Erweiterung eines BDD-basierten LTL-Modelcheckers

Bachelor-Arbeit

vorgelegt von

Martin Schwarick

geboren am 02.02.1980 in Herzberg/Elster

Betreuer und Prüfer: Prof. Dr.–Ing. Monika Heiner

Cottbus, 26. März 2004

Hiermit versichere ich, dass die vorliegende Bachelor-Arbeit von mir selbständig verfasst wurde. Die verwendeten Hilfsmittel und Quellen sind im sind Literaturverzeichnis vollständig angegeben.

Cottbus, 26. März 2004

Inhaltsverzeichnis

1 Einleitung

Die computergestützte Elektronik hält in immer höherem Maße Einzug in sämtliche Lebensbereiche. Die dabei entstehenden Systeme werden ständig größer und komplexer. Ein eventuelles Fehlverhalten oder Versagen einzelner Systemkomponenten ist dabei unerwünscht und hätte nicht selten katastrophale Folgen. Deshalb ist es erforderlich, solche Systemfehler zu identifizieren und gegebenenfalls zu beseitigen. Zu diesem Zweck wurden so genannte temporale Logiken entwickelt, mit denen sich bestimmte Systemeigenschaften formulieren lassen. Die Überprüfung, ob ein System einer in dieser Form spezifizierten Eigenschaft entspricht, wird als Modelchecking bezeichnet. Jochen Spranger entwickelte im Rahmen seiner Dissertation [Spr01] den Modelchecker EEMC, der eine mögliche Automatisierung dieses Vorgangs realisiert. Das zu überprüfende System liegt dabei als Petrinetz vor, und die an das System gestellten Forderungen werden in linearer temporaler Logik, kurz LTL, formuliert. Die beim Modelchecking überprüften Mengen von Systemzuständen werden dabei durch Binäre Entscheidungsgraphen repräsentiert.
Ziel dieser Arbeit ist die Erweiterung des LTL-Modelcheckers EEMC um folgende drei Funktionalitäten:

1. Implementierung einer Schnittstelle für das BDD-Visualisierungstool VisualizeSize, das von Dirk Beyer und Michael Vogel in der Programmiersprache Java implementiert wurde;
2. Ermittlung eines Gegenbeispiels für die Symbolische Tiefensuche im Falle eines negativen Ergebnisses des Modelcheckings;
3. Erweiterung des Kantenspektrums um Inhibitorkanten.

Diese Arbeit stützt sich inhaltlich auf die Dissertation von Jochen Spranger [Spr01]. Formalisierungen der hier nur verbal erläuterten Sachverhalte können dort bei Bedarf nachgeschlagen werden.

2 BDD-Visualisierung

2.1 Binäre Entscheidungsgraphen – Binary Decision Diagrams

In vielen Bereichen der Informatik, deren Schwerpunkt auf der Darstellung und Manipulation von booleschen Funktionen liegt, spielen binäre Entscheidungsgraphen (engl. Binary Decision Diagrams), kurz BDDs, eine sehr bedeutende Rolle. Sie sind eine sehr geeignete und vor allem kompakte Darstellungsmöglichkeit für Mengen von Belegungen boolescher Variablen, die bei geschickter Implementierung leistungsfähige Funktionen zur Verfügung stellen.

Eine andere Möglichkeit der Darstellung boolescher Funktionen sind binäre Entscheidungsbäume. Die folgende Abbildung zeigt einen Entscheidungsbaum für den Vergleich zweier zweistelliger Vektoren, also (x1, x2) und (y1, y2) mit x1 = y1 und x2 = y2. Ein Knoten repräsentiert eine Entscheidung über die Belegung der jeweiligen Variablen durch seine beiden ausgehenden Kanten. Ein Pfad durch einen solchen Entscheidungsbaum repräsentiert eine Belegung der Funktion, deren Wert dem Terminalknoten entnommen wird. Dabei ist auf einem Pfad jede Variable genau einmal vertreten. Geht man also auf einem solchen Pfad über den linken Sohn eines Knotens weiter, so ist die Belegung der Variablen des Knotens gleich 0, im anderen Falle entsprechend 1.

Schon auf den ersten Blick ist ersichtlich, dass dieser Entscheidungsbaum sehr viele Informationen mehrfach enthält. Wenn man diese überflüssige Redundanz entfernt, erhält man einen Binären Entscheidungsgraphen.

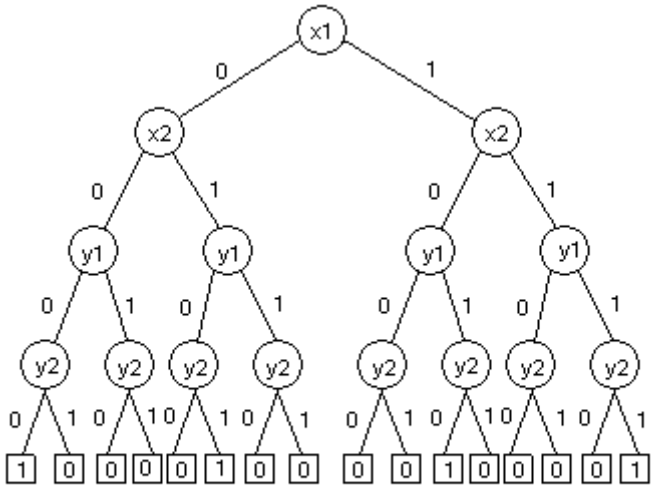


Abbildung 2.1: Entscheidungsbaum für 2-Bit-Vergleich

Dies lässt sich mit Hilfe der folgenden Reduktionsregel bewerkstelligen:
- Eliminierung isomorpher Unterbäume.
Zwei Unterbäume F und G sind isomorph, wenn es zwischen den sie repräsentierenden Knotenmengen eine bijektive Abbildung gibt.
Das Resultat ist ein gerichteter, azyklischer Graph mit einer Menge von Entscheidungsknoten und nur noch zwei Terminalknoten. Ein solcher Entscheidungsknoten ist, wie schon im Entscheidungsbaum, durch die jeweilige Variable und durch eine 0- und eine 1-Kante, auch Low- und High-Kante genannt, beschrieben.

