
          Just [id1,id2] -> forkComputation (run_checkprop id1 id2)
                                            (driver_loop menv)
 where
   run_checkprop id1 id2
     = case lookupMuEnv muenv id2 of
         Nothing -> hPutStrLn outh "unknown proposition"
         Just (_,f) ->
           case expandFormula f muenv of
             Left f' ->
               case gamecheck msize penv (makeProcFromId id1) f' of
                 True -> hPutStrLn outh
                   "TRUE, the process satisfies the formula."
                 False -> hPutStrLn outh
                   "FALSE, the process does not satisfy the formula."
             Right err -> hPutStrLn outh err
```

Abbildung 4.5: Die Anbindung der Funktion `gamecheck`.

Zweimaliges Betätigen der Tabulator-Taste zu Beginn des TRUTH-Prompts liefert eine Übersicht über alle Befehle, die TRUTH verarbeiten kann.[13]

```
truth>
agent             game              plot0             simulate
altdepth          gamegraph         plot1             size
bye               help              plot2             states
clear             init              print             statesexp
coloredgamegraph  list              process           tab
deadlocks         lsa               prop              tableau
debug             nodebug           property          tablesize
def               obs               quit              timing
derivatives       output            relabel           transitions
exit              play              save              vs
file              plot              set
truth>
```

Im folgenden werden die im Zusammenhang mit dieser Arbeit interessanten Befehle erläutert.

[13]Eine ausführlichere Beschreibung der einzelnen Befehle erhält man durch Eingabe von `help`.

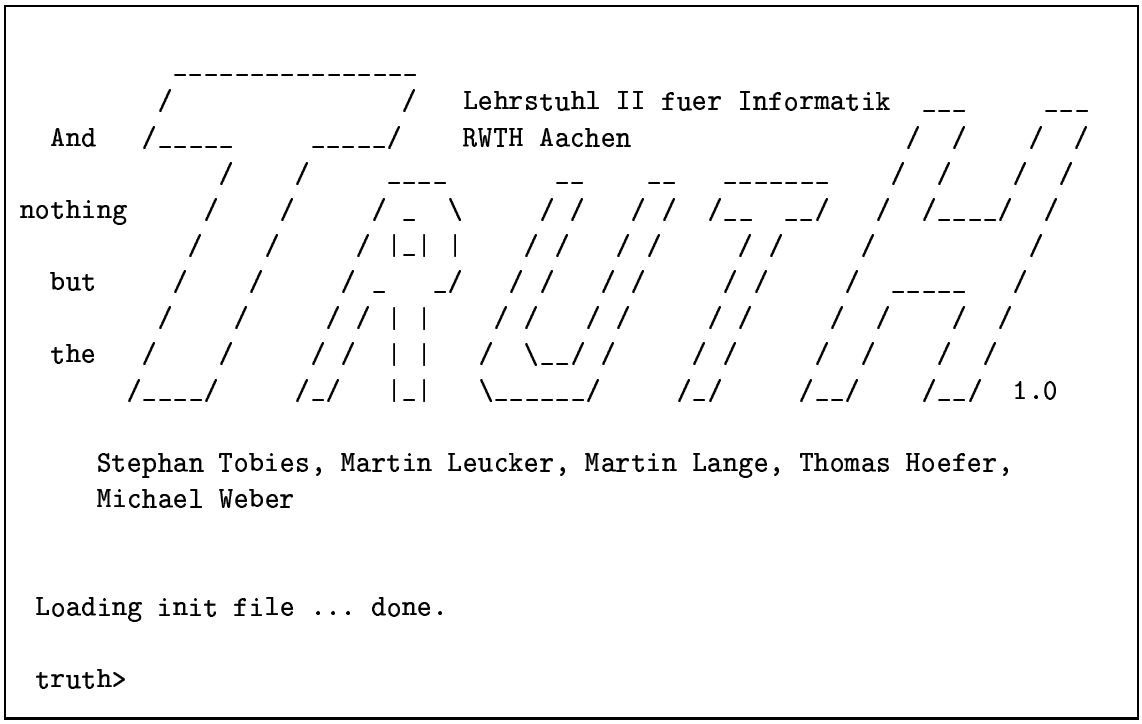

Abbildung 4.6: Der Start von TRUTH.

