

Technische Universität München

Proseminar

„Software Desaster

und wie man sie verhindern kann“

MODEL CHECKING

Hampel Manuel

Kursleitung: Professor Tobias Nipkow

Betreuer: Gerwin Klein

8.Januar 2003

Gliederung:

1. Einführung

2. Model Checking

2.1 Systeme modellieren

2.1.1 Hilfsstrukturen

2.1.2 nebenläufige Systeme -> First-Order-Formel

2.1.2.1 Stromkreise

2.1.2.2 Programme

2.1.3 First-Order-Formel -> Kripke Struktur

2.2 Die temporale Logik CTL

2.3 Model Checking Algorithmus

3. Ausblick

1. Einführung

In vielen Anwendungen sind Soft- und Hardwarefehler heutzutage nicht mehr tolerierbar. So zum Beispiel bei Systemen zur Verkehrskontrolle, Flugzeugen oder medizinischen Instrumenten, um nur ein paar der unzähligen Beispiele zu nennen. Je mehr solche Systeme Teil unseres täglich Lebens werden, umso mehr sollte auch ihr korrektes und fehlerfreies Verhalten sichergestellt sein.

Für komplexe Systeme gibt es 4 Methoden, mit denen man sie auf ihre Korrektheit überprüfen kann:

- Simulation
- Testen
- Verifikation mit deduktiven Methoden
- Model Checking

Für die **Simulation** und das **Testen** müssen Tests ausgeführt werden, bevor man das System nutzen kann. Die Simulation arbeitet mit einer Abstraktion oder einem Modell, das Testen hingegen wird auf dem eigentlichen Produkt, bzw. System ausgeführt. In beiden Methoden werden Signale an gewissen Stellen des Systems eingegeben und die Resultate ausgewertet. Diese beiden Methoden können für große Systeme allerdings sehr kosteneffizient sein. Außerdem ist es kaum möglich durch Testen und Simulieren alle möglichen Fehler des Systems zu finden.

Deduktive Methoden verwenden eine Menge von Axiomen und Beweisregeln um die Korrektheit eines Systems zu überprüfen. Dies ist jedoch recht zeitaufwendig und kann nur von Experten ausgeführt werden. Daher kommt sie in der Praxis nur in hochsensitiven Systemen, zum Beispiel in Sicherheitsprotokollen vor. Der Vorteil ist, dass man auch Systeme mit unbegrenzten Zustandsräumen behandeln kann.

Die Methode des **Model Checking** ist eine Technik, um nebenläufige Systeme mit endlichem Zustandsraum zu betrachten. Die Einschränkung auf den endlichen Zustandsraum hat den Vorteil, dass die Verifikation automatisch ausgeführt werden kann. Die Prozedur durchsucht normalerweise den Zustandsraum des Systems um zu entscheiden, ob eine Spezifikation, gegeben als Formel der temporalen Logik, wahr oder falsch ist, also die Zustände des Systems die Formel erfüllen. Obwohl die Einschränkung auf einen endlichen Zustandsraum im ersten Augenblick als Nachteil erscheinen mag, so ist Model Checking doch auf verschiedene, sehr wichtige Klassen von Systemen anwendbar, z. B. Hardware-Controller oder viele Kommunikationsprotokolle. In einigen Fällen können auch Systeme mit unendlichen Zustandsräumen behandelt werden, indem man sie durch Abstraktion oder Induktion verkleinert. Eine neue Forschungsrichtung versucht Model Checking und deductive Verification zu kombinieren.

2. Model Checking

Das Anwenden von Model Checking auf einen Entwurf beruht auf mehreren Schritten:

- Modellieren
- Spezifizieren
- Verifizieren

Beim **Modellieren** (Abschnitt 2.1) wandelt man den Entwurf in eine Formel um, die von einem Model-Checking-Tool akzeptiert wird. Man versucht also vom Entwurf aus

direkt oder über eine Formel einen Zustandsübergangsgraphen, zu erzeugen. In den meisten Fällen ist das nur eine Übersetzungsaufgabe, allerdings kann das Modellieren aufgrund von Zeit- und Speicherbegrenzung aber auch eine Abstraktion, das Löschen unwichtiger Details, erfordern.

Als nächstes muss man die Eigenschaften angeben, die das System erfüllen soll. Dies geschieht in der **Spezifikation** (Abschnitt 2.2), welche üblicherweise in einer Formel der temporalen Logik gegeben ist. Ein wichtiger Punkt, den man nicht übersehen darf, ist die Vollständigkeit. Denn Model Checking liefert nur Mittel um zu überprüfen, ob ein Modell eine bestimmte Spezifikation erfüllt, kann aber natürlich nicht bestimmen, ob die Spezifikation alle Eigenschaften abdeckt, die das System haben soll.

Der letzte Schritt ist die **Verifikation** (Abschnitt 2.3). Hierfür wird das in der Modellierungsphase erzeugte Modell und die in der Spezifikation erstellte Formel benötigt. Idealerweise funktioniert die Verifikation automatisch, allerdings kommt es in der Praxis oft vor, dass man von außen eingreifen muss. Eine solche manuelle Tätigkeit wäre z. B. die Analyse der Verifikationsergebnisse.

2.1 Systeme modellieren

2.1.1 Hilfsstrukturen

Um Systeme zu modellieren benötigen wir noch einige Begriffe, die wir an dieser Stelle einführen wollen.

Kripke Struktur

Sei AP eine Menge von atomaren Aussagen, dann ist eine **Kripke Struktur M** über AP ein Vier-Tupel $M = (S, S_0, R, L)$, wobei gilt:

- S ist eine endliche Mengen von Zuständen
- $S_0 \subseteq S$ ist die Menge der Anfangszustände
- $R \subseteq S \times S$ ist eine Übergangsrelation, die total sein muss, d.h. jeder Zustand muss einen Nachfolgezustand haben
- $L : S \rightarrow 2^{AP}$ ist eine Funktion, die jeden Zustand mit der Menge der atomaren Aussagen markiert, die in diesem Zustand wahr sind