Ist über der Variablenmenge eine feste Ordnung definiert, so spricht man auch von einem OBBD (Ordered Binary Decision Diagram). In einem OBDD kann es Knoten geben, deren ausgehende Kanten auf den gleichen Knoten verweisen, und somit keine Entscheidung im eigentlichen Sinne darstellen. Wendet man auf den OBDD eine zweite Regel an, die genau solche Knoten eliminiert, erhält man einen reduzierten OBDD, einen ROBDD (Reduced Ordered Binary Decision Diagram).

Ein Reduzierter Geordneter Binärer Entscheidungsgraph geht also aus einem Binären Entscheidungsbaum durch ein erschöpfendes Anwenden zweier Reduktionsregeln hervor. Im Falle des obigen Entscheidungsbaumes ergibt sich folgender BDD.

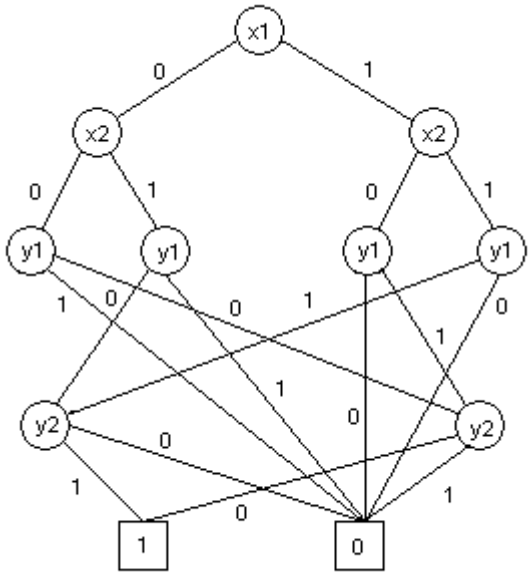


Abbildung 2.2: OBDD für 2-Bit-Vergleich

Die Anzahl der benötigten Knoten konnte sichtlich reduziert werden. Dennoch lässt sich selbst für dieses kleine Beispiel die Knotenanzahl weiter verringern. Wenn man die Reihenfolge der Variablen vorteilhaft anordnet, in diesem Falle der Form {x1, y1, x2, y2}, entsteht folgender BDD.

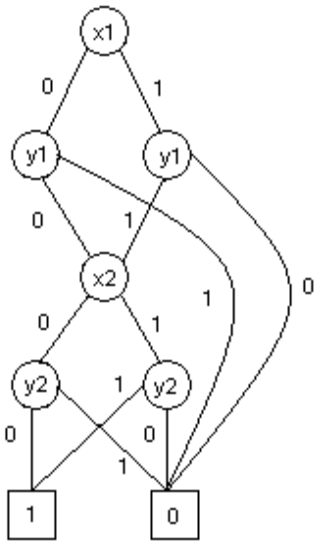


Abbildung 2.3: OBDD für 2-Bit-Vergleich

Die fest gewählte Ordnung über einer Variablenmenge hat also entscheidenden Einfluss auf die Größe des zur Repräsentation der Belegungsmenge konstruierten BDD.

2.2 ZBDDs

Wenn man einen OBDD dazu benutzt, die Markierungen des Erreichbarkeitsgraphen eines sicheren Petrinetzes zu repräsentieren, ist es möglich, ihn einer anderen Art der Reduktion zu unterziehen. Wenn die 1-Kante eines Knotens zum 0-Terminalknoten des BDD führt, so kann dieser Knoten eliminiert werden. Die eingehenden Kanten des besagten Knotens werden auf den Knoten umgelenkt, der auf dessen 0-Kante zeigt. Dann erhält man einen so genannten Zero-surpressed BDD, oder kurz ZBDD. ZBDDs wurden von S. Minato entwickelt und resultieren aus der Beobachtung, dass der Erreichbarkeitsgraph eines sicheren Petrinetzes bzgl. der Zustandsvektoren aber auch hinsichtlich der Anzahl der überhaupt erreichbaren Zustände sehr dünn besetzt ist. In einem solchen Fall ermöglicht die Verwendung von ZBDDs anstatt von ROBDDs eine deutliche Verringerung der Knotenanzahl.

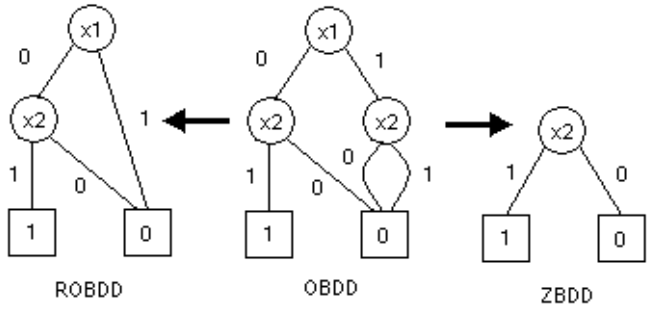


Abbildung 2.4: Erzeugung eines ROBDD und eines ZBDD aus einem OBDD

Vor diesem Hintergrund entwickelte Andreas Noack ein ZBDD-Paket (Siehe [Noa99]) in der Sprache C++, das im Modelchecker EEMC Verwendung findet.