4.3.1 Die Model Checking Befehle `game` und `play`

`game` erwartet als Argument einen CCS-Prozessbezeichner P, aus dem ein Transitionssystem $\mathcal{T}$ mit Startzustand s gewonnen wird, und einen Bezeichner für eine eingegebene μ-Kalkül-Formel φ. Die Ausgabe der Funktion ist dann entweder

- `"TRUE, the process satisfies the formula."`, falls $(\mathcal{T}, s) \models \varphi$, oder
- `"FALSE, the process does not satisfy the formula."`, falls $(\mathcal{T}, s) \not\models \varphi$.

Bsp. 4.3.2 Wendet man die Funktion `game` auf die in Bsp. 4.3.1 vorgestellten Argumente an, so sagt der Model Checking Algorithmus, daß das Transitionssystem die Formel erfüllt:

```
truth> game A Phi
TRUE, the process satisfies the formula.
```

`play` erwartet dieselben Argumente wie `game`. Nachdem wie bei `game` der spielbasierte Model Checking Algorithmus aufgerufen wurde, wird ein interaktives Model Checking Spiel gestartet. Die zwei möglichen Partien des Model Checking Spiels für $\mathcal{T}$ und φ aus Bsp. 2.5.1 sind in Abb. 4.7 und 4.8 angegeben. Da die Formel erfüllt ist, übernimmt TRUTH den Part von $\exists$*loise*, während der Benutzer $\forall$*belards* Züge übernimmt. Der Unterschied

```
truth> play A Phi
Computing winning strategy ...
O.K., I'm Eloise and you are Abelard.
Current game configuration is (1, min X.<b>tt || [-]X) ...
Unfolding fixed point formula ...
Current game configuration is (1, <b>tt || [-]X) ...
I choose next node ...
Current game configuration is (1, [-]X) ...
Make your choice:
 Enter 0 to select (2,X)
 Enter 1 to select (3,X)
Enter a number between 0 and 1: 0
Current game configuration is (2, X) ...
Unfolding fixed point formula ...
Current game configuration is (2, min X.<b>tt || [-]X) ...
Unfolding fixed point formula ...
Current game configuration is (2, <b>tt || [-]X) ...
I choose next node ...
Current game configuration is (2, <b>tt) ...
I choose next node ...
Current game configuration is (3, tt) ...
We have reached the end and I have won. Yippie!
truth>
```

Abbildung 4.7: Eine mögliche, interaktive Partie.

zwischen diesen beiden Partien liegt in der Konfiguration $(1, [-]X)$, in der der Benutzer wählen darf, ob mit dem Zustand 2 oder 3 weitergespielt wird.
Damit entsprechen diese beiden interaktiven Partien den beiden Gewinnpartien aus Abschn. 2.5.

4.3.2 Die Gamegraphbefehle

`gamegraph` bewirkt die Ausgabe des gesamten Gamegraphen zu einem Transitionssystem und einer Formel. Die Eingabe von

```
truth> gamegraph A Phi
```

liefert den Gamegraphen, wie er in Abb. 2.13 bereits gezeigt wurde.

`coloredgamegraph` zeigt den Teil des Gamegraphen zu einem Transitionssystem und einer Formel, der vom Algorithmus **CGG** wirklich gefärbt wurde.[14] Abb. 4.11 zeigt den von

[14]Da **CGG** lokal ist, muß das nicht der gesamte Gamegraph sein.

```
truth> play A Phi
Computing winning strategy ...
O.K., I'm Abelard and you are Eloise.
Current game configuration is (1, min X.<b>tt || [-]X) ...
Unfolding fixed point formula ...
Current game configuration is (1, <b>tt || [-]X) ...
I choose next node ...
Current game configuration is (1, [-]X) ...
Make your choice:
 Enter 0 to select (2,X)
 Enter 1 to select (3,X)
Enter a number between 0 and 1: 1
Current game configuration is (3, X) ...
Unfolding fixed point formula ...
Current game configuration is (3, min X.<b>tt || [-]X) ...
Unfolding fixed point formula ...
Current game configuration is (3, <b>tt || [-]X) ...
I choose next node ...
Current game configuration is (3, [-]X) ...
You have no choice but ...
Current game configuration is (2, X) ...
Unfolding fixed point formula ...
Current game configuration is (2, min X.<b>tt || [-]X) ...
