Otto-Friedrich-Universität Bamberg
Lehrstuhl Angewandte Informatik
Kognitive Systeme

Seminararbeit

Planen als Model Checking

Svetlana Balinova

Januar 2007

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einführung

Planen als Model Checking ist ein wichtiges Mittel, um unter Unsicherheit zu planen. Im Unterschied zum klassischen Planen, wo Determinismus, volle Beobachtbarkeit und erreichbare Ziele vorkommen, wird beim Planen unter Unsicherheit wenigstens eine dieser Eigenschaften verletzt:

- Wenn ein System nicht-deterministisch ist, hat die Anwendung einer Aktion mehrere mögliche Ergebnisse. Welches davon eintreten wird, kann nicht im voraus bestimmt werden.
- Die Steuerungskomponente eines Systems kann nicht immer über vollständige Information bezüglich der Zustände des Systems verfügen. Beispielsweise ein Proxy-Server kann nicht im voraus wissen ob eine Seite, die in seinem Cache gespeichert ist, aktualisiert wurde, bevor er eine HTTP-Anfrage (Hypertext Transfer Protocol) an dem Zielserver sendet. Falls der Zielserver mit nein antwortet (also die Seite wurde nicht aktualisiert),kann der Proxy-Server sehr schnell ein bestimmter Clientrequest bearbeiten. Andernfalls muss der Client warten, bis der Proxy-Server die aktualisierte Seite erhält.
- Ein System kann nie gewährleisten, dass ein Ziel mit Sicherheit erreicht wird - es sind viele Ausnahmen möglich wie z.B. Ausfall einer Systemkomponente. Erweiterte Ziele können solche Unsicherheiten berücksichtigen. Beispielsweise werden für die Übertragung von Medien-Dateien sehr oft unsichere Protokolle wie UDP (User Datagram Protocol) verwendet, weil sie deutlich schneller sind. Es gibt dabei keine Garantie, dass alle Pakete des zugehörigen Internet-Stroms erfolgreich ans Ziel ankommen. Ein System kann aber garantieren, dass auch bei Paketverluste die Sprachqualität einer Datei dank spezieller Wiederherstellungsverfahren nicht spürsam verletzt wird.

Anhand von Model Checking Techniken können diese Probleme erfolgreich gelöst werden. Das ist ein Verfahren zur Software- und Hardwareverifikation, welches üblicherweise dort eingesetzt wird, wo die zu prüfenden Systeme sicherheitskritische Eigenschaften erfüllen müssen. Mit Model Checking können alle Zustände eines Systems vollautomatisiert geprüft werden, im Unterschied zu den gewöhnlichen manuellen Testverfahren zur Verifikation.

2 Explicit Model Checking

Model Checking ist ein formales Verfahren (auf mathematische Techniken basierend), um nebenläufige Systeme automatisiert zu verifizieren. Damit werden Sicherheits- und Lebendigkeitseigenschaften zur Erhaltung eines Systems überprüft. Sicherheitseigenschaften beziehen sich auf Zustände, die in einem System nie vorkommen dürfen - beispielsweise dass zwei Prozesse gleichzeitig schreibend auf einer Datei zugreifen. Lebendigkeitseigenschaften stellen eine Art Garantie bereit, dass ein erwünschter Zustand irgendwann eintritt - z.B. irgendwann in der Zukunft wird ein Prozess den Schreibzugriff auf einer Datei bekommen. Wenn das System fair ist, liegt dieser Zeitpunkt nicht so weit.

Der Zustandsraum eines Systems wird beim expliziten Model Checking in Form einer Kripke Struktur dargestellt(Abb.2.1 A). Während der Verifizierung dieser Struktur wird ein Berechnungsbaum aufgebaut, der alle möglichen Abläufe im System berücksichtigt(Abb.2.1 B). Wird die zu überprüfende Eigenschaft bestätigt, antwortet der Model Checker mit ja. Falls der Modell Checker eine mögliche Verletzung der Eigenschaft feststellt, antwortet er mit nein und einen Gegenbeispiel.

Ein wichtiges Problem des Explicit Modell Checking ist die Tatsache, dass die Zustände der Kripke Struktur häufig eine enorm große Anzahl aufweisen. Eine mögliche Alternative bietet das symbolische Model Checking an - dabei wird der Zustandsraum anders aufgebaut, mit Hilfe von BDDs (Binary Decision Diagram). Es werden nicht mehr die einzelnen Zustände betrachtet, sondern Gruppen von Zuständen. BDDs und Symbolisches Model Checking werden in Kapitel 5 und 6 vorgestellt.

Eine Kripke Struktur K ist ein 4-Tupel (S,S_0,R,L) wobei

- S eine endliche Menge von Zuständen ist und
- $S_0 \subseteq$ S eine initiale Zustandsmenge.
- R $\subseteq$ S x S ist die Übergangsrelation, wobei für jeden Zustand s $\in$ S ein Nachfolgezustand s'$\in$ S angegeben werden kann.
- L: S $\rightarrow$ 2^P ist eine Beschriftungsfunktion, welche jedem Zustand die elementaren Aussagen von P zuweist. P ist eine Menge von elementaren Aussagen.

Abbildung 2.1 zeigt eine einfache Kripke Struktur, die ein Kollisionswarnsystem für Flugzeuge modelliert. Das System überwacht den umgebenden Raum des Flugzeugs und liefert Informationen darüber, ob sich ein anderes Flugzeug gefährlich nahe befindet. Wenn einen im voraus festgelegten Abstand zum eigenen Flugzeug unterschritten wird, wird das eindringende Flugzeug als grünes Licht auf einem Display im Cockpit dargestellt. Falls das andere Flugzeug noch näher kommt, wird die Farbe des Lichts im Display rot und es ertönt ein