Bei der Implementierung des ZBDD-Pakets wird das Konzept der Shared-BDDs umgesetzt. Alle ZBDDs werden in einer gemeinsamen Datenstruktur gehalten. Dies ermöglicht unter anderem eine Identitätsüberprüfung in konstanter Zeit. Alle Knoten werden in einem Feld *nodes* gespeichert. Ein ZBDD wird allein durch seinen Wurzelknoten repräsentiert. Jeder Knoten tritt im Knotenfeld nur ein einziges Mal auf. Deshalb ist es beim Einfügen eines neuen Knotens notwendig, zu prüfen, ob ein Knoten mit den gleichen Eigenschaften bereits vorhanden ist. Damit diese Überprüfung in konstanter Zeit erfolgen kann, wird jeder Knoten in einer Hashtabelle gemerkt. Ein Knoten im Knotenfeld ist durch fünf Attribute gekennzeichnet, seine Variablennummer, die Feldindizes seines Low- und seines High-Knotens, einer Markierung und einem *next*-Zeiger auf den nächsten Knoten. Mit Hilfe der Markierung kann überprüft werden, ob ein Knoten tatsächlich zu einem BDD gehört, also ob der Knoten von irgendeinem Wurzelknoten aus erreichbar ist. Der *next*-Zeiger ist erforderlich, um die freien Feldelemente des Knotenfeldes zu verketten und um die Knoten mit einem gleichen Hashwert in der Hashtabelle zu verknüpfen. Der Hashwert wird mit Hilfe der Variablennummer und den Nummern der Knotennachfolger bestimmt. Da das Knotenfeld eine feste Größe hat, ist es möglich, dass irgendwann keine Knoten mehr in das Feld eingefügt werden können. Im Knotenfeld können sich aber auch nicht mehr benötigte Knoten bereits zerstörter ZBDDs befinden. In einem weiteren Feld werden deshalb die Wurzelknoten aller aktuellen ZBDDs verwaltet. Mit Hilfe der Funktion *Mark* ist es möglich, alle Knoten eines so referenzierten ZBDD von der Wurzel aus zu markieren. So realisiert A. Noack eine Garbage-Collection, die im Falle eines vollen Knotenfeldes alle Knoten referenzierter BDDs markiert. Alle unmarkierten Knoten werden dadurch als überflüssig gekennzeichnet und können entfernt werden. A. Noack stellt in seinem ZBDD- Paket neben üblichen Mengenoperationen auch Operationen zum Schalten von Transitionen zur Verfügung, die zur Manipulation von Markierungsmengen von Petrinetzen benötigt werden [Noa99].

2.3 Visualisierung

BDDs, im konkreten Fall ZBDDs, stellen in EEMC die interne Repräsentation für Markierungsmengen von Petrinetzen dar. Informationen über ihre Struktur bleiben jedoch in der Datenstruktur verborgen. Deshalb ist es wünschenswert, dem Nutzer bei Bedarf diese Informationen in einer geeigneten Weise sichtbar zu machen.

Jeder OBDD, und damit auch jeder ZBBD, ist bezüglich der Anzahl und der Ordnung seiner Variablen eindeutig beschrieben. Zählt man in jeder Tiefe des BDD die Anzahl der unterschiedlichen Knoten, erhält man eine Liste von Zahlenwerten, die aufsummiert der Knotenanzahl des BDD entspricht. Stellt man jeden einzelnen Wert als Länge eines horizontalen Balkens dar und reiht diese Balken in der Reihenfolge der Variablen zentriert und untereinander auf, erhält man eine geeignete graphische Darstellung für einen BDD, die nützliche Informationen über dessen Struktur liefern kann. Ein solches Diagramm erinnert ein wenig an ein demographisches Diagramm zur Darstellung von Bevölkerungsentwicklungen.

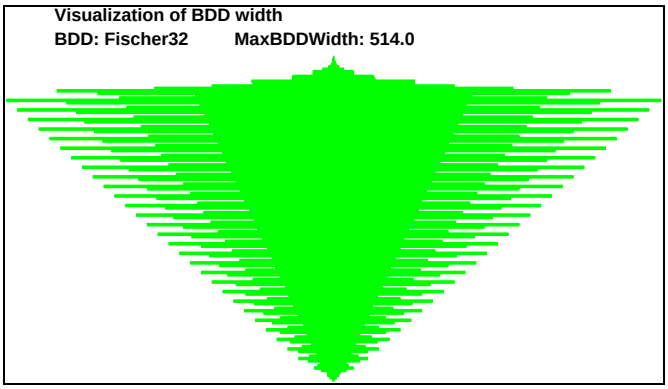


Abbildung 2.5: BDD-Visualisierung

Dirk Beyer und Michael Vogel entwickelten ein Java-Programm, das eine Zahlenliste in der oben beschriebenen Art und Weise visualisiert. Es ermöglicht auch, mehrere BDDs übereinander zu “legen” und dadurch zu vergleichen. Die Zahlenliste, die auch mehrere BDDs enthalten kann (durch “#” getrennt), wird aus einer Datei mit der Endung .dat gelesen.

Ein Ziel der Arbeit war es, das ZBDD-Paket um diese Visualisierung zu erweitern. Dazu wurde eine Funktion eingeführt, die an einem ZBDD eine solche Variablenzählung vornimmt und das in Form einer Liste von Knotenanzahlen vorliegende Ergebnis in eine Datei ausgibt, die dann von dem Java-Programm VisualizeSize gelesen werden kann. Dies leistet die Funktion *export2dat(char *filename)*, die als Parameter den gewünschten Dateinamen mit Pfadangabe erwartet. Mit VisualizeSize kann man sich einen BDD allerdings nur auf dem Bildschirm graphisch darstellen lassen. Deshalb wurde das Programm um eine Funktion erweitert, die den visualisierten BDD in eine PostScript Datei ausgibt.

Auch das ZBDD- Paket von A. Noack wurde um eine Funktion *export2PS(char *filename, char *name, bool ordered)* ergänzt, die die Ausgabe in eine PostScript Datei direkt vornimmt. Sie erwartet als Parameter den Dateinamen, einen Beschreibungsnamen für den jeweiligen BDD und eine Information über die Ordnung der Variablen.