Unfolding fixed point formula ...
Current game configuration is (2, <b>tt || [-]X) ...
I choose next node ...
Current game configuration is (2, <b>tt) ...
I choose next node ...
Current game configuration is (3, tt) ...
We have reached the end and I have won. Yippie!
truth>
```

Abbildung 4.8: Eine zweite mögliche, interaktive Partie.

	# Zustände	# Transitionen
`Flugzeug`	17	24
`ABP`	23	28
`Big`	12167	44436
`Mutex`	84	160
`SYSTEM`	1715	8305
`Knuth`	253	508
`KBig`[17]	> 30000	

Abbildung 4.9: Die Größen der einzelnen Transitionssysteme.

Dotty dargestellten, gefärbten[15] Gamegraphen zu Bsp. 4.3.1. Der Knoten, der mit `N2F5` bezeichnet ist, ist nicht gefärbt und zeigt, daß der Algorithmus nicht unbedingt den gesamten Gamegraphen aufbauen muß, um für die Startkonfiguration eine korrekte Färbung zu erzeugen.

```
truth> coloredgamegraph A Phi
```

4.4 Laufzeitmessungen

Zur Messung der Laufzeit des implementierten Algorithmus dient die in TRUTH eingebauten Funktion `timing`, die dafür sorgt, daß TRUTH mit jeder normalen Ausgabe noch drei Zeiten ausgibt, die in Millisekunden angeben, wie lange TRUTH zum Abarbeiten eines Befehls gebraucht hat. Diese Zeiten haben die Namen *user*, *system* und *ellapsed*. Ihre Unterscheidung ist notwendig, weil auf Multitasking-Systemen mit anderen, laufenden Programmen, ansonsten die gemessenen Zeiten wenig Aussagekraft hätten.

Die Laufzeiten des Model Checking Algorithmus **CGG** werden mit denen des ebenfalls in TRUTH implementierten *Tableau*-Algorithmus anhand der mit TRUTH mitgelieferten Beispiele[16] verglichen.
Diese Beispielspezifikationen sind:

- `flugzeug`: Dies ist eine einfache Modellierung eines Flugzeugs durch atomare Komponenten wie Höhensensor, Höhensteuerung, usw. Das betrachtete Transitionssystem heißt `Flugzeug`.
- `abp`: Dies ist eine **CCS**-Implementierung eines alternierenden Bit-Protokolls. Sie stellt zwei Transitionssysteme, `ABP` und `Big`, zur Verfügung.

[15] Dabei stellt eine rechteckige Umrandung eines Knoten wieder die Farbe Rot dar, während alle anderen Knoten bis auf eine Ausnahme grün gefärbt sind.
[16] s. [Tob98]
[17] Aus Speicherplatzgründen ist die Größe von `KBig` mit TRUTH nicht genauer ermittelbar.

	CGG	*Tableau*
`Flugzeug`	50	10
`ABP`	110 60	60 40
`Big`	31170	18750
`Mutex`	190 380	110 200
`SYSTEM`	6340	4390
`Knuth`	340 90 1060	250 60 670
`KBig`	69950 200 800	40100 140 550

Abbildung 4.10: Die Laufzeiten im Vergleich.

- `petersen`: Dies ist eine **CCS**-Modellierung des wechselseitigen Ausschlußproblems mit der Lösung von Petersen. Das zugehörige Transitionssystem heißt `Mutex`.
- `atm`: Dies ist eine Spezifikation der Connect-Phase zweier ATM-Switches mit dem Transitionssystem `SYSTEM`.
- `knuth`: Dies ist eine ähnliche Modellierung wie `petersen`, jedoch enthält sie die Lösung von Knuth. Dazu gehören zwei Transitionssysteme: `Knuth` und `KBig`.

Die Größen der einzelnen Transitionssystem sind in Abb. 4.9 angegeben.

Da die Formeln, die zu den Beispielspezifikationen gehören, zumeist speziell auf das entsprechende Transitionssystem zugeschnitten sind und somit wertlos für andere Transitionssysteme sind, sparen wir uns die explizite Angabe der $\mathcal{L}_\mu^1$-Formeln, die in den Laufzeitmessungen verwendet wurden.

Abb. 4.10 zeigt die einzelnen Laufzeiten der Implementierung des Algorithmus **CGG**, verglichen mit dem *Tableau*-Algorithmus. Dabei ist nur die reine *user*-Zeit angegeben.