First Order Representation

Weiterhin brauchen wir ein Basiswissen in der First-Order-Logik, d.h. man sollte u. a. mit den Operatoren „and“ \wedge , „or“ \vee , „not“ \neg , „impliziert“ \Rightarrow vertraut sein. Wir benutzen interpretierte First-Order-Formeln um nebenläufige Systeme zu beschreiben, wobei interpretiert hier meint, dass wir der Syntax eine Bedeutung (Semantik) zuordnen. Sei $V = \{v_1, v_2, \dots, v_n\}$ die Menge der Systemvariablen. Die Werte der Variablen v_i seien aus D , der Domäne oder dem Universum der Interpretation. Eine Belegung für V ist also eine Funktion, die jeder Variable aus V einen Wert aus D zuordnet. Der Zustand eines nebenläufigen Systems kann durch die Belegung aller Elemente in V beschrieben werden. D.h. ein Zustand ist eine Belegung $s: V \rightarrow D$ für die Menge der Variablen in V . Haben wir eine Belegung, so können wir eine Formel angeben, die genau für diese Belegung wahr ist. Sei beispielsweise $V = \{v_1, v_2\}$ und die Belegung $\langle v_1 \leftarrow 4, v_2 \leftarrow 6 \rangle$ gegeben, so erhalten wir die Formel $(v_1 = 4) \wedge (v_2 = 6)$. Eine Formel kann aber auch für mehrere Belegungen wahr sein. Steht eine Formel

also für alle Belegungen, die die Formel erfüllen, so können wir eine Menge von Zuständen durch First-Order-Formeln, beschreiben.

Da wir nicht nur Mengen von Zuständen, sondern auch Mengen von Übergängen repräsentieren wollen, benutzen wir diesmal eine Formel, die eine Menge von geordneten Zustandspaaren beschreibt. Hierfür brauchen wir eine zweite Menge von Variablen V' . Wir betrachten die Variablen aus V als Variablen des gegenwärtigen Zustands und die Variablen aus V' als Variablen des nächsten Zustands. Jede Variable $v \in V$ hat somit eine entsprechende Variable $v' \in V'$. Einen Übergang oder ein geordnetes Zustandspaar erhalten wir durch eine Belegung der Variablen aus V und V' . Mengen dieser Belegungen können durch Formeln repräsentiert werden. Eine Übergangsrelation entspricht also einer Menge von geordneten Zustandspaaren. Ist R eine Übergangsrelation, so schreiben wir $R(V, V')$ um eine Formel zu beschreiben, die diese Relation beschreibt. Um Spezifikationen (Formeln) zu erhalten, die Eigenschaften unseres Systems beschreiben, müssen wir eine Menge von atomaren Aussagen definieren. Beispielsweise ist eine Aussage $v = d$ wahr, wenn gilt $s(v) = d$, die Variable v im Zustand s also den Wert d hat.

2.1.2 Aus einem nebenläufigen System eine First-Order-Formel erzeugen

Da wir jetzt die nötigen Hilfsstrukturen eingeführt haben, können wir mit der Modellierung beginnen. Ein nebenläufiges System besteht aus einer Menge von Komponenten die gemeinsam ablaufen. Normalerweise haben die Komponenten mehrere Möglichkeiten um miteinander zu kommunizieren. Wir unterscheiden 3 dieser Möglichkeiten, nämlich gemeinsame Variable, Warteschlangen, Handshaking-Protokolle, werden aber nur gemeinsame Variable behandeln. Ferner unterscheiden wir 2 Arten der Ausführung: synchron und asynchron. Bei der synchronen Ausführung machen alle Komponenten zum selben Zeitpunkt einen Schritt, während bei der asynchronen Ausführung immer nur eine Komponente einen Schritt pro Zeitpunkt macht. Im folgenden wollen wir zeigen, wie man einige wichtige Typen von nebenläufigen Systemen mit First-Order-Formeln ausdrücken kann. Aus diesen Formeln können wir dann Kripke Strukturen erhalten (Abschnitt 2.1.3).

2.1.2.1 Schaltkreise

Zuerst wollen wir Schaltkreise durch Formeln beschreiben. Der Einfachheit halber gehen wir davon aus, dass die zustandshaltenden Elemente, z.B. Flip-Flops, nur den Wert 0 oder 1 haben. Bei synchronen Schaltkreisen besteht die Menge V aus den Ausgaben aller zustandshaltenden Elementen im Schaltkreis zusammen mit den Anfangseingaben. Bei asynchronen Schaltkreisen werden alle Drähte im Schaltkreise als zustandshaltende Elemente betrachtet.

synchrone Schaltkreise

Die Tätigkeit eines synchronen Schaltkreises besteht aus einer Reihe von Schritten. In jedem Schritt ändern sich die Eingaben des Schaltkreises und der Schaltkreis kann sich stabilisieren. Dann tritt ein Taktgeber auf und die zustandshaltenden Elemente verändern sich.

Die Methode zum Erhalten einer Übergangsrelation aus einem synchronen Schaltkreis soll zuerst einmal an einem Beispiel (Modulo-8-Zähler) erläutert werden. Sei $V = \{v_0,$

v_1, v_2 } die Menge der Zustandsvariablen für unseren Schaltkreis und $V' = \{ v'_0, v'_1, v'_2 \}$ eine Kopie der Zustandsvariablen. Die Übergänge seien wie folgt definiert:

$$v'_0 = \neg v_0$$

$$v'_1 = v_0 \oplus v_1$$

$$v'_2 = (v_0 \wedge v_1) \oplus v_2,$$

wobei \oplus das exklusive oder bezeichne. Mit diesen Gleichungen kann man dann folgende Relationen definieren:

$$R_0(V, V') \equiv (v'_0 \Leftrightarrow \neg v_0)$$

$$R_1(V, V') \equiv (v'_1 \Leftrightarrow v_0 \oplus v_1)$$

$$R_2(V, V') \equiv (v'_2 \Leftrightarrow (v_0 \wedge v_1) \oplus v_2).$$

Diese Relationen beschreiben die Bedingungen, die jedes v'_i in einem legalen Übergang erfüllen muss. Da wir uns in einem synchronen Stromkreis befinden, treten alle Veränderungen genau zur selben Zeit auf, d.h. um die Formel für die Übergangsrelation zu erhalten müssen alle Bedingungen mit dem Konjunktionsoperator verbunden werden:

$$R(V, V') \equiv R_0(V, V') \wedge R_1(V, V') \wedge R_2(V, V').$$

Der allgemeine Fall hierzu geht völlig analog:

Jetzt enthalten V und V' n Zustandsvariablen, d.h. $V = \{ v_0, v_1, \dots, v_n \}$ und $V' = \{ v'_0, v'_1, \dots, v'_n \}$. Weiterhin haben wir wieder für jede Zustandsvariable eine boolesche Funktion f_i , so dass gilt

$$v'_i = f_i(v).$$

Mit diesen Gleichungen können wir dann wieder unsere Relationen R_i definieren:

$$R_i(V, V') \equiv (v_i \Leftrightarrow f_i(v)).$$

Abschließend müssen nun nur noch alle R_i mit dem Konjunktionsoperator verknüpft werden:

$$R(V, V') \equiv R_0(V, V') \wedge R_1(V, V') \wedge \dots \wedge R_{n-1}(V, V')$$