Zunächst ist es allerdings erforderlich, die besagte Knotenzählung vorzunehmen. Dabei liegt es nahe, die Struktur, die einem binären Baum sehr ähnlich ist, auszunutzen und beispielsweise einen Inorderdurchlauf durchzuführen und dabei in einem Feld, dessen Größe der Variablenanzahl entspricht, für jede Variable die Anzahl der mit ihr beschrifteten Knoten zu zählen. Im Gegensatz zu einem Binärbaum enthält ein BDD Maschen. Deshalb wird man einzelne Knoten von mehreren anderen Knoten aus erreichen können und bei einem rekursiven Durchlauf entsprechend mehrmals besuchen. Dies führt dazu, dass man in diesem Falle einen Knoten mehrmals in die Zählung aufnehmen würde. Darüber hinaus nimmt eine solche Traversierung eben wegen der vielen Redundanz sehr viel Zeit in Anspruch. Im ZBDD-Paket ist eine Funktion *print* implementiert, die nach diesem Schema den BDD als Klammerausdruck auf die Konsole ausgibt. Für einen BDD mit sehr vielen Knoten beansprucht eine solche Ausgabe durchaus mehrere Stunden.

Daher dürfen bei der Knotenzählung Teile des BDD, die bereits komplett durchlaufen wurden, nicht mehr berücksichtigt werden und müssen daher auch nicht mehr besucht werden. Dies kann durch eine geeignete Datenstruktur wie einer Hashtabelle realisiert werden. Sobald von einem Knoten aus alle erreichbaren Knoten besucht wurden, wird er gezählt und vermerkt. Dank der implementierten Garbage-Collection erübrigt sich hier eine weitere Hashtabelle. Wie bereits angemerkt, besitzt jeder Feldknoten ein Attribut, das Auskunft über seine Markierung gibt. Da Knoten erst im Falle einer benötigten Garbage-Collection markiert werden, ist es unproblematisch, einen Knoten bei der Zählung zu markieren. Danach muss für jeden Knoten die Markierung allerdings wieder aufgehoben werden, damit er später nicht als zu einem BDD gehörig betrachtet wird, auch wenn er es zu diesem Zeitpunkt vielleicht nicht mehr ist. Das Zählen der Knoten übernimmt die Funktion *exportBDD_*. Also werden nur unmarkierte Knoten besucht, mitgezählt und dann ebenfalls markiert. Schließlich wird das Feld mit den Ergebnissen nur noch mit Hilfe einer formatierten Ausgabe in eine entsprechende Datei übertragen.

Die Ausgabe in eine PostScript Datei gestaltet sich ähnlich einfach. Im Zuge der Knotenzählung muss in diesem Fall auch das Vorkommen der Variablen bestimmt werden, deren Knoten am häufigsten im BDD vertreten sind. So wird das Maß für die Breite des BDD bestimmt. PostScript ist eine Seitenbeschreibungssprache, hinter der sich allerdings eine ausgewachsene Programmiersprache verbirgt. PostScript ist durchweg stackorientiert. Anweisungen werden also in Postfixnotation angegeben. Eine Operation, die zum Beispiel eine Linie von den aktuellen Koordinaten zum Punkt (x, y) beschreibt, findet als ihre Argumente die beiden obersten Stackelemente vor. Zur Visualisierung des BDD anhand des Zahlenfeldes genügen entsprechend zwei Anweisungen zum Zeichnen einer Linie und zum Versetzen der aktuellen

Position. “100 100 moveto” legt beispielsweise zweimal den Wert 100 auf den Stack und bezieht die Operation moveto auf diese beiden Werte, um die aktuellen Koordinaten dorthin zu verschieben.

Die BDD-Visualisierung ist auf das A4 Format zugeschnitten. Die PostScript Anweisungen werden mit Hilfe der formatierten Ausgabe in eine entsprechende Datei geschrieben. Zunächst werden Kommentare (Version, Bounding Box usw.) in der Datei vermerkt. Anschließend werden Informationen über den Namen des BDD, seine maximale Breite, die Anzahl seiner Knoten, die Anzahl der Variablen und ein Kommentar bzgl. ihrer Ordnung geschrieben. Dann werden in einer Iteration über die Variablenanzahl die Anfangs- und Endkoordinaten jeder Linie berechnet und untereinander in die Postscript Datei übertragen, die das Vorkommen der jeweiligen Variablen repräsentieren. Das A4 Format lässt eine maximale Breite von 596 Pixel zu. Da ein BDD aber theoretisch eine größere Breite haben kann, muss anhand der maximalen Breite des BDD der Wert bestimmt werden, der multipliziert mit der Knotenanzahl einer konkreten Variablen die Länge des zu zeichnenden Balkens ergibt. Die Breite einer Linie ist mit dem Wert „zwei“ vorbestimmt.

In der vertikalen Richtung stehen 843 Pixel zur Verfügung. Die oberen 93 Pixel werden für die Ausgabe der BDD-Informationen benötigt. Für die graphische Darstellung stehen in der Vertikalen entsprechend 750 Pixel zur Verfügung. Sollte eine BDD demnach mehr als 375 Variablen besitzen, muss auch die Dicke der Balken angepasst werden.

Das Programm EEMC bietet nun die Möglichkeit, einige der beim Modelchecking entstehenden BDDs wahlweise in eine PostScript Datei, eine dat-Datei oder beides zu exportieren. Dazu muss beim Programmaufruf die Option -P, -D oder entsprechend -B und ein Dateiname mit Pfad angegeben werden. Wählt man als Modelchecking-Verfahren die Symbolische Tiefensuche, wird im Falle eines negativen Ergebnisses der Inhalt des Kellers visualisiert. Im Falle des BwdCycleCheck und des FwdCycleCheck werden die Visualisierungsdaten der Markierungsmengen der ermittelten Fixpunkte exportiert. Beim BwdCycleCheck wird zusätzlich die Menge aller erreichbaren Markierungen visualisiert.