4.4.1 Fazit

Wie man sieht, ist der eingebaute *Tableau*-Model Checker schneller als die implementierte Version des Algorithmus **CGG**. Die Laufzeiten $t_{Tableau}$ und $t_{\mathbf{CGG}}$ verhalten sich im

Durchschnitt etwa folgendermaßen:

$$t_{Tableau} \leq t_{\mathbf{CGG}} \leq 2 \cdot t_{Tableau}$$

Dies ist nicht schlecht, da bei der Implementierung des *Tableau*-Algorithmus Wert auf Geschwindigkeit gelegt wurde, während es sich bei der Implementierung des Algorithmus **CGG** um eine nicht optimierte Version handelt. Folgende zwei Möglichkeiten zur Beschleunigung des Model Checkers lassen geringere Laufzeiten erwarten:

- Knoten im Gamegraphen, die nur einen einzigen Nachfolger haben, sind uninteressant und tragen nichts wesentliches zur Struktur und damit zur Färbung des Gamegraphen bei. Viele dieser Knoten lassen sich direkt eliminieren, da sie anhand der Formelkomponente identifiziert werden können. In einer Implementierung können sie somit weggelassen werden. Dies wurde in TRUTH jedoch nicht gemacht, weil dann die interaktiven Spiele und die Theorie der Model Checking Spiele nicht mehr exakt übereinstimmen würden.

- Ein Umordnen der Formel kann ebenfalls das Model Checking beschleunigen. So ist es für die Gültigkeit vollkommen egal, ob eine Formel z. B. als $\varphi \vee \psi$ oder als $\psi \vee \varphi$ aufgeschrieben wird. In der Implementierung macht es jedoch einen Unterschied, zu welcher Unterformel im Gamegraphen zuerst übergegangen wird, falls eine Formel in einen Zykel, die andere jedoch in einen endlichen Pfad führt. Endliche Pfade werden niemals von **COLOR-BACKWARDS** besucht und sind somit schneller zu färben.

  Die Bedingung, daß eine Formel keine Variablen enthält, ist lediglich hinreichend, leider aber nicht notwendig dafür, daß sie in einen endlichen Pfad führt.

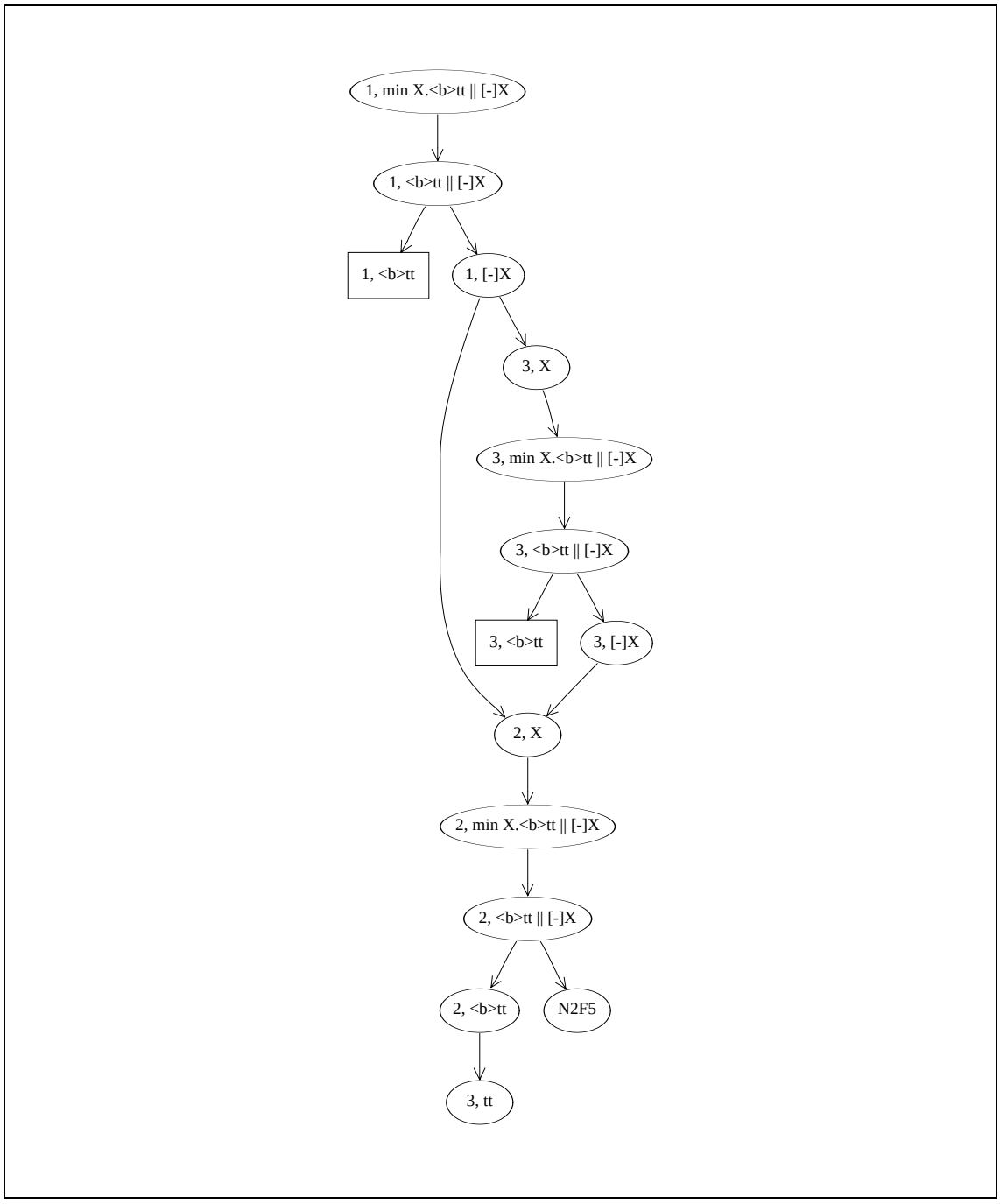

Abbildung 4.11: Der gefärbte Gamegraph zu Bsp. 2.5.1.

Zusammenfassung und Ausblick

Zusammenfassung