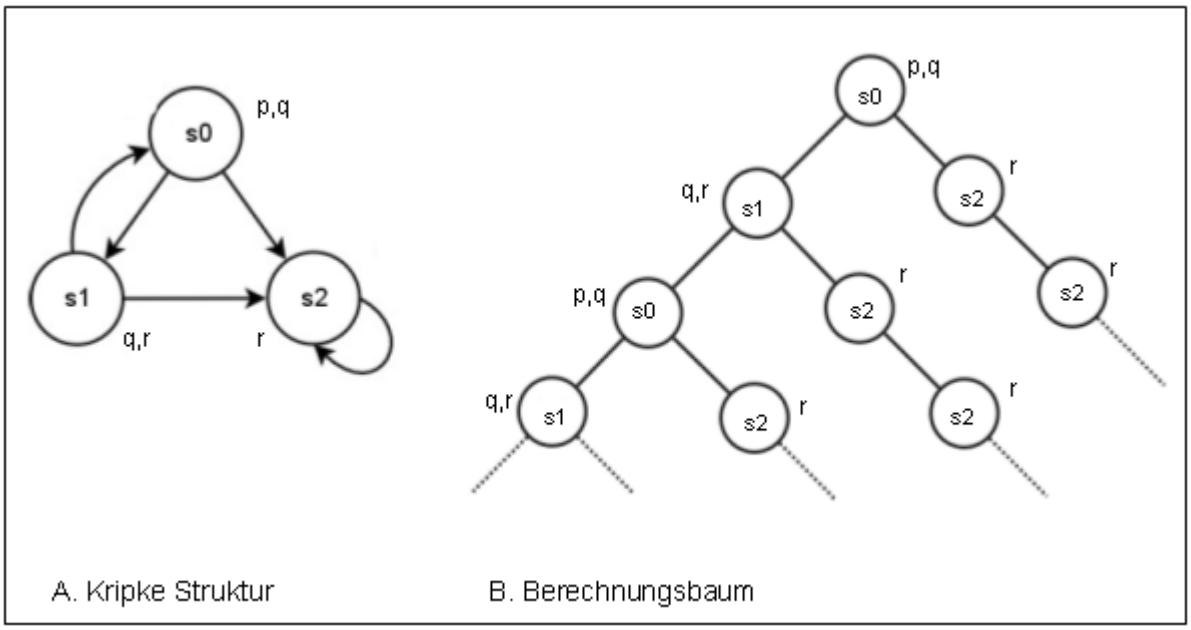

Abbildung 2.1: Bespiel für eine Kripke Struktur und der zugehörige Berechnungsbaum (Quelle[4])

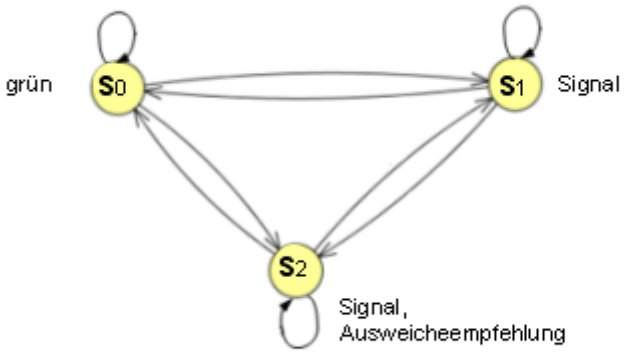

Abbildung 2.2: Beispiel einer Kripke Struktur (Quelle Selbst erstellt)

akustisches Signal. Sollte ein Zusammenstoß von dem System berechnet werden, so gibt das System eine Ausweicheempfehlung [1]. Formal kann das beschriebene Systemverhalten als eine Kripke Struktur entsprechend der obigen Definition folgendermaßen dargestellt werden:

- S=$\{s_0, s_1, s_2\}$
- S_0=$\{s_0\}$
- R=$\{(s_0, s_1), (s_1, s_0), (s_1, s_2), (s_2, s_1), (s_0, s_2), (s_2, s_0), (s_0, s_0), (s_1, s_1), (s_2, s_2)\}$
- L(s_0)= green, L(s_1)= Signal, L(s_2)= Signal, Ausweicheempfehlung

Eine Sicherheitseigenschaft, die im obigen System gelten muss ist z.B. AG (Ausweicheempfehlung $\rightarrow$ $\neg$grün $\vee$Signal).

3 Temporale Logik

Damit ein Model Checker eine bestimmte Eigenschaft des Systems verifizieren kann, muss sie in Form eines logischen Ausdrucks vorliegen. Zu diesem Zweck werden Temporale Logiken verwendet. Die Temporale Logik stellt neben den Operatoren der klassischen Logik weitere Operatoren zur Verfügung, die die Darstellung zeitlicher Zusammenhängen ermöglichen. Dabei werden ebenso 2 Pfadquantoren definiert - always und exist.

Operatoren:

- F(in the future) eine Eigenschaft wird irgendwann erfüllt.
- X (next time) eine Eigenschaft wird im nächsten Zustand des Systems erfüllt.
- G (globally) eine Eigenschaft wird in jedem Zustand des Systems erfüllt.
- U (until) eine Eigenschaft gilt solange bis eine andere Eigenschaft entlang des Pfades vorkommt.
- R (realese) Wenn p und s zwei Eigenschaften sind, dann bedeutet pRs, dass s solange gilt, bis p vorkommt. Dabei gilt s auch in dem ersten Zustand, in dem p erfüllt wird. Falls s gar nicht vorkommt, gilt p entlang des ganzen Pfades.

Quantoren:

- E(exists) eine Eigenschaft gilt mindestens entlang eines Pfades.
- A(always) eine Eigenschaft gilt entlang aller Pfade

Es existieren verschiedene Varianten der temporale Logik wie LTL(Linear Time Temporal Logic),CTL (Computation Tree Logic) etc. Sie unterscheiden sich in der Verfügbarkeit der temporalen Operatoren und Pfadquantoren sowie deren Semantik [2]. LTL Formeln beziehen sich auf einzelne Pfade des Systems, wobei die Zeit als eine lineare Sequenz betrachtet wird. Deswegen entfallen die Pfadquantoren. Bei CTL Formeln wird die Zeit als eine verzweigte Struktur betrachtet. Es werden die folgenden Basisoperatoren verwendet, welche die Gültigkeit einer Eigenschaft φ bzw. ψ folgendermaßen festlegen:

1. AXφ entlang aller Pfade im nächsten Zustand gilt φ.
2. EXφ entlang wenigstens eines Pfades im nächsten Zustand gilt φ.
3. AFφ entlang aller Pfade irgendwann gilt φ.
4. EFφ entlang wenigstens eines Pfades irgendwann gilt φ.
5. AGφ entlang aller Pfade in allen Zuständen irgendwann gilt φ.
6. EGφ entlang wenigstens eines Pfades in allen Zuständen gilt φ.

7. φAUψ entlang aller Pfade irgendwann gilt ψ, wobei in allen vorherigen Zuständen (entlang aller Pfaden) φ gilt.
8. φEUψ es gibt wenigstens einen Pfad auf dem irgendwann ψ gilt, wobei φ in allen vorherigen Zuständen (entlang des Pfades) gilt.