Man kann eine Übergangsrelation für einen synchronen Schaltkreis also als Konjunktion der Übergangsrelationen der individuellen Prozesse ausdrücken.

asynchrone Schaltkreise

Die Übergangsrelation wird durch eine Disjunktion beschrieben, da immer nur ein Prozess zu einem Zeitpunkt ausgeführt wird. Um uns den Weg zur Übergangsrelation zu erleichtern, nehmen wir an, dass alle Komponenten des Stromkreises genau eine Ausgabe und keine internen Zustandsvariablen haben. So ist es möglich jede Komponente durch eine Funktion $f_i(v)$ zu beschreiben. Gibt man einer Komponente einen Wert für die gegenwärtige Zustandsvariable v als Eingabe, so ändert die Komponente ihren Wert zu dem durch $f_i(v)$ spezifizierten Wert. Als Resultat erhalten wir folgende Disjunktion:

$$R(V, V') \equiv R_0(V, V') \vee R_1(V, V') \vee \dots \vee R_{n-1}(V, V')$$

wobei

$$R_i(V, V') \equiv (v_i \Leftrightarrow f_i(v)) \wedge (\forall j \neq i)(v'_j \Leftrightarrow v_j)$$

Das heißt, dass sich nur v_i ändert, während alle anderen $v_j, j \neq i$, gleich bleiben. Es sei angemerkt, dass es möglich ist, dass sich einige Komponenten ständig ändern,

während andere immer gleich bleiben. Mit der Problematik der Fairness werden wir uns an dieser Stelle aber nicht weiter auseinandersetzen.

Zum Abschluß noch ein Beispiel um den Unterschied zwischen synchronen und asynchronen Modellen zu demonstrieren. Sei $V = \{v_0, v_1\}$, wobei $(v'_0 = v_0 \oplus v_1) \wedge (v'_1 = v_0 \oplus v_1)$. Ferner sei s ein Zustand in dem gilt, dass $(v_0 = 1) \wedge (v_1 = 1)$. Bei der synchronen Abarbeitung der Anweisungen erhalten wir $(v_0 = 0) \wedge (v_1 = 0)$. Bei der asynchronen Abarbeitung haben wir jedoch 2 Lösungen, je nachdem ob zuerst v_0 oder v_1 ausgeführt wird:

- $(v_0 = 0) \wedge (v_1 = 1)$, falls v_0 zuerst
- $(v_0 = 1) \wedge (v_1 = 0)$, falls v_1 zuerst.

2.1.2.2 Programme

Wir wollen das Modellieren von Programmen am Beispiel eines interagierenden, asynchronen, nebenläufigen Programms demonstrieren, d.h. das Programm hat mehrere Prozesse (sequentielle Folgen von Anweisungen), wobei zu einem Zeitpunkt immer nur eine Anweisung eines Prozesses ausgeführt werden kann. Die Interaktion zwischen den Prozessen findet in unserem Fall durch gemeinsame Variable statt.

Um die Übersetzungsprozedur C , die den Text eines Programms als Eingabe nimmt und daraus eine First-Order-Formel R (die Menge der Übergänge eines Programms) erzeugt, am Beispiel vorführen zu können, müssen wir noch eine Markierungstransformation einführen. Dazu nehmen wir an, dass jede Anweisung einen Anfangs- und einen Endpunkt hat. Die Markierungstransformation erzeugt aus einem unmarkiertem Programm P ein markiertes Programm P^L , indem es an jedem Anfangs- und Endpunkt einer Anweisung eine Markierung setzt. Alle Markierungen müssen unterschiedlich sein. Ein nebenläufiges Programm P , mit $P = \mathbf{cobegin} P_1 \parallel P_2 \parallel \dots \parallel P_n \mathbf{coend}$, wobei P_1, \dots, P_n sequentielle Prozesse sind, hat nach der Markierungstransformation die Form $P^L = \mathbf{cobegin} l_1 : P_1^L \ l'_1 \parallel l_2 : P_2^L \ l'_2 \parallel \dots \parallel l_n : P_n^L \ l'_n \mathbf{coend}$

Ein nebenläufiges Programm kann nicht als Anweisung in einem sequentiellen Programm auftreten. Weiterhin gehen wir davon aus, dass wir nur bereits markierte Programme P^L als Eingabe bekommen. Auf die genaue Markierungsprozeduren der einzelnen Anweisungstypen soll hier nicht weiter eingegangen werden. Der Anfangs- und Endpunkt von P sei mit m , bzw. m' markiert. Sei pc , der Befehlszähler (program counter), eine spezielle Variable aus der Menge der Programmmarkierungen, wobei pc auch noch den Wert \perp (undefinierter Wert) annehmen kann, um auszudrücken, dass das Programm nicht aktiv ist. V bezeichne die Menge der Programmvariablen v , V' die Menge der Variablen des Nachfolgezustands v' für jedes $v \in V$. Ferner seien V_i die Menge der Variablen, die von Prozess P_i verändert werden können. Die Variablenmengen V_i und V_j dürfen natürlich gemeinsame Variablen enthalten. Der Befehlszähler eines Prozesses P_i sei pc_i , PC sei die Menge aller Befehlszähler und pc' die Variablen des Nachfolgezustands für pc . Da jeder Übergang normalerweise nur

eine kleine Anzahl der Programmvariablen verändert, führen wir folgende Abkürzung ein

$$\text{same}(y) = \bigwedge_{y \in Y} (y' = y),$$

um auszudrücken, dass alle $y \in Y$ unverändert bleiben. Folgende Formel beschreibt die Menge der Anfangszustände eines nebenläufigen Programms P:

$$S_0(V, PC) \equiv \text{pre}(V) \wedge \text{pc} = m \wedge \bigwedge_{i=1}^n (pc_i = \perp).$$

Nun zur eigentlichen Übersetzungsprozedur C, die wir an hand eines Beispiels beschreiben wollen.

Sei $P = m : \text{cobegin } P_0 \parallel P_1 \text{ coend } m'$ ein Programm für gegenseitigem Ausschluß, wobei

```

P0 :: l0 : while True do
    NC0 : wait(turn = 0);
    CR0 : turn := 1;
endwhile;
l'0

```

und

```

P1 :: l1 : while True do
    NC1 : wait(turn = 1);
    CR1 : turn := 0;
endwhile;
l'1.