Diese Arbeit beschreibt die Entwicklung und die Implementierung eines Model Checking Algorithmus für den alternierungsfreien μ-Kalkül. Sie untersucht ebenso drei bestehende Algorithmen für dieses Problem ([CS92, And94, BC96]. Dabei fallen zwei Dinge auf:

1. Alle bestehenden Algorithmen benutzen die gleiche Struktur, um Model Checking zu betreiben, benennen sie aber jeweils anders. Die Begriffe *Boolescher Graph*, *And-Or Kripke-Struktur* und *Gleichungsblöcke* sind somit äquivalent und können zu **Model Checking Graph** vereinigt werden. Dabei handelt es sich um einen baumähnlichen Graphen, der auch Zykel und sich vereinigende Pfade besitzen kann.

2. Ein globaler Algorithmus, der diese Struktur vollständig zur Verfügung hat, kann das Model Checking Problem für den alternierungsfreien μ-Kalkül in linearer Zeit lösen. Ein lokaler Algorithmus, der diese Struktur je nach Bedarf während der Berechnug aufbaut, rechnet mit dem zeitlichen Aufwand $O(n \cdot log\, n)$, was mit den zur Zeit bekannten Datenstrukturen zur Verwaltung von Knoten eines Graphen nicht unterboten werden kann, falls die zugrundeliegende Struktur der Model Checking Graph ist.

Das Model Checking läßt sich auf das Markieren bzw. Färben dieses Graphen zurückführen. Andere Ansätze sind die Reduktion auf Boolesche Gleichungssysteme oder alternierende Automaten.

Da die betrachtete Logik Quantoren in Fixpunktform bereitstellt, muß beim Markieren bzw. Färben des Graphen unter Umständen ein Initialwert vergeben werden, der sich später als falsch herauststellen kann. Die Kombination von Zykeln, sich vereinigenden Pfaden, falschen Markierungen und der Forderung nach einem lokalen Algorithmus läßt das Problem nicht-trivial werden.

Die bestehenden Algorithmen sind zwar effizient, aber zum Einsatz auf dem Gebiet der *Verifikation* und *Spezifikation* nur bedingt geeignet. Ihnen fehlt eine theoretische Grundlage, mit der sich erklären läßt, *warum* eine Formel nicht erfüllt ist, anstatt lediglich zu zeigen, *daß* sie nicht erfüllt ist.