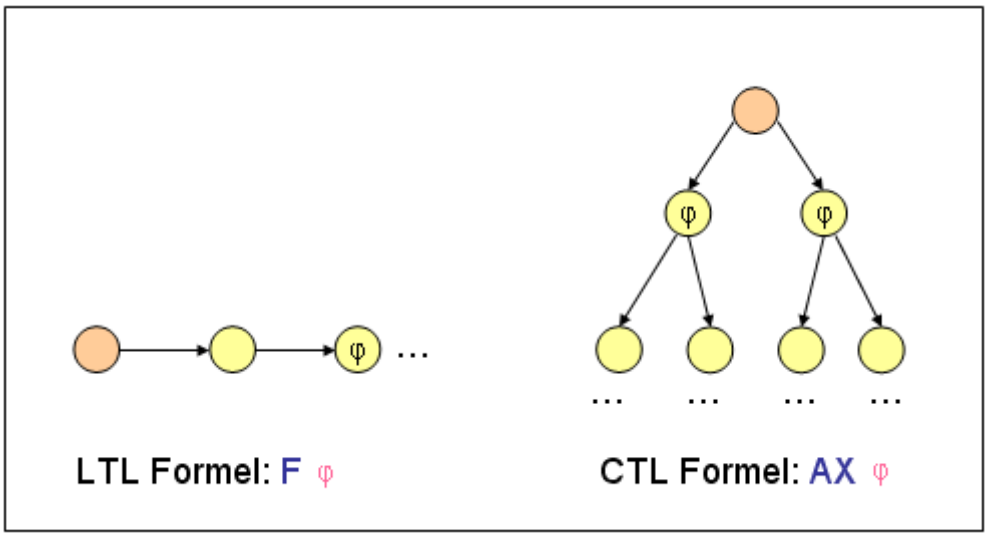

Abbildung 3.1: LTL und CTL Formeln (Quelle Selbst erstellt)

Abbildung 3.2 zeigt zwei Berechnungsbäume, deren Anfangszustände eine LTL bzw. CTL Formel erfüllen. Die gezeigten Strukturen werden von dem Model Checker generiert, nachdem die zu prüfenden Eigenschaften von einem Benutzer eingegeben wurden. In dem ersten Fall gilt die LTL Formel F φ, d.h. irgendwann in der Zukunft gilt φ. In dem zweiten Fall gilt die CTL Formel AX φ, d.h. in allen Nächstzuständen gilt φ.

Ist φ eine Eigenschaft, dann bedeutet K,s $\models \varphi$, dass φ im Zustand s der Kripke Struktur K erfüllt ist. Wir sagen, dass φ in K gültig ist (K $\models \varphi$), falls K,s $\models \varphi$ für alle s $\in$ S gilt. Also bezieht sich das Modell Checking Problem bei CTL Formeln auf die Feststellung, ob eine CTL Formel in einer Kripke Struktur gilt oder nicht. Bezogen auf Kripke Strukturen kann die CTL Semantik induktiv folgendermaßen dargestellt werden [3]:

- wenn $\varphi \in$ P, dann K,s $\models \varphi$ gdw. $\varphi \in$ L(s)
- K,s $\models \neg \varphi$ gdw. K,s $\not\models \varphi$
- K,s $\models \varphi \vee \psi$ gdw. K,s $\models \varphi$ oder K,s $\models \psi$
- K,s $\models$ AXφ gdw. für alle Pfade $\pi = s_0$,s_1,s_2,... für die gilt s $= s_0$,haben wir K,$s_1 \models \varphi$
- K,s $\models$ EXφ gdw. es existiert wenigstens einen Pfad $\pi = s_0, s_1, s_2$,... für den gilt s $= s_0$, haben wir K,$s_1 \models \varphi$
- K,s $\models$ A(φUψ) gdw. für alle Pfade $\pi = s_0, s_1, s_2$,... für die gilt s $= s_0$, es existiert ein i $\geq$ 0 so dass K,$s_i \models \psi$ und für alle 0 $\leq$ j $<$ i, K,$s_(j) \models \varphi$

- K,s $\models$ E(φUψ) gdw. es gibt wenigstens einen Pfad $\pi = s_0, s_1, s_2$,... für den gilt s = s_0 und eine Zahl i $\geq$ 0 so dass K,s_i $\models \psi$ und für alle 0 $\leq$ j $<$ i, K,$s_(j)$ $\models \varphi$

Nach dieser Definition gilt in der Kripke Struktur von Abb. 2.2 die Formel K$\models$ AG(Ausweicheempfehlung $\rightarrow$ rot $\wedge$ Signal).

4 Model Checking Algorithmen

Die Gültigkeit von LTL und CTL Formeln kann mit Hilfe von Model Checking Algorithmen überprüft werden. Beispielsweise prüft der Algorithmus von Abb. 4.1 die die Gültigkeit der Formel EF p.

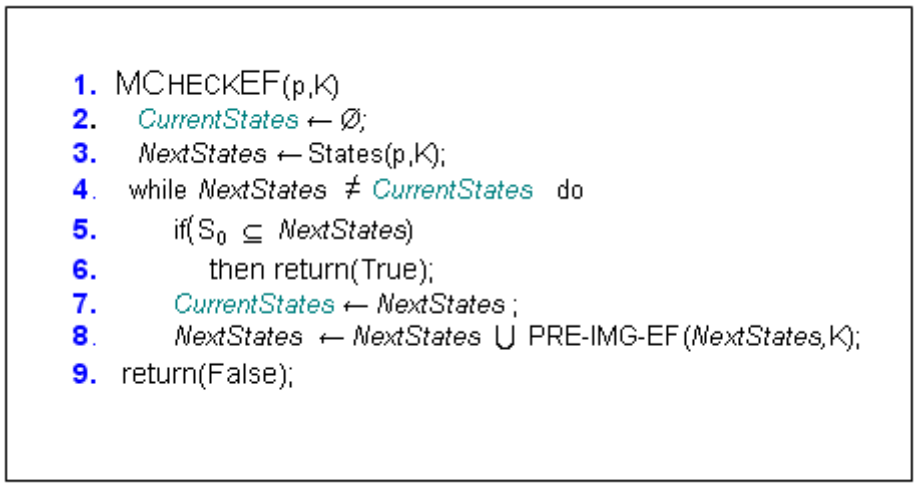

Abbildung 4.1: Model Checking EF p (Quelle[3])

Als Input werden eine Kripke Struktur und eine propositionale Formel p benutzt. Zuerst berechnet der Algorithmus die Zustände in denen p gilt (Zeile 3):

$$States(p, K) = \{s \in S : p \in L(s)\}$$

Als nächstes wird der Zustandsraum von K untersucht. In NextStates werden die von PRE-IMG-EF gelieferten Zustände gespeichert. Die Funktion PRE-IMG-EF ist so definiert, dass sie eine Menge von Zuständen zurückgibt, mit wenigstens einem Nachfolgezustand, der zu States gehört, also in dem p gilt:

$$PRE - IMG - EF(States, K) = \{s \in S : \exists s'.(s' \in States \wedge R(s, s'))\}$$

Die Schleife terminiert erfolgreich wenn NextStates alle Anfangszustände enthält.