```

Der Befehlszähler pc nimmt nur drei Werte an: m, m' und \perp . Außerdem hat jeder Prozess P_i einen Befehlszähler, der entweder l_i , l'_i , NC_i , CR_i oder \perp enthält. Die einzige Programmvariable ist turn. Hat der Befehlszähler pc_i des Prozesses P_i den Wert CR_i , so befindet sich dieser Prozess in einer kritischem Bereich, d.h. der andere Prozess darf zu diesem Zeitpunkt den kritischen Bereich nicht betreten. Hat pc_i einen von CR_i verschiedenen Wert, so befindet sich P_i in einem nichtkritischen Bereich und wartet, bis turn den Wert i bekommt und den kritischen Bereich zu betreten. Die Anfangszustände von P sind durch folgende Formel beschrieben:

$$S_0(V, PC) \equiv \text{pc} = m \wedge pc_0 = \perp \wedge pc_1 = \perp.$$

Der Wert von turn kann zu Beginn des Prozesses also 0 oder 1 sein. Wenn wir die Übersetzungsprozedur C anwenden erhalten wir die gewünschte Formel für die Übergangsrelation von P, $R(V, PC, V', PC')$, was folgender Formeln entspricht:

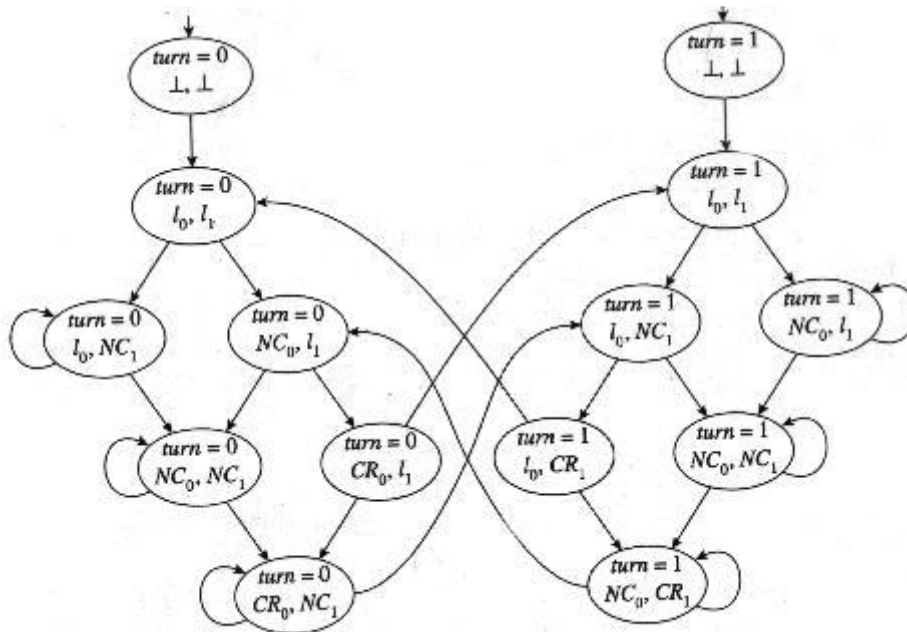
$$\begin{aligned}
& (\text{pc} = m \wedge pc'_0 = l_0 \wedge pc'_1 = l_1 \wedge \text{pc}' = \perp) \vee \\
& (pc_0 = l'_0 \wedge pc_1 = l'_1 \wedge \text{pc}' = m' \wedge pc'_0 = \perp \wedge pc'_1 = \perp) \vee \\
& C(l_0, P_0, l'_0) \wedge \text{same}(\text{pc}, pc_1) \vee \\
& C(l_1, P_1, l'_1) \wedge \text{same}(\text{pc}, pc_0).
\end{aligned}$$

Das erste Disjunkt drückt nur aus, dass das Programm P^L startet, im nächsten Schritt also die beiden Prozesse P_0 und P_1 gestartet werden. Das zweite Disjunkt wird wahr, falls die beiden Prozesse P_0 und P_1 beendet sind und somit P^L zu Ende ist. Im dritten Disjunkt wird P_0 und im vierten P_1 ausgeführt.

Die beiden Prozess $P_i \text{ C}(l_i, P_i, l'_i), i \in \{1, 2\}$, sind durch folgende Formeln definiert:

$$\begin{aligned}
 & (pc_i = l_i \wedge pc'_i = NC_i \wedge \text{True} \wedge \text{same}(\text{turn})) \vee \\
 & (pc_i = NC_i \wedge pc'_i = CR_i \wedge \text{turn} = i \wedge \text{same}(\text{turn})) \vee \\
 & (pc_i = CR_i \wedge pc'_i = l_i \wedge \text{turn}' = (i + 1) \bmod 2) \vee \\
 & (pc_i = NC_i \wedge pc'_i = NC_i \wedge \text{turn} \neq i \wedge \text{same}(\text{turn})) \vee \\
 & (pc_i = l_i \wedge pc'_i = l'_i \wedge \text{False} \wedge \text{same}(\text{turn})).
 \end{aligned}$$

Hier steht das erste Disjunkt für den Fall, dass der Prozess P_i beginnt und zum nicht-kritischen Bereich NC_i übergegangen wird. Im zweiten Disjunkt wird zum kritischen Bereich CR_i übergegangen, wobei hier die Bedingung $\text{turn} = i$ zu erfüllen ist. Im dritten Teil der Formel wird der kritische Bereich CR_i wieder verlassen, zuvor aber noch der Wert der Variablen turn entsprechend der Formel $\text{turn}' = (i + 1) \bmod 2$ verändert. Das vierte Disjunkt wird wahr, falls der Prozess P_i im nicht-kritischen Bereich NC_i verweilen muss, da die Bedingung $\text{turn} = i$, zum Betreten des kritischen Bereichs CR_i , nicht erfüllt ist. Der letzte Teil der Formel meint, dass der Prozess P_i gar nicht erst ausgeführt wird, falls die Bedingung in der while-Schleife False ist. In unserem Beispiel kann dieser Fall nicht eintreten, d.h. unser Programm terminiert nicht. Die folgende Abbildung zeigt die Kripke Struktur unseres Beispiels.



Wie man allgemein die Kripke Struktur aus den Formeln S_0 und R erhält, wird im folgenden beschrieben.