Eine erforderliche Grundlage liefert die Theorie der Model Checking Spiele, wie sie von Stirling vorgestellt wurde ([Sti97]). Diese Spiele liefern wiederum eine andere Interpretation des Model Checking Graphen, *Gamegraph* genannt. Das Färben dieses Gamegraphen entspricht dem Beantworten der Frage, welcher der beiden Spieler eine Gewinnstrategie für ein Model Checking Spiel zu einem Transitionssystem und einer alternierungsfreien

μ-Kalkül-Formel hat. Aufgrund der Natur der Spiele wird somit gleichzeitig die Frage beantwortet, ob die gegebene Formel in einem gegebenen Zustand des Transitionssystems gilt.
Es wird ein Algorithmus beschrieben, der genau dies leistet, seine Terminierung und Korrektheit bewiesen und seine Komplexität angegeben. Da der Algorithmus lokal ist, ist seine Zeitkomplexität ebenfalls $O(n \cdot log\, n)$.

Da Model Checking alleine im Bereich der *Spezifikation* und *Verifikation* verteilter Systeme unzureichend ist, wird ein Verfahren beschrieben, das mithilfe des spielbasierten Model Checking Algorithmus zeigt, *warum* eine Formel nicht erfüllt ist, und somit einen interaktiven Prozess zwischen Benutzer und Verifikationstool bei der Modellierung (verteilter) Systeme ermöglicht.

Der spielbasierte Model Checking Algorithmus und das Verfahren zum interaktiven Spielen sind in dem Verifikationstool TRUTH ([Tob98, Tru98]) implementiert worden. Damit hat der Benutzer von TRUTH nun die Möglichkeit, seine abstrakten Modelle eines verteilten Systems im Hinblick auf eine Eigenschaft, die sich im alternierungsfreien μ-Kalkül ausdrücken läßt, genauer zu untersuchen und eventuell zu erkennen, welche Veränderungen an dem Modell notwendig sind, um es bezüglich der angesprochenen Eigenschaft fehlerfrei zu machen.
Dies wird durch interaktive Model Checking Spiele verwirklicht, in denen TRUTH die Gewinnstrategie eines Spielers ausnutzt, um dem Benutzer einen „Fehler“ in der Spezifikation anzuzeigen.

Ausblick

Die untersuchten Algorithmen legen die Vermutung nahe, daß sich die bisher gefundenen Komplexitätsschranken des Model Checking für den alternierungsfreien μ-Kalkül nicht unterbieten lassen. Daher ist es nicht sinnvoll, nach weiteren Algorithmen dafür zu suchen, solange nicht eine neue Anforderung an solche Algorithmen besteht, wie z. B. die Frage nach der Generierung eines Gegenbeispiels.

Die weiteren Arbeiten zur Beschleunigung der Algorithmen liegen also auf dem praktischen Gebiet der Optimierung von Implementierungen.

Die in TRUTH eingebauten, interaktiven Model Checking Spiele sind rein textbasiert und werden daher leicht unübersichtlich. Sie sind somit nicht optimal geeignet, um interaktive Model Checking Spiele für Transitionssysteme, die wesentlich größer sind als die in dieser Arbeit benutzten Beispiele, zu spielen. Da jede einigermaßen interessante Modellierung eines realen Systems jedoch größer sein wird, besteht der Bedarf nach einer besseren Darstellung dieser Spiele. Ein grafischer Ansatz, der dem Benutzer besser zeigt, welches der aktuelle Zustand des untersuchten Transitionssystems im Spiel ist, ist somit ein sinnvoller Schritt in der Weiterentwicklung von TRUTH.

Index

Abbildungsverzeichnis

Literaturverzeichnis

[And94] H. R. Andersen. Model checking and Boolean graphs. *Theoretical Computer Science*, 126(1):3–30, April 1994.

[BC96] Girish Bhat and Rance Cleaveland. Efficient model checking via the equational μ-calculus. In *Proceedings, 11$^{\text{th}}$ Annual IEEE Symposium on Logic in Computer Science*, pages 304–312, New Brunswick, New Jersey, 27–30 July 1996. IEEE Computer Society Press.

[Bir98] Richard J. Bird. *Introduction to Functional Programming using Haskell.* Prentice-Hall Series in Computer Science. Prentice-Hall Europe, London, UK, second edition, 1998.