Mit diesem Algorithmus kann die Formel EF grün in der Kripke Struktur von Abb.3.1 geprüft werden. Wir nehmen als Anfangszustand s_1 oder s_2. Der Algorithmus weist den Zustand s_0

der Variable *NextStates* zu. Nach der ersten Iteration ist *NextStates* $= \{s_0, s_1, s_2\}$ und der Algorithmus terminiert mit Ergebnis true.

5 Binary Decision Diagrams

Ein BDD präsentiert Boolean Funktionen als einen gerichteten, azyklischen Graphen. Dabei muss der Graph die Topologie eines Baumes haben. Abb.5.1 zeigt die Präsentation der Funktion $f(x_1, x_2, x_3)$ als binärer Entscheidungsbaum, die durch die gegebene Wahrheitstabelle definiert ist. Wenn die Werte von oben nach unten gelesen werden, entspricht die Reihenfolge den Werten der Endknoten des Baumes von links nach rechts [5].

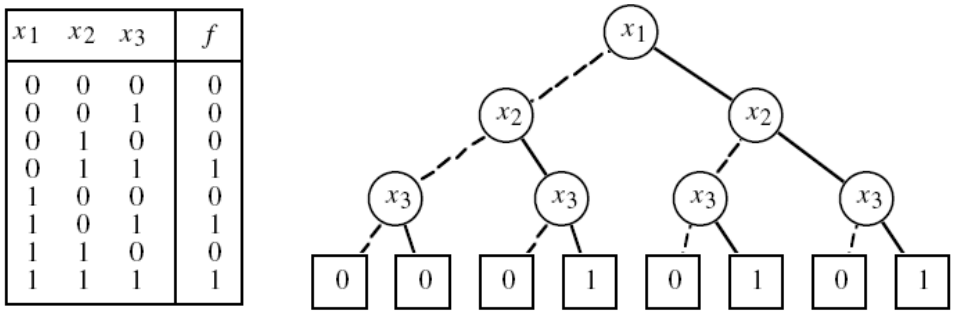

Abbildung 5.1: Wahrheitstabelle und Entscheidungsbaum Darstellung einer Boolean Funktion (Quelle[5])

Jeder interne Knoten des Baumes v ist mit einer Variable var(v) beschriftet - z.B. x_1, x_2 - und hat zwei Kinder - lo(v) und hi(v). lo(v) bedeutet, dass die Variablenbelegung des Elternknoten 0 ist (punktierte Linie) und hi(v), dass die Variablenbelegung des Elternknoten 1 ist (durchgezogene Linie). Jeder Endknoten ist mit 0 oder 1 bzw. true oder false beschriftet.

Damit ein BDD geordnet ist, muss die Variablenanordnung entlang aller Pfade von dem Wurzel bis zum Knoten gleich sein. D.h. wir führen eine totale Ordnung auf die Menge der Variablen ein, so dass für jeden interne Knoten u und jeder interne Kindknoten v, var(u) $<$ var(v) gilt. Beispielsweise ist in Abb. 5.1 die Reihenfolge $x_1 < x_2 < x_3$. Im Prinzip kann die Variablenanordnung beliebig ausgewählt werden - die BDDs Algorithmen werden bei jeder Anordnung korrekt funktionieren. Für die effiziente symbolische Manipulation ist aber die Anordnung wichtig, weil die Form und Größe eines OBDD (Ordered Binary Decision Diagram) davon abhängig sind. Abb. 5.2 zeigt zwei mögliche Darstellungen des Ausdrucks $a_1.b_1 + a_2.b_2 + a_3.b_3$, wobei . eine UND-Verknüpfung bedeutet und $+$ eine ODER-Verknüpfung. In dem ersten Fall ist die Variablenanordnung $a_1 < b_1 < a_2 < b_2 < a_3 < b_3$ und in dem zweiten Fall $a_1 < a_2 < a_3 < b_1 < b_2 < b_3$. Die erste Anordnung ergibt 2n interne Knoten (n ist der

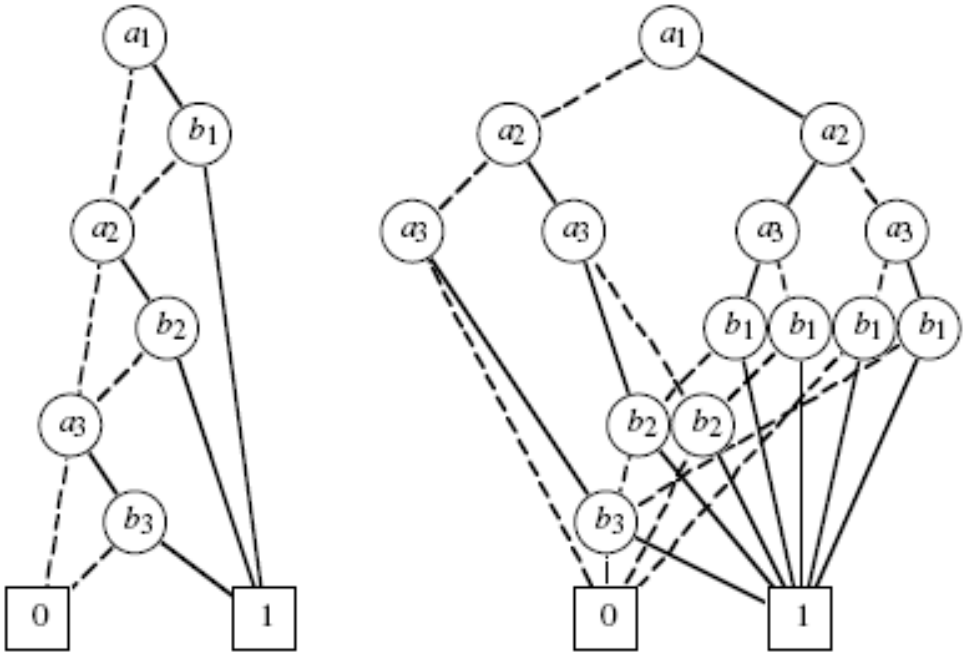

Abbildung 5.2: OBDD Darstellung einer Funktion bei unterschiedlichen Variablenanordnungen (Quelle[5])

größte Wert der Indexmenge {1...n} für die Variable a bzw. b. Die zweite Anordnung ergibt $2(2^n - 1)$ interne Knoten [5].

Bei den meisten Anwendungen, die BDDs verwenden, wird die Variablenanordnung am Anfang ausgewählt und danach werden alle Graphen entsprechend aufgebaut. Die Auswahl erfolgt entweder manuell oder durch heuristische Analyse des zu präsentierenden Systems. Die Heuristiken müssen nicht die beste mögliche Variablenanordnung finden - soweit eine Anordnung gefunden wird, die einen exponentiellen Wachstum vermeidet, sind die Operationen auf BDDs eher effizient [5].