Die Anzahl der Knoten eines BDD hängt sehr stark von der Ordnung der Variablen ab. Um zu sehen, wie sich eine fehlende Umordnung der Variablen auf die Größe des BDD auswirkt, kann mit der Option -p diese unterbunden werden. Beispielsweise steigt die Knotenanzahl des BDD aller erreichbaren Markierungen bzgl. des Petrinetzes closed_system5 und der LTL-Formel
„*! (G ((arm1_pick_up_angle && arm1_pick_up_ext && arm1_magnet_on) -> (arm1_magnet_on U (arm1_release_angle && arm1_release_ext))))*“
durch unterlassen der Variablenordnung von 4778 auf 28750. Die maximale Breite des BDD wächst auf 1984 an. Siehe Abbildung 2.6 und 2.7.

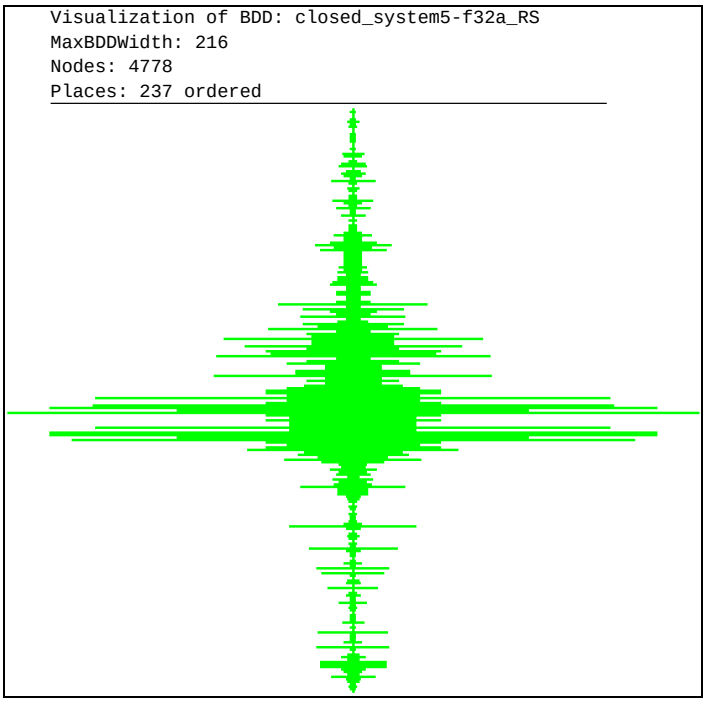


Abbildung 2.6: Mit Variablenordnung

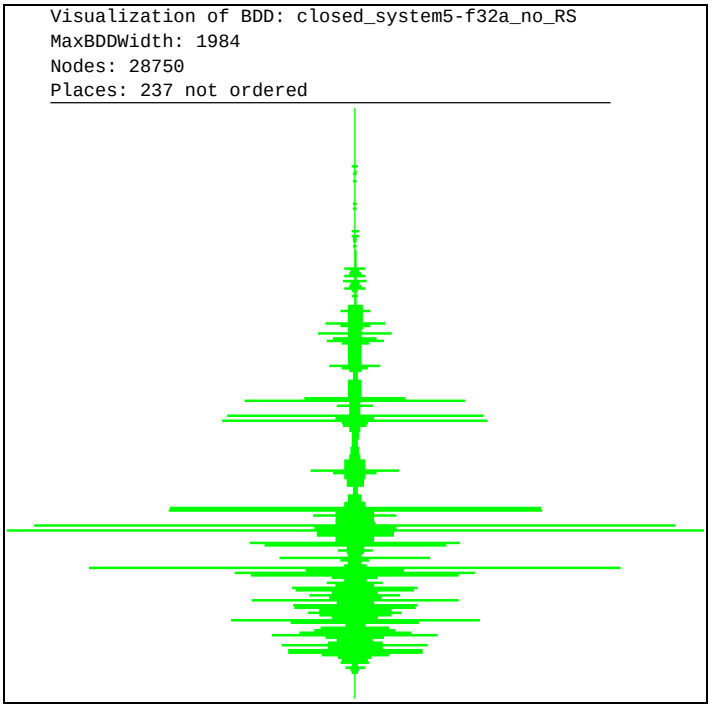


Abbildung 2.7: Ohne Variablenordnung

Eine geeignete Variablenordnung wird in EEMC mit Hilfe eines heuristischen Verfahrens von Andreas Noack bestimmt ([Noa99], S. 13; [Spr01], S. 65). Weitere Beispiele für die BDD-Visualisierung sind im Anhang zu finden.

3 Modelchecking mit Petrinetzen

3.1 Petrinetze

Automaten sind ein geeigneter Formalismus zur Beschreibung und Modellierung sequentieller Systeme. Ein entsprechendes Pendant bezüglich nebenläufiger Systeme stellen die von Carl Adam Petri eingeführten und nach ihm benannten Petrinetze dar. Eine formale Einführung in die Thematik der Petrinetze findet sich in der Arbeit von J. Spranger [Spr01] in Kapitel 2.

Hinter dem Begriff Petrinetz verbirgt sich ein gerichteter Graph, bestehend aus zwei disjunkten Mengen von Knoten und einer Menge von Kanten. Eine Kante verbindet niemals zwei Knoten der gleichen Menge. In solch einem Graphen gibt es also zwei unterschiedliche Knotenarten, die als Stellen bzw. Plätze und Transitionen bezeichnet werden. Eine Stelle wird graphisch durch einen Kreis, eine Transition durch ein Quadrat repräsentiert. Die Kanten lassen sich je nach Art des Netzes in eine Vielzahl von Klassen mit unterschiedlichen Eigenschaften unterteilen. Eine Stelle des Netzes kann Marken enthalten, die als schwarze Punkte dargestellt werden. Wenn in einem Petrinetz eine Stelle höchstens eine Marke enthalten kann, spricht man von einem sicheren Netz. Es haben sich seit ihrer Einführung 1962 zahlreiche unterschiedliche Klassen von Petrinetzen entwickelt. Hier soll es jedoch nur um sichere Petrinetze gehen.