2.1.3 Aus einer First-Order-Formel eine Kripke Struktur erzeugen

Wir wollen nun aus den First-Order-Formeln S_0 und R eine Kripke Struktur erhalten, die das nebenläufige System repräsentiert. Dazu definieren wir:

- die Menge der Zustände S ist die Menge aller Belegungen für V
- die Menge der Anfangszustände S_0 ist die Menge aller Belegungen für V , die die Formel S_0 erfüllen
- seien s und s' 2 Zustände. Wenn man jedem $v \in V$ den Wert $s(v)$ und jedem $v' \in V'$ den Wert $s(v')$ zuordnet und dann R , die Formel für die Übergangsrelation, wahr ist, dann gilt $R(s, s')$.
- die Markierungsfunktion $L : S \rightarrow 2^{AP}$ ist so definiert, dass $L(s)$ die Teilmenge aller atomaren Aussagen ist, die in S wahr sind. Da die Übergangsrelation total sein muss, müssen wir sie erweitern, wenn ein Zustand s keinen Nachfolger hat. In diesem Fall modifizieren wir R so, dass $R(s, s)$ gilt.

Nun ein kleines Beispiel, um die neuen Begriffe besser zu verstehen. Wir betrachten ein System mit den Variablen $x, y \in D = \{0, 1\}$. Eine Belegung für x und y ist also ein Paar $(d1, d2) \in D \times D$, wobei $d1$ der Wert für x und $d2$ der Wert für y ist. Nehmen wir an, das System besteht aus dem Übergang

$$x := (x + y) \text{ mod } 2,$$

Das System kann dann durch zwei First-Order-Formeln beschrieben werden. Die Menge der Anfangszustände ist beschreiben durch

$$S_0(x, y) \equiv (x = 1) \wedge (y = 1)$$

und die Menge der Übergänge durch die Formel

$$R(x, y, x', y') \equiv (x' = x + y \text{ mod } 2) \wedge (y' = y).$$

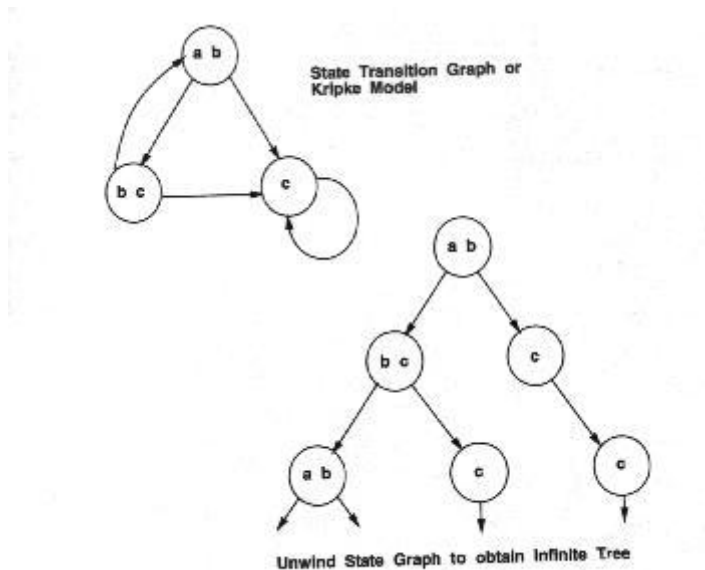
Somit erhält man folgende Kripke Struktur $M = (S, S_0, R, L)$:

- $S = D \times D$
- $S_0 = \{(1, 1)\}$
- $R = \{((1, 1)(0, 1)), ((0, 1)(1, 1)), ((1, 0)(1, 0)), ((0, 0)(0, 0))\}$
- $L((1, 1)) = \{x = 1, y = 1\}$, $L((0, 1)) = \{x = 0, y = 1\}$, $L((1, 0)) = \{x = 1, y = 0\}$,
 $L((0, 0)) = \{x = 0, y = 0\}$

2.2 Temporale Logiken

Die temporale Logik soll Eigenschaften eines Zustandübergangssystems oder (in unserem Fall) einer Kripke Strukturen spezifizieren. Sie benutzt dazu atomare Aussagen und boolesche Operatoren, wie z.B. Konjunktion oder Disjunktion, um kompliziertere Ausdrücke aufzubauen, um die Eigenschaften eines Zustandes zu beschreiben. Die temporale Logik ist ein Formalismus um Sequenzen von Übergängen zwischen Zuständen in einem reaktiven System zu beschreiben. Die temporalen Logiken unterscheiden sich in ihren Operatoren und der Bedeutung ihrer Operatoren. Es gibt folglich mehrere temporale Logiken. Wir betrachten hier nur die CTL (Computation Tree Logic), die wie die LTL (Linear Temporal Logik) eine Unterlogik der CTL* darstellt.

Hier müssen wir den Begriff des Berechnungsbaumes einführen. Dieser wird gebildet, indem man einen Zustand in einer Kripke-Struktur als Anfangszustand bestimmt und dann die Struktur in unendlich viele Bäume aufzweigt. Der Berechnungsbaum zeigt dann alle möglichen Ausführungen bei einem Start vom Anfangszustand aus. Die folgende Abbildung soll dies illustrieren:



Die CTL beschreibt die Pfade, die von einem bestimmten Zustand aus erreichbar sind. Dies geschieht durch Formeln, die aus Pfadquantifizierer und temporalen Operatoren bestehen.

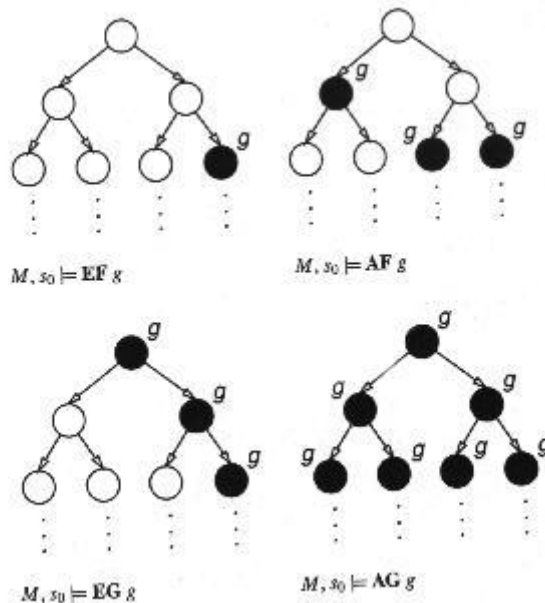
Es gibt folgende 2 Pfadquantifizierer

- **A** (für alle Berechnungspfade)
- **E** (für einige Berechnungspfade)

und unter anderem die folgenden 4 temporalen Operatoren:

- **X** („next“, wenn eine Eigenschaft im nächsten Zustand eines Pfades gilt)
- **F** („eventually“ oder „in the future“, wenn eine Eigenschaft in einigen Zuständen eines Pfades gilt)
- **G** („always“ oder „globally“, wenn eine Eigenschaft in allen Zuständen eines Pfades gilt)
- **U** („until“, wenn es einen Zustand auf dem Pfad gibt, in dem die zweite Eigenschaft und in jedem vorherigen Zustand die erste Eigenschaft gilt)

Die 4 am häufig auftretendsten Fälle seien im folgenden dargestellt:



In der CTL hat eine Formel die Form <Pfadquantifizierer><temporaler Operator>.