Mit Hilfe von drei Transformationsregeln, die bottom-up angewendet werden, können wir ein geordnetes BDD in einem reduzierten geordneten BDD überführen :

- *Entferne doppelt vorkommende Endknoten*: eliminiere so viele Endknoten, dass es für jede mögliche Beschriftung jeweils einen Endknoten bleibt. Verbinde alle Kanten der eliminierten Knoten mit den gebliebenen Endknoten.
- *Entferne doppelt vorkommende interne Knoten*: Falls zwei interne Knoten dieselbe Kinder haben, eliminiere einer von den Knoten. Verbinde die eingehenden Kanten des eliminierten Knoten mit dem gebliebenen Knoten.
- *Entferne redundante Tests*: Falls die ausgehenden Kanten eines internen Knoten mit demselben Kindknoten verbunden sind, entferne den Knoten. Verbinde die eingehenden Kanten des eliminierten Knoten mit seinem Kindknoten.

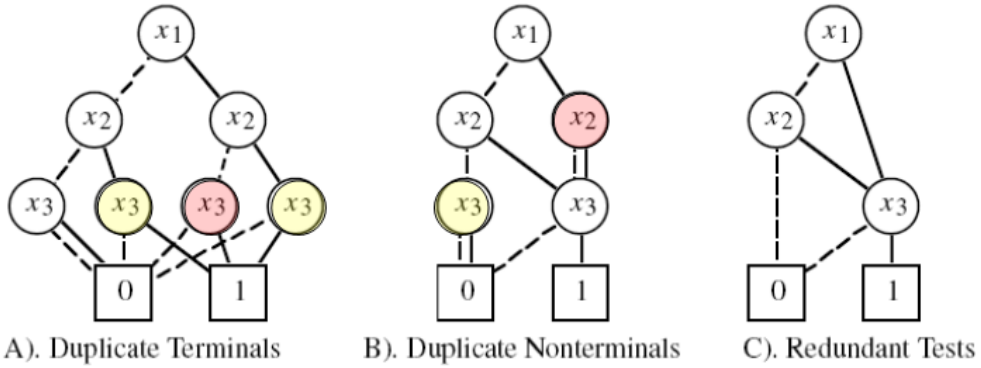

Abbildung 5.3: Reduktion von OBDDs (Quelle[5])

Abb.5.3 verdeutlicht die Anwendung der genannten Transformationsregeln.

BDDs können ebenso genutzt werden, um das Ergebnis von der Anwendung boolescher Operationen zu zeigen - wenn man die Knoten eines BDD vertauscht, bekommt man die Negation des zugehörigen Ausdrucks. Mehrere BDDs können so kombiniert werden, dass sie beispielsweise die Konjunktion bzw. Disjunktion der dahinter stehenden Formeln darstellen.

6 Symbolisches Model Checking

Wie bereits erwähnt bietet das Symbolische Model Checking eine Möglichkeit an, das State Explosion Problem beim expliziten Model Checking zu vermeiden. Es werden keine einzelne Zustände betrachtet, sondern Mengen von Zuständen. Um ein Model Checking Problem symbolisch darzustellen, müssen die zugehörige Kripke Struktur und die Model Checking Algorithmen symbolisch dargestellt werden. Für die symbolische Präsentation der Kripke Struktur, müssen die Mengen von Zuständen und die Übergangsrelation durch logische Formeln ersetzt werden.

Der Prozess des Übergangs zu einer symbolischen Darstellung kann mit drei Schritten beschrieben werden:

1. *Symbolische Präsentation der Menge von Zuständen*: Wir bilden einen Vektor von booleschen Variablen, mit deren Hilfe alle elementaren Aussagen aus P dargestellt werden können z.B. x={grün, Signal, Ausweicheempfehlung}. Jeder Zustand s wird durch eine Formel ξ(s) dargestellt - z.B. für die Kripke Struktur in Abb. 2.2 können die folgenden Formeln definiert werden:

   $\xi(s_0) =$ grün $\wedge \neg Signal \wedge \neg Ausweicheempfehlung \cong s_0$(x) = <1,0,0>
   $\xi(s_1) = \neg$ grün $\wedge Signal \wedge \neg Ausweicheempfehlung \cong s_1$(x) = <0,1,0>
   $\xi(s_2) = \neg$ grün $\wedge Signal \wedge Ausweicheempfehlung \cong s_2$(x) = <0,1,1>

   Wenn die Formeln von zwei oder mehrere Zustände übereinstimmen, werden die entsprechenden Zuständen zu einer Menge von Zuständen zusammengefasst. So können mit einer einzigen Formel, mehrere Zustände dargestellt werden. Eine Formel kann als ein BDD betrachtet werden. Die Disjunktion aller eindeutigen Formeln über einer Kripke Struktur bildet ein BDD, welches das ganze System beschreibt.

   Wenn Q eine Menge von Zuständen ist, dann ist $\xi(Q)$ gleich der Disjunktion aller Formeln $\xi(s)$, für die s $\in$ Q gilt:

   $$\xi(Q) = \bigvee_{s \in Q} \xi(s)$$

   Beispielsweise wenn Q = $\{s_1, s_2)$ gilt, ist $\xi(Q) = \xi(s_1) \wedge \xi(s_2)$.

2. *Symbolische Präsentation der Übergangsrelation*: Zu diesem Zweck werden zwei Vektoren von booleschen Variablen benötigt - einer, der die Variablen des aktuellen Zustands enthält x = $<x_1, ...x_n>$, und zweiter der die Variablen des nächsten Zustands speichert x' = $<x'_1, ...x'_n>$. Die Formeln/BDDs, die sich auf die nächsten Zustände

beziehen werden mit ξ'_s bzw. ξ'_Q bezeichnet. Die Menge von Zuständen der Nächst-Zustandsvariblen können folgendermaßen dargestellt werden:

$$\xi'(s) = \xi(s)[x \leftarrow x']$$

$\phi[x \leftarrow x']$ ist die Operation Vorwärtsschiften und dient zur Transformation der aktuellen Menge von Zuständen in der entsprechenden Menge der Nächst-Zustandsvariablen. Die umgekehrte Transformation kann mit der Operation Rückwärtsschiften $\phi[x' \leftarrow x]$ durchgeführt werden.