Die Belegung aller Stellen eines Netzes wird als Markierung m bezeichnet und repräsentiert einen Zustand des durch das Petrinetz modellierten Systems.
Jedes Petrinetz besitzt eine Anfangsmarkierung m_0. Ein sicheres Petrinetz mit n Stellen kann demnach eine theoretische Anzahl von 2^n unterschiedlichen Markierungen besitzen. Der Übergang von einem Systemzustand zu einem anderen bzw. von einer Markierung zu einer anderen Markierung erfolgt über das Schalten oder auch Feuern einer Transition. Eine Transition kann also die Belegung der Stellen eines Netzes verändern. Jede eingehende Kante einer Transition entspringt stets einer Stelle, jede ausgehende Kante mündet in eine Stelle. Für Stellen gilt das Analoge. Die Menge der Stellen mit ausgehenden Kanten zu einer Transition t bildet den Vorbereich von t, die Menge der Stellen mit von t aus eingehenden Kanten bildet den Nachbereich von t. Die Gesamtheit aller mit einer Transition verbundenen Stellen nennt man ihre Umgebung. Damit eine Transition die Belegung des Netzes verändern, also schalten kann, muss sie aktiviert sein. Sie ist aktiviert oder hat Konzession, wenn ihr gesamter Vorbereich markiert ist. Beim Schalten entfernt eine Transition alle Marken aus ihrem Vorbereich und markiert ihren gesamten Nachbereich. Somit darf in einem sicheren Petrinetz in einer Markierung, die eine Transition aktiviert, keine Nachstelle der Transition markiert sein.

Alle von der Anfangsmarkierung m_0 durch Schalten von Transitionen erreichbaren Markierungen bilden den so genannten Erreichbarkeitsgraphen des Petrinetzes. Der Erreichbarkeitsgraph eines Petrinetzes spiegelt dessen Verhalten und somit das des modellierten Systems wider. Es handelt sich dabei um einen gerichteten Graphen, dessen Knoten den erreichbaren Zuständen (Markierungen) und dessen Kanten den für die jeweiligen Zustandsübergänge verantwortlichen Transitionen entsprechen. Aus ihm lassen sich viele Informationen über die Eigenschaften des Systems, die es beim Modelchecking zu überprüfen gilt, gewinnen. Mit seiner Hilfe lässt sich zum Beispiel die Frage nach toten Markierungen oder der Reversibilität eines Petrinetzes beantworten. Eine tote Markierung (Deadlock) erlaubt es keiner Transition zu schalten. Der zugeordnete Knoten im Erreichbarkeitsgraphen besitzt also keine ausgehenden Kanten. Ein Petrinetz wird als reversibel bezeichnet, wenn es von jeder Markierung aus einen Pfad zur Anfangsmarkierung gibt.

Der Erreichbarkeitsgraph lässt sich jedoch auch auf andere Weise interpretieren. Lässt man seine Kanten unbeschriftet, gehen die Informationen über das Schalten der einzelnen Transitionen bei bestimmten Markierungen verloren. Es entsteht ein so genanntes Transitionssystem, das sämtliche Systeminformationen durch die in den Knoten codierten Systemzustände repräsentiert. Ein gerichteter Graph mit einer Menge von Zuständen, einer Menge unbeschrifteter Zustandsübergänge und einem ausgezeichneten Anfangszustand wird als Transitionssystem bezeichnet. Ein Pfad oder Weg in einem solchen Graphen ist eine im Anfangszustand beginnende Sequenz aufeinanderfolgender Zustände. Man kann die Zustandsmenge des Transitionssystems T als Alphabet Σ auffassen. Dann bildet die Menge aller Pfade durch T eine Sprache über Σ. Im konkreten Fall sind die das Alphabet bildenden Zustände die erreichbaren Markierungen eines sicheren Petrinetzes. Dadurch bildet jedes Petrinetz eine Sprache über seine erreichbaren Markierungen. Damit ergibt sich ein auf Automaten basierender Ansatz zur Beschreibung durch Petrinetze repräsentierter Systeme.

3.2 Temporale Logik

Die bereits angesprochenen sicheren Petrinetze eignen sich hervorragend zur Modellierung einer speziellen Klasse von Systemen, die durch eine kontinuierliche Interaktion mit ihrer Umgebung gekennzeichnet sind. Sie werden als reaktive Systeme bezeichnet und terminieren in der Regel nicht. Ein nebenläufiges Kommunikationsprotokoll ist ein Vertreter dieser Systemklasse. Besitzt ein solches System tote Zustände, lässt dies in der Regel auf einen schwerwiegenden Systemfehler schließen. Nun stellt sich die Frage, wie sich die Eigenschaften reaktiver Systeme geeignet formulieren und überprüfen lassen. Der Erreichbarkeitsgraph eines Petrinetzes lässt sich bequem als ein Transitionssystem interpretieren und definiert somit die Sprache des betrachteten Systems. Durch die Eigenschaft reaktiver Systeme, in der Regel nicht zu terminieren, handelt es sich bei der besagten Sprache um eine Menge unendlicher Worte. Ein Wort der Systemsprache repräsentiert dann das kontinuierliche Verhalten des Systems.

Jeder Zustand, also jeder Knoten des Transitionssystems, wird als ein konkreter Zeitpunkt aufgefasst. Im Falle einer toten Markierung würde die Zeit aber entsprechend stehen bleiben. Dies ist natürlich unerwünscht und wird durch das Ergänzen einer Kante, die vom betroffenen Knoten zu ihm zurückführt, behoben. Dann kann man uneingeschränkt von einer unendlichen Systemsprache sprechen.