[Bra96] J. C. Bradfield. The modal mu-calculus alternation hierarchy is strict. In Ugo Montanari and Vladimiro Sassone, editors, *CONCUR '96: Concurrency Theory, 7th International Conference*, volume 1119 of *Lecture Notes in Computer Science*, pages 233–246, Pisa, Italy, 26–29 August 1996. Springer-Verlag.

[BVW94] O. Bernholtz, M. Y. Vardi, and P. Wolper. An automata-theoretic approach to branching-time model checking. *Lecture Notes in Computer Science*, 818:142 ff, 1994.

[CES85] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications. Technical report, Austin University of Texas, Austin, 1985.

[CKS81] Ashok K. Chandra, Dexter C. Kozen, and Larry J. Stockmeyer. Alternation. *Journal of the ACM*, 28(1):114–133, January 1981.

[Cle90] R. Cleaveland. Tableau-based model checking in the propositional mu-calculus. *Acta Informatica*, 27(8):725–748, 1990.

[CLR92] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to algorithms.* MIT Press and McGraw-Hill Book Company, 6th edition, 1992.

[CS92] R. Cleaveland and B. Steffen. A linear–time model–checking algorithm for the alternation–free modal mu–calculus. In Kim G. Larsen and Arne Skou, editors, *Proceedings of Computer Aided Verification (CAV '91)*, volume 575 of *LNCS*, pages 48–58, Berlin, Germany, July 1992. Springer.

[Dam94] M. Dam. CTL* and ECTL* as fragments of the modal μ-calculus. *Theoretical Computer Science*, 126(1):77–96, April 1994.

[Dav92] A. J. T. Davie. *An Introduction to Functional Programming Using Haskell.* Cambridge University Press, Cambridge, UK, 1992. (pbk).

[DR95] V. Diekert and G. Rozenberg. *The Book of Traces*. World Scientific, Singapore, 1995.

[DRCS97] G. Denker, J. Ramos, C. Caleiro, and A. Sernadas. A linear temporal logic approach to objects with transactions. *Lecture Notes in Computer Science*, 1349:170 ff, 1997.

[Eme90] E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 14, pages 996–1072. Elsevier Science Publishers B.V.: Amsterdam, The Netherlands, New York, N.Y., 1990.

[Eme96] E. A. Emerson. *Automated Temporal Reasoning about Reactive Systems*, volume 1043 of *Lecture Notes in Computer Science*, page 41 ff. Springer-Verlag Inc., New York, NY, USA, 1996.

[Fec96] C. Fecht. *Abstrakte Interpretation logischer Programme: Theorie, Implementierung, Generierung*. PhD thesis, Saarbrücken, Universität des Saarlandes, 1996.

[HF92] P. Hudak and J. H. Fasel. A gentle introduction to haskell. *SIGPLAN Notices*, 27(5), May 1992.

[HHP+92] C. V. Hall, K. Hammond, W. D. Partain, S. L. Peyton Jones, and P. L. Wadler. The glasgow haskell compiler: a retrospective. In *Functional Programming, Glasgow 1992*. Springer-Verlag, New York, NY, 1992. Springer-Verlag Workshops in Computing.

[Hoa78a] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978. See corrigendum [Hoa78b].

[Hoa78b] C. A. R. Hoare. Corrigendum: "Communicating Sequential Processes". *Communications of the ACM*, 21(11):958–958, November 1978. See [Hoa78a].

[HPW+91] Paul Hudak, Simon Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joeseph Fasel, María M. Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, Dick Kieburtz, Rishiyur Nikhil, Will Partain, and John Peterson. Report on the Functional Programming Language Haskell. Research Report 1.1, Department of Computing Science, University of Glasgow, Glasgow, UK, June 1991. Final Draft.

[HU80] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, N. Reading, MA, 1980.

[Hug90] J. Hughes. Why functional programming matters. In D. A. Turner, editor, *Research Topics in Functional Programming*, UT Year of Programming Series, chapter 2, pages 17–42. Addison-Wesley, 1990.

[JP93] SL. Peyton Jones and Wadler P. Imperative functional programming. In *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages, Charleston, South Carolina*, pages 71–84, January 1993.