Also können wir für die Kripke Struktur in Abb. 2.2 den Übergang von s_0 nach s_2 folgendermaßen darstellen:

$$\xi(< s_0, s_2 >) = \xi(s_0) \wedge \xi'(s_2)$$

oder konkret

$\xi(< s_0, s_2 >) =$ (grün $\wedge \neg Signal \wedge \neg Ausweicheempfehlung) \wedge (\neg$ grün' $\wedge Signal' \wedge Ausweicheempfehlung')$

Formal ist die Übergangsrelation R einer Kripke Struktur eine Menge von Übergängen:

$$\xi(R) = \bigvee_{r \in R} \xi(r)$$

3. *Symbolische Präsentation der Model Checking Algorithmen*:

   Um explizit zu zeigen, dass eine Formel/BDD $\xi(Q)$ die Variablen $x_i, ..., x_n$ enthält, wird im folgenden den Ausdruck Q(x) statt $\xi(Q)$ verwendet. Entsprechend wird Q(x') mit $\xi'(Q)$ dargestellt. S(x), R(x,x') und $S_0(x)$ präsentieren die Zustände, die Übergangsrelation und die Anfangszustände einer Kripke Struktur. Um boolesche Formeln symbolisch manipulieren zu können, ist es hilfreich quantifizierte boolesche Formeln (QBFs) zu verwenden. QBFs erweitern die Aussagenlogik um zwei Elemente - Existenz- und Universalquantoren [3]. D.h. aussagenlogische Formeln sind QBFs und QBFs können aussagenlogische Formeln sein.

   Mit einer einzigen Substitution $[x' \leftarrow x]$ (x wird durch x' ersetzt) auf einem QBF können wir den Übergang von x nach x' aller Zustände, die in Q enthalten sind und mit x' in Relation stehen, ausdrücken. Wenn wir die Substitution $[x' \leftarrow x]$ auf die Formel $\exists x.(R(x, x') \wedge Q(x))$ anwenden, ist das Ergebnis

   $$(\exists x.(R(x, x') \wedge Q(x)))[x' \leftarrow x]$$

   Also wir können den Aufruf der Funktion PRE-IMG-EF(Q) in dem Algorithmus von Abb. 4.1 mit ξ(PRE-IMG-EF(Q)) ersetzen, um das Ganze symbolisch darzustellen:

   $$\exists x'.(R(x, x') \wedge Q(x'))$$

   Um einen Model Checking Algorithmus symbolisch zu präsentieren, müssen alle Operationen gecastet werden, so dass sie auf Mengen von Zuständen anwendbar sind und die Funktionsaufrufe durch die entsprechenden symbolischen Formeln ersetzt werden.

7 Planen für erreichbare Ziele

Wenn erreichbare Ziele betrachtet werden, ist die Plan-Domain ein nicht-deterministisches Zustandsübergangssystem. Formal kann dieses System als $\sum = (S, A, \gamma)$ beschrieben werden, wobei:

- S eine endliche Menge von Zuständen ist und
- A eine endliche Menge von Aktionen.
- γ : S x A $\rightarrow 2^S$ ist die Zustand-Übergangsfunktion.

Der Nichtdeterminismus wird durch γ ausgedrückt: $\gamma(s, a)$ kann für ein beliebig ausgwähltes a (eine Aktion) und ein beliebig ausgewähltes s (ein Zustand) eine Menge von direkten Nachfolgezuständen als Ergebnis haben.

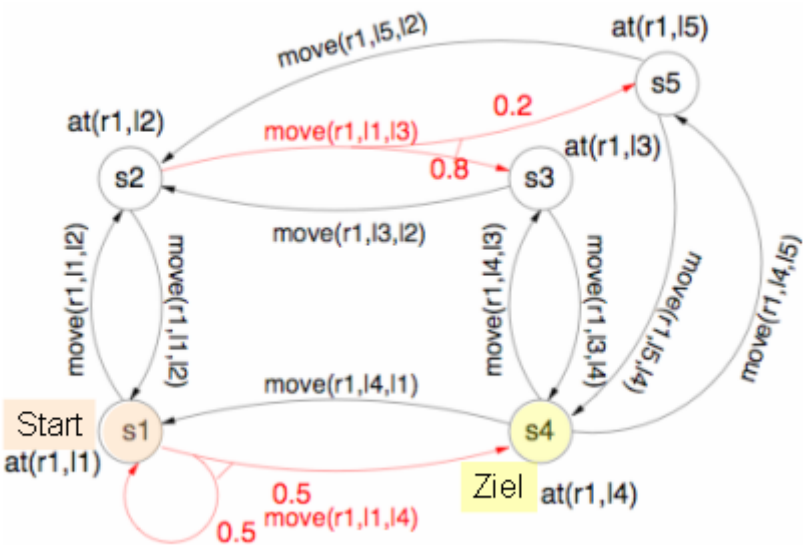

Abbildung 7.1: Nicht-deterministisches Zustandsübergangssystem (Quelle[3])

Ein Beispiel für nicht-deterministisches Zustandsübergangssystem zeigt Abb.7.1. Wir betrachten eine vereinfache DWR (dock worker robots) Domain. Auf einem Hafen befinden sich mehrere Robotkarren, die Container von einer bis zum anderen Stelle übertragen müssen. Die at-Terme beschreiben Zustände, die move-Terme mögliche Aktionen. Die Zustände

werden mit s1, s2, s3, s4 und s5 beschriftet, r steht für Robotkarre und l für location. Es gibt zwei nicht-deterministische Aktionen :

- move(r1,l1,l2)
- move (r1,l1,l4)

und zwei Quellen für Nichtdeterminismus:

- γ(s2,move(r1,l2,l3)) = {s3,s5}
- γ(s1,move(r1,l1,l4)) = {s1,s4}

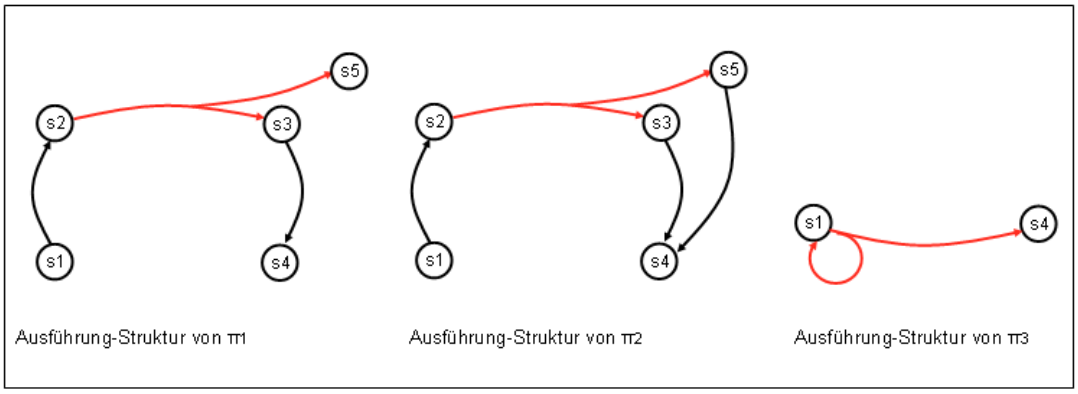

Abbildung 7.2: Ausführung-Strukturen (Quelle[3])

Ein Plan wird auch mit dem Begriff Strategie bezeichnet. Eine Strategie π für eine Plan-Domain $\sum = (S, A, \gamma)$ ist eine Menge von Paare (s,a) mit s $\in$ S und a $\in$ A(s). Für jeden Zustand s muss höchstens eine Aktion a definiert sein, wobei (s,a) $\in \pi$ gilt. Dann ist die Menge von Zuständen einer Strategie $S_\pi = \{s|(s, a) \in \pi\}$.