Der Faktor Zeit spielt demnach eine bedeutende Rolle bei der Beschreibung reaktiver Systeme. Ob ein System irgendwann in einen gewünschten Zustand oder niemals in einen unerwünschten Zustand gerät, ergibt sich dann als Kernfrage bei der Verifizierung von Systemen.

Liegt ein sicheres Petrinetzmodell zugrunde, ist ein Zustand durch die Markiertheit der einzelnen Stellen charakterisiert. In diesem Fall kann man die Belegung einer Stelle bzgl. einer bestimmten Markierung als eine elementare Aussage auffassen, die genau dann wahr ist, wenn die Stelle im jeweiligen Zustand markiert ist. Vor diesem Hintergrund lässt sich die bekannte Aussagenlogik dazu benutzen, Eigenschaften einzelner Zustände eines Systems zu beschreiben. Dies genügt allerdings noch nicht, um das zeitliche Systemverhalten auszudrücken. Durch Hinzunahme fünf temporallogischer Operatoren (X, G, F, U und R) wird dies möglich.
Die Bedeutung dieser Operatoren, bezogen auf eine aussagenlogische Formel, kann wie folgt beschrieben werden:

$X\varphi$	im nächsten Zustand gilt φ;
$G\varphi$	in allen folgenden Zuständen gilt φ;
$F\varphi$	es wird irgendwann ein Zustand erreicht, in dem φ gilt;
$\varphi U\psi$	es wird irgendwann ein Zustand erreicht, in dem ψ gilt, bis dahin gilt φ;
$\varphi R\psi$	wird ein Zustand erreicht in dem ψ nicht gilt, wird bis dahin ein Zustand durchlaufen, in dem φ gilt.

Die dadurch entstehende Logik wird temporale Logik genannt.
Temporallogische Formeln lassen sich über einer Menge von Aussagen induktiv formulieren.

1. true und false sind temporallogische Formeln;
2. Ist φ eine Aussage, sind φ und $\neg\varphi$ temporallogische Formeln;
3. Sind φ und ψ temporallogische Formeln, so auch $\varphi \wedge \psi$ und $\varphi \vee \psi$;
4. Sind φ und ψ temporallogische Formeln, so auch $F\varphi$, $X\varphi$,$G\varphi$, $\varphi U\psi$ und $\varphi R\psi$.

Nun lässt sich überprüfen, ob ein Pfad in einem Transitionssystem eine temporallogische Formel erfüllt oder nicht. Die Relation “ein Pfad p erfüllt die Formel φ” wird folgendermaßen ausgedrückt: $p \models \varphi$.

Die Menge aller möglichen unendlichen Pfade lässt sich als Baum darstellen, dessen Wurzel genau dem Initialzustand entspricht. Eine gegebene Formel kann in solch einem Baum nun von einer Menge von Pfaden erfüllt sein. Möglicherweise erfüllen sogar alle Pfade, unter Umständen erfüllt aber auch gar kein Pfad die Formel. Durch die Angabe so genannter Pfadquantoren lässt sich eine temporallogische Formel auch diesbezüglich genauer formulieren. In Anlehnung an die bekannten Quantoren $\exists$ und $\forall$, wurden die Pfadquantoren E (es gibt mindestens einen Pfad) und A (auf allen Pfaden) eingeführt. Dann sind auch $A\psi$ und $E\psi$ temporallogische Formeln, insofern ψ eine ist.

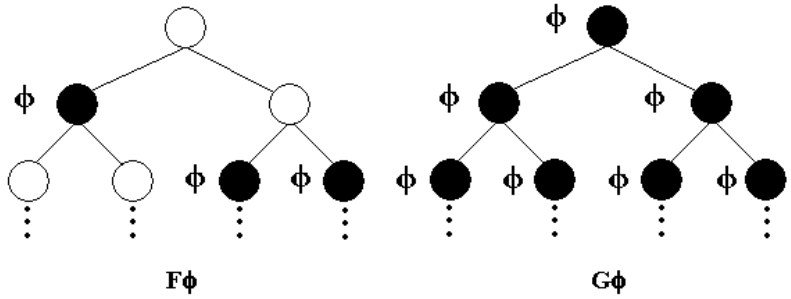


Abbildung 3.1: temporallogische Formeln mit erfüllendem Transitionssystem

Je nach Verwendung dieser Pfadquantoren werden mehrere temporale Logiken unterschieden, deren wichtigste Vertreter CTL (computation tree logic) und LTL (linear temporal logic) sind. Die beiden oberen Darstellungen sind ein Beispiel für LTL-Formeln [CGP01]. Sie bestehen nur aus elementaren Aussagen und temporallogischen Operanden. In einer LTL-Formel kommen keine Pfadquantoren vor. Wenn ein System eine solche Formel erfüllen soll, müssen alle beim Startzustand beginnenden Pfade durch das zugehörige Transitionssystem dieser Formel genügen. Sollte dies nicht der Fall sein, gibt es mindestens einen Pfad, der die Formel nicht erfüllt.

Die Gültigkeitsrelation lässt sich also auf das gesamte Transitionssystem T erweitern und mittels einer Formel φ durch T |= φ ausdrücken. Die Sprache von T erfüllt die Formel φ. Jede LTL-Formel definiert eine Sprache unendlicher Worte, die sie erfüllen.

Sowohl Transitionssysteme, die aus dem Erreichbarkeitsgraphen sicherer Petrinetze gewonnen wurden, als auch LTL-Formeln lassen sich durch Sprachen charakterisieren. Sie gehören zur Klasse der ω-regulären Sprachen. Ihnen gemeinsam ist die Eigenschaft, dass ihre Worte die gleiche Struktur, genauer einen endlichen Wortanfang und ein sich unendlich oft wiederholendes Restwort, besitzen. Wie auch die regulären Sprachen, lassen sich ω-reguläre Sprachen mit Hilfe von Automaten repräsentieren. Jedoch ist zu bemerken, dass ein unendliches Wort anders von einem Automaten akzeptiert wird als ein endliches.