[JW96] D. Janin and I. Walukiewicz. On the expressive completeness of the modal mu-calculus w.r.t. monadic second order logic. In *Proc. CONCUR'96*, volume 1119 of *Lecture Notes in Computer Science*, 1996.

[Kou94] Eleftherios Koutsofios. Editing graphs with *dotty*. Technical report, AT&T Bell Laboratories, Murray Hill, NJ, USA, July 1994.

[Koz83] Dexter Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, 27:333–354, December 1983.

[KWV98] O. Kupferman, P. Wolper, and M. Vardi. An automata-theoretic approach to branching-time model checking.
http://www.cs.rice.edu/ vardi/papers/cav94rj.ps.gz, 1998.

[Lau93] J. Launchbury. Lazy imperative programming. In *Proceedings of the ACM SIGPLAN Workshop on State in Programming Languages, Copenhagen, DK, SIPL '92*, pages 46–56, 1993.

[LJ94] J. Launchbury and S. Peyton Jones. Lazy functional state threads. In *Programming Languages Design and Implementation*, Orlando, 1994. ACM Press.

[LLNT99] M. Lange, M. Leucker, T. Noll, and S. Tobies. Truth – a verification platform for concurrent systems. In *Tool Support for System Specification, Development, and Verification*, Advances in Computing Science. Springer-Verlag Wien New York, 1999.

[LT98] Martin Leucker and Stephan Tobies. Truth—a platform for verification of distributed systems. Technical Report 98-05, RWTH Aachen, May 1998.

[Mad97] A. Mader. *Verification of Modal Properties Using Boolean Equation Systems.* PhD thesis, Munich, University of Technology, 1997.

[Maz88] A. Mazurkiewicz. Basic notions of trace theory. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Proceedings of the School/Workshop on Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, pages 364–397. LNCS 354. Springer, June 1988.

[MH84] S. Miyano and T. Hayashi. Alternating finite automata on omega-words. *Theoretical Computer Science*, 32(3):321–330, August 1984.

[Mil80] R. Milner. A calculus of communicating systems. *Lecture Notes in Computer Science*, 92, 1980.

[Mil89] R. Milner. *Communication and Concurrency.* International Series in Computer Science. Prentice Hall, 1989.

[MS87] D. E. Muller and P. E. Schupp. Alternating automata on infinite trees. *Theoretical Computer Science*, 54(2-3):267–276, October 1987.

[MSS92] D. E. Muller, A. Saoudi, and P. E. Schupp. Alternating automata, the weak monadic theory of trees and its complexity. *Theoretical Computer Science*, 97(2):233–244, April 1992.

[Pet62] C. A. Petri. Fundamentals of a theory of asynchronous information flow. *Proc. IFIP Congress, N-H*, 1962.

[Pet86] Carl Adam Petri. Concurrency Theory. In *Advanced Course on Petri Nets*, pages 1–22, Bad Honnef, September 8–19, 1986. Gesellschaft für Mathematik und Datenverarbeitung, St. Augustin, Germany.

[Rei82] W. Reisig. *Petri Nets.* Springer, Berlin, 1982.

[Sti92] Stirling, C. Modal and temporal logics. In *Handbook of Logic in Computer Science*, volume 2 (Background: Computational Structures), pages 477–563. Clarendon Press, Oxford, 1992.

[Sti97] C. Stirling. Games for bisimulation and model checking, June 1997. Notes for Mathfit instructional meeting on games and computation, Edinburgh, http://www.dcs.ed.ac.uk/home/cps/mathfit.ps.

[Tar55] A. Tarski. A lattice-theoretical fixpoint theorem and its application. *Pacific J.Math.*, 5:285–309, 1955.

[Tho96] Simon Thompson. *Haskell: The Craft of Functional Programming.* Addison-Wesley, Harlow, England, 1996.

[Tob98] Stephan Tobies. Design und Implementierung einer Plattform zur Verifikation verteilter Systeme. Master's thesis, Aachen, University of Technology, 1998. (German).

[Tru98] TRUTH, 1998. http://www-i2.informatik.rwth-aachen.de/Forschung/MCS/Truth/.

[WN93] Glynn Winskel and Mogens Nielsen. Models for concurrency. Technical report, Basic Research in Computer Science (BRICS), Aarhus, 1993. IB-B941066.