Für die Domain in Abb. 7.1 können beispielhaft die folgenden Strategien definiert werden:

- π_1 = {(s1, move(r1,l1,l2)), (s2, move(r1,l2,l3)), (s3, move(r1,l3,l4))}
- π_2 = {(s1, move(r1,l1,l2)), (s2, move(r1,l2,l3)), (s3, move(r1,l3,l4)), (s5, move(r1,l3,l4))}
- π_3 = {(s1, move(r1,l1,l4))}

Sie zeigen unterschiedliche Wege, um den Endzustand s4 zu erreichen. Die Ausführung einer Strategie in einer Plan-Domain wird mit Ausführung-Strukturen dargestellt. Eine Ausführung-Struktur ist ein gerichteter Graph, dessen Knoten diejenigen Zustände der Domain darstellen, welche durch die Anwendung von Aktionen, die Teil einer bestimmten Strategie sind, erreicht werden können. Mit Pfeile werden mögliche Übergänge dargestellt, die sich aus der Anwendung von Aktionen ergeben. Eine Ausführung-Struktur bezieht sich immer auf einer konkreten Strategie.

Formal wird eine Ausführung-Struktur über eine Plan-Domain $\sum = (S, A, \gamma)$ mit $\sum_\pi = (Q, T)$ dargestellt. Dabei gilt $Q \subseteq S$ und T $\subseteq$ S x S. Abb.7.2 zeigt die Ausführung-Strukturen der drei beschriebenen Strategien.

Eine Strategie ist eine Lösung eines Plan-Problems. Ein Plan-Problem ist ein Tripel $(\sum, S_0, S_g)$, wobei $\sum = (S, A, \gamma)$ eine Domain ist, $S_0 \subseteq S$ eine Menge von Anfangszuständen und $S_g \subseteq S$ eine Menge von Zielzuständen. Es können drei unterschidliche Arten von Lösungen eines Plan-Problems definiert werden:

- *Schwache Lösungen* können einen Zielzustand erreichen, dies ist aber nicht gewährleistet. Ein Plan ist eine schwache Lösung, wenn es wenigstens einen endlichen Pfad existiert, der ein Ziel erreicht.
- *Starke Lösungen* erreichen garantiert einen Zielzustand. Ein Plan ist eine starke Lösung, wenn alle Pfade endlich sind und jeder Endzustand einen Zielzustand darstellt.
- *Starke zyklische Lösungen* erreichen garantiert einen Zielzustand unter der Annahme, dass ein System Fairness aufweist. Fairness ist eine Art Lebendigkeitseigenschaft (Liveness) und bedeutet hier, dass die auf einem Pfad vorkommenden Schleifen, irgendwann verlassen werden.

Entsprechend dieser Kriterien kann π_1 als schwache Lösung klassifiziert werden, π_2 als starke Lösung und π_3 als starke zyklische Lösung.

Mit Hilfe von Plan-Algorithmen können schwache, starke oder starke zyklische Lösungen eines Plan-Problems erzeugt werden. Schwache Lösungen sind wie optimistische Plans und starke Lösungen wie sichere Plans. Allerdings gibt es Situationen, in denen schwache Lösungen nicht akzeptabel sind und starke Lösungen nicht existieren. In solchen Fällen können starke zyklische Lösungen sehr hilfreich sein [3]. Abb. 7.3 zeigt ein Algorithmus, der schwache Lösungen generieren kann.

```
Weak-Plan(P)
    π ← failure; π' ← ∅
    While π' ≠ π and S0 ⊈ (Sg ∪ Sπ') do
        PreImage ← WeakPreImg(Sg ∪ Sπ')
        π'' ← PruneStates(PreImage, Sg ∪ Sπ')
        π ← π'
        π' ← π' ∪ π''
    if S0 ⊆ (Sg ∪ Sπ') then return(MkDet(π'))
        else return(failure)
end
```

Abbildung 7.3: Weak-Plan Algorithmus (Quelle[3])

Die Funktionsaufrufe innerhalb des Algorithmus sind wie folgt definiert:

- WeakPreImg(S)= $\{(s, a) : \gamma(s, a) \cap S \neq \oslash\}$ gibt alle (s,a)-Paare zurück, für die wenigstens einen Nahfolger in S liegt.

- PruneStates($\pi', S) = \{(s, a) \in \pi | s \notin S\}$ entferne alle (s,a)-Paare, deren Zustand s bereits in der Menge S enthalten ist.
- MkDet(π') gibt eine deterministische Strategie zurück. Dabei ist π' eine eventuell nichtdeterministische Strategie, von der bestimmte (s,a)-Paare entfernt werden, damit jeder von den gebliebenen Zuständen nur einen Nachfolgezustand hat.

Wenn wir den Algorithmus auf die Domain von Abb.7.1 anwenden, geht der Algorithmus folgendermaßen vor:

- 1.Durchlauf

  $\pi = failure$
  $\pi' = \oslash$
  $S_{\pi'} = \oslash$
  $S_g \cup S_{\pi'} = \{S_4\}$
  $PreImage = \{(s1, move(r1, l1, l4)), (s3, move(r1, l3, l4)), (s5, move(r1, l5, l4))\}$
  $\pi'' = PreImage = \{(s1, move(r1, l1, l4)), (s3, move(r1, l3, l4)), (s5, move(r1, l5, l4))\}$

  $\pi \leftarrow \pi' = \oslash$
  $\pi \leftarrow \pi' \cup \pi'' = \{$(s1,move(r1,l1,l4)), (s3,move(r1,l3,l4)), (s5,move(r1,l5,l4))$\}$

- 2.Durchlauf

  $\pi = \oslash$
  $\pi' = \{(s1, move(r1, l1, l4)), (s3, move(r1, l3, l4)), (s5, move(r1, l5, l4))\}$
  $S_{\pi'} = \{s1, s3, s4, s5\}$
  $S_g \cup S_{\pi'} = \{s1, s3, s4, s5\}$
  $S_0 \subseteq S_g \cup S_{\pi'}$
  $MkDet(\pi') = \pi'$

Der Algorithmus terminiert nach dem zweiten Durchlauf und liefert als Ergebnis die Strategie:

$\pi' = \{(s1, move(r1, l1, l4)), (s3, move(r1, l3, l4)), (s5, move(r1, l5, l4))\}$

Das ist eigentlich eine starke zyklische Lösung, aber sie ist natürlich auch schwache Lösung. Jede starke Lösung ist auch eine schwache oder starke zyklische Lösung, und jede starke zyklische Lösung ist auch eine schwache Lösung.