Syntax

Weiterhin gibt es 2 Arten von Formeln. Die Zustandsformeln und die Pfadformeln. Erstere gelten in einem bestimmten Zustand, die anderen entlang eines bestimmten Pfades. Sei AP die Menge der atomaren Aussagen.

Dann ist die Syntax der Zustandsformeln der CTL wie folgt festgelegt:

- ist $p \in AP$, so ist p eine Zustandsformel
- sind f und g Zustandsformeln, so sind auch $\neg f$, $f \wedge g$ und $f \vee g$ Zustandsformeln
- ist f eine Pfadformel, so sind $E f$ und $A f$ Zustandsformeln

Die Syntax der Pfadformel der CTL:

- sind f und g Zustandsformeln, dann sind $X f$, $F f$, $G f$, $f U g$ Pfadformeln

CTL ist die Menge aller Zustandsformeln, die durch die obigen Regeln erzeugt werden.

Für die CTL gibt es also die folgenden 8 Basisoperatoren

- **AX** und **EX**
- **SF** und **EF**
- **AG** und **EG**
- **AU** und **EU**,

die sich alle durch Terme mit den drei Operatoren **EX**, **EG** und **EU** ausdrücken lassen.

Semantik

Die Semantik von CTL definieren wir im Bezug auf die Kripke Struktur. Ein Pfad in M ist eine unendliche Sequenz von Zuständen, $p_i = s_0, s_1, \dots$ so dass für jedes $i \geq 0$ $(s_i, s_{i+1}) \in R$ ist. Man kann sich einen Pfad auch als unendlichen Zweig im Berechnungsbaum einer Kripke Struktur vorstellen. Wir verwenden π^i um das Suffix von π zu bezeichnen, das in s_i startet. Ist f eine Zustandsformel s , so bedeutet $M, s \models f$, dass f im Zustand s der Kripke Struktur M gilt. Sei f nun eine Pfadformel, so bedeutet $M, \pi \models f$, dass f entlang des Pfades π in der Kripke Struktur gilt. Die Relation \models ist induktiv wie folgt definiert:

1. $M, s \models p \Leftrightarrow p \in L(s)$
2. $M, s \models \neg f_1 \Leftrightarrow M, s \not\models f_1$
3. $M, s \models f_1 \vee f_2 \Leftrightarrow M, s \models f_1 \text{ oder } M, s \models f_2$
4. $M, s \models \mathbf{E} g_1 \Leftrightarrow$ es existiert ein Pfad π von s aus, entlang dem $M, s \models g_1$ gilt
5. $M, \pi \models f_1 \Leftrightarrow$ s ist der erste Zustand von π und es gilt $M, s \models f_1$
6. $M, \pi \models \neg g_1 \Leftrightarrow M, s \not\models g_1$
7. $M, \pi \models g_1 \vee g_2 \Leftrightarrow M, s \models g_1 \text{ oder } M, s \models g_2$
8. $M, \pi \models \mathbf{X} g_1 \Leftrightarrow M, s \models g_1$
9. $M, \pi \models g_1 \mathbf{U} g_2 \Leftrightarrow$ es ex. ein $k \geq 0$, so dass $M, \pi^k \models g_2$ und für alle $0 \leq j < k$, $M, \pi^j \models g_1$ gilt

Mit $\vee, \neg, \mathbf{X}, \mathbf{U}$ und \mathbf{E} können alle anderen CTL-Formeln ausgedrückt werden können.

2.3 CTL-Model-Checking-Algorithmus

Ziel beim Model Checking ist es zu einer gegebenen Kripke Struktur $M = (S, R, L)$ (die ein nebenläufiges System mit begrenztem Zustandsraum repräsentiert) und einer temporalen logischen Formel f (die gewisse gewünschte Spezifikationen ausdrückt) die Menge aller Zustände in S zu finden, die f erfüllen:

$$\{s \in S \mid M, s \models f\}.$$

Der Algorithmus wird jeden Zustand s mit der Menge der Unterformeln von f markieren, die wahr in s sind. Die Menge der Unterformeln sei durch $\text{label}(s)$ dargestellt, was gleichbedeutend mit $L(s)$ ist. Der Algorithmus durchschreitet mehrere Stufen: in der i -ten Stufe werden Unterformeln mit $i-1$ -geschachtelten CTL Operatoren bearbeitet. Wenn eine Unterformel bearbeitet wird, wird sie zur Markierung jedes Zustands hinzugefügt ist, für den sie wahr ist. Hat der Algorithmus terminiert, so erhalten wir $M, s \models f$ gdw $f \in \text{label}(s)$. Da ja alle CTL Formeln durch Terme mit $\neg, \vee, \mathbf{EX}, \mathbf{EU}, \mathbf{EG}$ ausgedrückt werden können, genügt es sechs Fälle zu behandeln. Entweder ist g atomar, oder hat eine der folgenden Formen: $\neg f_1, f_1 \vee f_2, \mathbf{EX} f_1, \mathbf{E}[f_1 \mathbf{U} f_2]$, oder $\mathbf{EG} f_1$.

Fall 1: g ist atomar

Wir markieren alle Zustände, in denen g gilt, mit g .

Fall 2: $g = \neg f_1$

Für Formeln der Form $\neg f_1$ markieren wir die Zustände, die nicht mit f_1 markiert sind, mit f_1 .

Fall 3: $g = f_1 \vee f_2$

Mit $f_1 \vee f_2$ markieren wir jeden Zustand, der entweder mit f_1 oder f_2 markiert ist.

Fall 4: $g = \mathbf{EX} f_1$

Für $\mathbf{EX} f_1$ markieren wir jeden Zustand, der einen Nachfolger hat, der mit f_1 markiert ist.

Fall 5: $g = \mathbf{E}[f_1 \mathbf{U} f_2]$

Wir müssen alle Zustände mit $\mathbf{E}[f_1 \mathbf{U} f_2]$ markieren, die die Bedingung $\mathbf{E}[f_1 \mathbf{U} f_2]$ erfüllen, d.h. alle Zustände von denen aus ich einen mit f_2 markierten Zustand entlang eines Pfades erreichen kann, in dem alle Zustände mit f_1 markiert sind. Der Pfad darf auch leer sein.

Um die Formel $\mathbf{E}[f_1 \mathbf{U} f_2]$ zu behandeln, suchen wir zuerst alle Zustände, die mit f_2 markiert sind. Dann benutzen wir die Umkehrung der Übergangsrelation und gehen den Pfad rückwärts entlang. Dies machen wir solange bis wir alle Zustände gefunden haben, die durch einen Pfad, auf dem jeder Zustand mit f_1 markiert ist, (von f_2 aus) erreicht werden können. Wir gehen hier wie davon aus, dass f_1 und f_2 schon erfolgreich bearbeitet wurden.

Dies führt auf folgende Prozedur CheckEU, die $\mathbf{E}[f_1 \mathbf{U} f_2]$, für jeden Zustand s , der $\mathbf{E}[f_1 \mathbf{U} f_2]$ erfüllt, zu $\text{label}(s)$ hinzufügt.

```

procedure CheckEU ( $f_1, f_2$ )
   $T := \{s \mid f_2 \in \text{label}(s)\};$ 
  for all  $s \in T$  do  $\text{label}(s) := \text{label}(s) \cup \{\mathbf{E}[f_1 \mathbf{U} f_2]\};$ 
  while  $T \neq \emptyset$  do
    choose  $s \in T;$ 
     $T := T \setminus \{s\};$ 
    for all  $t$  such that  $R(t, s)$  do
      if  $\mathbf{E}[f_1 \mathbf{U} f_2] \notin \text{label}(t)$  and  $f_1 \in \text{label}(t)$  then
         $\text{label}(t) := \text{label}(t) \cup \{\mathbf{E}[f_1 \mathbf{U} f_2]\};$ 
         $T := T \cup \{t\};$ 
      endif;
    end for all;
  end while;
end procedure

```

Fall 6: $g = \mathbf{EG} f_1$

Der Fall $g = \mathbf{EG} f_1$ ist leicht komplizierter. Hierzu müssen wir den Graphen in nichttriviale, stark zusammenhängende Komponenten zerlegen. Strongly Connected Components (SCC) sind maximale Teilgraphen, so dass jeder Knoten in C von jedem anderen Knoten aus C durch einen gerichteten, vollständig in C enthaltenen Pfad erreichbar ist. Nichttrivial bedeutet, dass C mehr als einen Knoten hat oder mindestens einen Knoten mit einer Schleife enthält. Die eingeschränkte Kripke Struktur M' erhalten wir, indem wir in der Kripke Struktur M alle Zustände aus S entfernen, in denen f_1 nicht gilt und L und R entsprechend einschränken. Es gilt also $M' = (S', R', L')$, wobei $S' = \{s \in S \mid M, s \not\models f_1\}$, $R' = R|_{S' \times S'}$, $L' = L|_{S'}$. Es gilt $M, s \models \mathbf{EG} f_1$, genau dann wenn folgende 2 Bedingungen erfüllt sind:

- $s \in S'$
- Es existiert ein Pfad in M' , der von einem Knoten s zu einem Knoten t führt, wobei t in einem nichttrivialen SCC des Graphen (S', R') liegt

Der Algorithmus für den Fall $g = \mathbf{EG} f_1$ folgt direkt aus den obigen 2 Bedingungen. Wir konstruieren wieder die eingeschränkte Kripke Struktur M' und partitionieren den Graphen in SCC. Nachdem wir alle Knoten der SCC mit $\mathbf{EG} f_1$ markiert haben, suchen wir noch solche Zustände, die zu nichttrivialen Komponenten gehören. Dazu

gehen wir mit der Umkehrung von R' rückwärts und markieren alle Zustände, von denen aus man einen SCC entlang eines Pfades erreichen kann, auf dem jeder Zustand mit f_1 markiert ist. Die folgende Prozedur CheckEG fügt $\mathbf{EG} f_1$ zu jedem Zustand s hinzu, der $\mathbf{EG} f_1$ erfüllt.

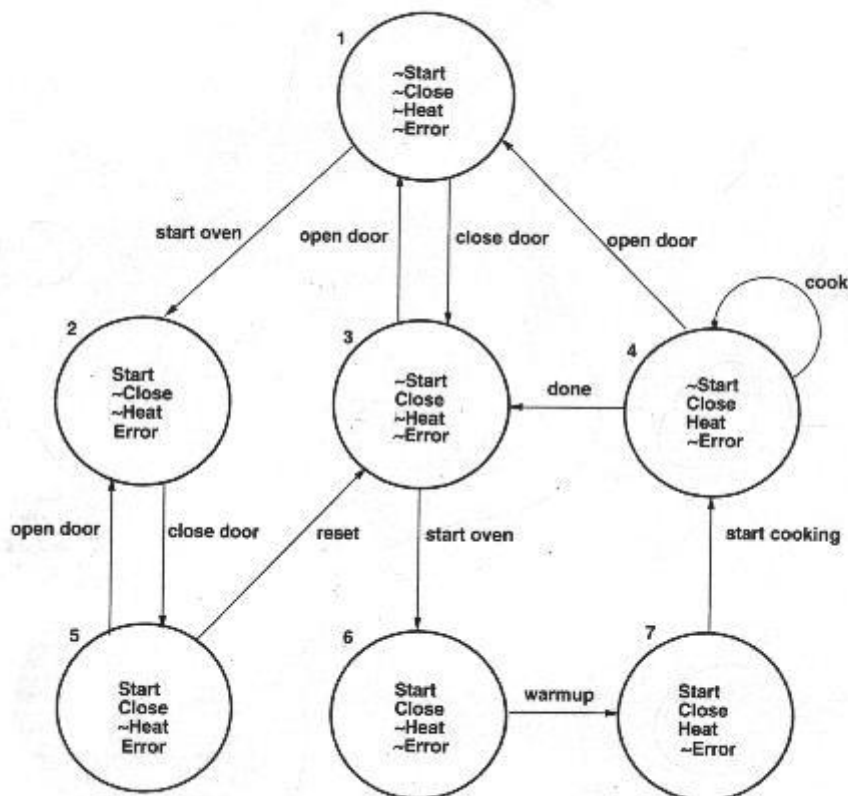
```

procedure CheckEG ( $f_1$ )
   $S' := \{s \mid f_1 \in \text{label}(s)\};$ 
   $\text{SCC} := \{C \mid C \text{ is a nontrivial SCC of } S'\};$ 
   $T := \bigcup_{C \in \text{SCC}} \{s \mid s \in C\};$ 
  for all  $s \in T$  do  $\text{label}(s) := \text{label}(s) \cup \{\mathbf{EG} f_1\};$ 
  while  $T \neq \emptyset$  do
    choose  $s \in T;$ 
     $T := T \setminus \{s\};$ 
    for all  $t$  such that  $t \in S'$  and  $R(t, s)$  do
      if  $\mathbf{EG} f_1 \notin \text{label}(t)$  then
         $\text{label}(t) := \text{label}(t) \cup \{\mathbf{EG} f_1\};$ 
         $T := T \cup \{t\};$ 
      end if;
    end for all;
  end while;
end procedure