Mit dem gezeigten Algorithmus können starke Lösungen erzeugt werden, wenn statt die Funktion WeakPreImg die Funktion StrongPreImg aufgerufen wird:

$StronPreImg(S) = \{(s, a) | \gamma(s, a) \neq \oslash \wedge \gamma(s, a) \subseteq S\}$

Da der Algorithmus so implementiert ist, dass er mit Mengen von Zuständen funktioniert, kann der zugehörige Zustandsraum mit BDDs aufgebaut werden und dank dessen effizient bearbeitet werden.

8 MBP Planer

Mit dem MBP Planer können Pläne für nicht-deterministische Domains erzeugt werden. Zu diesem Zweck werden Techniken des Symbolischen Model Checking verwendet. Als Input wird eine Domain-Beschreibung in NuPDDL benutzt, die in einem endlichen Automaten überführt wird (eine Kripke Struktur kann als endlicher Automat betrachtet werden), dessen Mengen von Zuständen als propositionale Formeln symbolisch dargestellt werden. NuPDDL ist eine Erweiterung von PDDL 2 (Planning Domain Definition Language), mit deren Hilfe Unsicherheit und Nichtdeterminismus dargestellt werden können.

Die Suche durch den Zustandsraum wird als eine Menge von logischen Transformationen auf propositionale Formeln realisiert. MBP benutzt reduzierte geordnete BDDs, die eine kompakte Darstellung und effektive Manipulation von propositionalen Formeln erlauben.

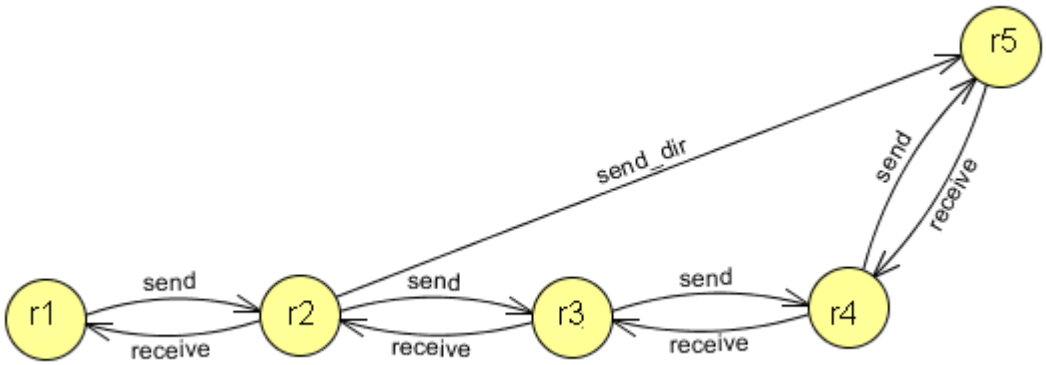

Abbildung 8.1: Ein System von Routern (Quelle Selbst erstellt)

Abb.8.1 zeigt die grafische Darstellung einer stark-vereinfachten Internet Problem-Domain.

```
(define(problem pack_problem)
    (:domain i_simulation)
    (:init (=(package)r1))
    (:stronggoal (=(package)r5)))
```

Abbildung 8.2: NuPDDL Problembeschreibung (Quelle Selbst erstellt)

Es handelt sich von einem Programm, dass die Übertragung von Paketen im Internet simulieren soll.

Es wird nach dem kürzesten Weg zwischen Routern gesucht, damit ein Paket möglichst schnell ans Ziel ankommt. Abb.8.2 zeigt die Problemspezifikation in NuPDDL und Abb.8.3 die Plan-Domain, ebenso in NuPDDL. Wenn als Startzustand r1 genommen wird und als Endzustand r5, generiert der MBP Planer die in Abb.8.4 gezeigte Lösung. Wir können aber als Anfangs- und Endzustand beliebige Zustände auswählen, indem wir die Problembeschreibung durch triviale Änderungen leicht modifizieren.

```
(define (domain i_simulation)
(:types space)
(:constants r1 r2 r3 r4 r5 -space)
(:functions (package) - space)

(:action send
 :precondition (or (=(package) r1)
                   (=(package) r2)
                   (=(package) r3)
                   (=(package) r4))
 :effect (and
               (when  (=(package) r1)
               (assign(package) r2))

               (when (=(package) r2)
               (assign(package) r3))

               (when (=(package) r3)
               (assign(package) r4))

               (when (=(package) r4)
               (assign(package) r5)) ))

(:action receive
 :precondition (or (=(package)r5)
                     (=(package)r4)
                     (=(package)r3)
                     (=(package)r2))
 :effect (and
               (when (=(package)r5)
               (assign(package)r4))

               (when (=(package)r4)
               (assign(package)r3))

               (when (=(package)r3)
               (assign(package)r2))

               (when (=(package)r2)
               (assign(package)r1))  ))


(:action send_dir
 :precondition (=(package)r2)

 :effect       (when(=(package)r2)
               (assign(package)r5)) ))
```

Abbildung 8.3: NuPDDL Domainbeschreibung (Quelle Selbst erstellt)

```
****************************************
**** Total preprocessing time = 0.010 seconds ******
****************************************

Problem class: STRONG

The plan has been found.
;;; generated plan
(define (plan generated_plan)
      (:domain i_simulation)
      (:problem pack_problem)
      (:body (repeat
        (switch
          (case (and (=(package) r5))
            (done))
          (case (and (=(package) r1))
            (action (send)))
          (case (and (=(package) r2))
            (action (send_dir)))
          (case (and (=(package) r3))
            (action (send)))
          (case (and (=(package) r4))
            (action (send)))
          (else (fail))
        ))
      )
)
;;; end generated plan
```

Abbildung 8.4: MBP Plan (Quelle Selbst erstellt)

Literaturverzeichnis

[1] Tobias Opel : *Model Checking Beispiele aus der Praxis*, Seminararbeit, 2004

[2] Annekatrin Tretow : *Model Checking - Grundlagen*, Seminararbeit, 2004

[3] M. Ghallab, D. Nau, P. Traverso: *Automated Planning: Theory and Practice*, Morgan Kaufmann, 2004

[4] Florian Heyer : *Model Checking 1 - CTL*, www.se-uni.hanover.de, 2006

[5] Randal E. Bryant : *Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams*, Fujitsu Laboratories, 1992