```

Um jede CTL Formel f behandeln zu können, wenden wir den Zustandsmarkierungs-Algorithmus sukzessive auf die Teilformeln von f an, indem wir mit der kürzesten, am tiefsten verschachtelten starten und uns dann nach außen arbeiten. So garantieren wir, dass wenn wir eine Teilformel bearbeiten, alle Teilformeln dieser Teilformel schon bearbeitet wurden.

Abschließend wollen wir den Modell-Checking-Algorithmus am Beispiel eines Ofens illustrieren. Wir nehmen die folgende, dazugehörige Kripke-Struktur als gegeben an:



Jeder Zustand ist mit den atomaren Aussagen, die wahr in diesem Zustand sind, und mit der Negation der Aussagen, die falsch in diesem Zustand sind, markiert. Die Markierungen an den Kanten beschreiben die Aktionen, die die Übergänge verursachen und sind nicht Teil der Kripke-Struktur.

Wir betrachten die CTL-Formel $\mathbf{AG}(\text{Start} \rightarrow \mathbf{AF} \text{Heat})$, die zu der Formel $\neg \mathbf{EF}(\text{Start} \wedge \mathbf{EG} \neg \text{Heat})$ äquivalent ist (wir benutzen hier $\mathbf{EF} f$ als Abkürzung für $\mathbf{E}[\text{true} \mathbf{U} f]$). Wenn ich auf Start drücke, soll also der Ofen beginnen zu heizen.

Was wir also jetzt überprüfen wollen ist, ob das System, gegeben durch die Kripke-Struktur, die durch die CTL-Formel gestellten Spezifikationen erfüllt. Wir starten damit, die Menge der Zustände zu berechnen, die die atomaren Formeln erfüllen und gehen dann weiter zu komplizierteren Teilformeln von $\mathbf{AG}(\text{Start} \rightarrow \mathbf{AF} \text{Heat})$, bzw. $\neg \mathbf{EF}(\text{Start} \wedge \mathbf{EG} \neg \text{Heat})$. $S(g)$ bezeichne die Menge der Zustände, die mit der Teilformel g markiert sind.

1. Schritt:

$$S(\text{Start}) = \{2, 5, 6, 7\}$$

2. Schritt:

$$S(\neg \text{Heat}) = \{1, 2, 3, 5, 6\}$$

Um $S(\mathbf{EG} \neg \text{Heat})$ berechnen zu können, müssen wir zuerst entsprechend dem Algorithmus die Menge der nichttrivialen, stark zusammenhängenden Komponenten in $S' = S(\text{Heat})$ finden. Es ergibt sich als einzige stark zusammenhängende Komponente die Menge $\{1, 2, 3, 5\}$. Nun müssen wir noch mit f_1 markierte Zustände

suchen, von denen aus man entlang eines komplett mit f_1 markierten Pfades unsere stark zusammenhängende Komponente erreichen kann. Wir sehen, dass keine Zustände existieren, die die geforderte Bedingung erfüllen, kommen also zu $S(\mathbf{EG} \neg \text{Heat}) = \{1, 2, 3, 5\}$

3. Schritt:

$$S(\text{Start} \wedge \mathbf{EG} \neg \text{Heat}) = \{2, 5\}$$

4. Schritt:

$$S(\mathbf{EF}(\text{Start} \wedge \mathbf{EG} \neg \text{Heat})) = \{1, 2, 3, 4, 5, 6, 7\}$$

5. Schritt:

$$S(\neg \mathbf{EF}(\text{Start} \wedge \mathbf{EG} \neg \text{Heat})) = \emptyset$$

Da der Anfangszustand 1 in dieser Menge nicht enthalten ist, schließen wir daraus, dass das durch die Kripke-Struktur beschriebene System die geforderten Spezifikationen nicht erfüllt.

3. Ausblick

Abschließend ist zu sagen, dass Model Checking eine gerade im Hardware-Bereich oft verwendete und sichere Verifikationsmethode darstellt.

Ein beim Model Checking häufig auftretendes Problem ist allerdings, dass die Anzahl der Zustände im Zustandsübergangsgraphen zu groß wird. Eine Möglichkeit diesem Problem entgegenzuwirken sind sogenannte geordnete binäre Entscheidungsdiagramme (ordered binary decision diagrams). Diese OBDDs stellen eine Datenstruktur zur Repräsentation Boolescher Funktionen dar und finden vor allem in der digitalen Logik und im Schaltungsentwurf Anwendung. Der Vorteil der OBDDs liegt darin, dass sie Boolesche Funktionen sehr kompakt, d.h. unter Verwendung möglichst weniger Ressourcen, darstellen können. Diese Kompaktheit beruht darauf, dass keine isomorphen Teilgraphen betrachtet werden, d.h. identische Funktionsrepräsentationen werden zu einer zusammengefasst. Denn ist die Reihenfolge in der die Variablen aufgerufen werden fest, so ist die OBDD-Darstellung einer Booleschen Funktion (bis auf die Isomorphie) eindeutig. Da nun zu jeder repräsentierbaren Funktion nur noch ein Graph existiert, ist die Funktion eindeutig durch den Zeiger auf die Wurzel bestimmt. Jede repräsentierbare Funktion hat genau einen Zeiger. Will man also nun 2 Funktionen vergleichen, so müssen nur die beiden entsprechenden Zeiger verglichen werden. Auch die logischen Verknüpfungen wie Negation oder Disjunktion können recht einfach durch rekursive Prozeduren berechnet werden. Natürlich können mit OBDDs auch logische Spezifikationen zur Beschreibung reaktiver Systeme, wie wir sie oben verwendet haben, repräsentiert werden.