So genannte Büchiautomaten unterscheiden sich exakt in dieser Besonderheit von üblichen nichtdeterministischen Automaten. Auch die Zustandsmenge eines Büchiautomaten ist endlich. Ein unendliches Wort muss folglich eine Teilmenge der Automatenzustände unendlich oft durchlaufen. Der Automat akzeptiert ein unendliches Wort dann, wenn diese Teilmenge unendlich oft durchlaufener Zustände einen nicht leeren Schnitt mit den akzeptierenden Zuständen hat. Mit anderen Worten wird mindestens ein akzeptierender Zustand unendlich oft durchlaufen ([Spr01], S.85).

3.3 Modelchecking

Modelchecking stellt ein Verfahren dar, das die Frage beantworten soll, ob ein System bestimmte Eigenschaften erfüllt. Vor dem Hintergrund, dass mit Petrinetzen modellierte Systeme und LTL-Formeln Sprachen bilden, die sich durch Büchiautomaten repräsentieren lassen, wird das Modelchecking zu einer Problemstellung, die sich mit Hilfe von Automaten lösen lässt.
Eine LTL-Formel generiert eine Sprache unendlicher Worte, die die Formel erfüllen. Ein durch ein Petrinetz modelliertes System erzeugt ebenfalls eine Sprache, beschrieben durch die Menge aller unterschiedlichen Zustandsfolgen.

Wenn alle Zustandsfolgen eine LTL-Formel erfüllen, genügt also das System der entsprechenden Formel. Dann ist die Sprache des Systems eine Teilmenge aller Worte, die die LTL-Formel erfüllen, also ihrer Sprache. Beim Modelchecking wird jedoch das Problem der Teilmengenbeziehung in ein Problem des leeren Schnitts zweier Sprachen umgewandelt. Wenn eine Menge A Teilmenge einer Menge B ist, so ist der Schnitt von A und dem Komplement von B stets leer. Dies sind zwei Problemstellungen, die sich bezüglich regulärer und ω-regulärer Sprachen mit Hilfe von Automaten leicht lösen lassen.
Seien A1 und A2 Automaten, die zwei Sprachen repräsentieren, so wird der Schnitt dieser Sprachen durch den Produktautomaten A der beiden Automaten repräsentiert. Bei der Bildung von Produktbüchiautomaten sind im Vergleich zu gewöhnlichen finiten Automaten einige Besonderheiten zu beachten
([Spr01], S. 88-90).

Entscheidend ist also die Frage nach der Leerheit der durch den Automaten A repräsentierten Sprache. Sie ist leer, wenn es keinen unendlichen Lauf durch den Automaten gibt, auf dem ein akzeptierender Zustand liegt. Es wurde bereits erwähnt, das sich die Menge aller Pfade bzgl. eines Transitionssystems als Baumstruktur darstellen lässt. Hinsichtlich der Zustandsfolgen durch den entsprechenden Baum des Produktautomaten kann diese Frage nach einem Kreis mit mindestens einem akzeptierenden Zustand durch eine einfache Tiefensuche beantwortet werden. Das Lösen dieses Problems lässt sich mit linearem Aufwand bzgl. der Graphengröße bewältigen. Im Falle eines akzeptierenden Kreises stellt der Inhalt des Kellers, der bei der Tiefensuche verwendet wird, ein Gegenbeispiel dar, also eine Zustandsfolge, die die LTL-Formel nicht erfüllt.

Der automatentheoretische Ansatz des Modelcheckings lässt sich wieder auf Petrinetze zurückführen. So genannte Büchinetze sind sichere Petrinetze, deren Erreichbarkeitsgraph einem Büchiautomaten entspricht. Jedes sichere Petrinetz ist ein Büchinetz, wenn man eine Teilmenge seiner Stellen als akzeptierende Stellen auszeichnet. Wenn der Erreichbarkeitsgraph eines sicheren Netzes einen Büchiautomaten repräsentiert, so muss auch für das Büchinetz ein vergleichbares Akzeptierungsverhalten definiert werden. Demnach sind alle Markierungsfolgen des Netzes akzeptierend, wenn sie eine Markierung unendlich oft enthalten, bei der eine akzeptierende Stelle markiert ist.

Auch der zu einer LTL-Formel gehörige Büchiautomat lässt sich in ein Büchinetz transformieren. Dazu wird für jeden Zustand des Automaten eine Stelle und für jede Regel der Regelmenge eine Transition eingeführt. Die akzeptierenden Zustände des Automaten werden dabei durch akzeptierende Stellen repräsentiert. Ein in einer Regel d definierter Übergang von einem Zustand q1 zu einem Zustand q2 wird im Büchinetz durch zwei Kanten realisiert, die die entsprechenden Stellen q1 und q2 über die Transition d verbinden. Wenn die Anfangsmarkierung des Büchinetzes nur die dem Startzustand entsprechende Stelle q0 markiert, dann spiegelt das Netz den Formelbüchiautomaten wider. Da allerdings negierte Aussagen in der LTL-Formel berücksichtigt werden müssen, werden sowohl dem Systembüchinetz als auch dem Formelbüchinetz zusätzliche Stellen hinzugefügt. Dies ist erforderlich, da das Schalten von Transitionen nur von der Markiertheit einer Stelle abhängig gemacht werden kann. Wenn eine Stelle also laut Formel unmarkiert sein soll, so wird eine weitere Stelle eingefügt, die genau dann markiert ist, wenn ihr Pendant unmarkiert ist. Solche Stellen werden als Komplementärstellen bezeichnet und bilden mit ihrer zugehörigen Stelle eine Invariante. Bei jeder Markierung des Netzes ist stets nur einer der beiden Stellen markiert.

Es lässt sich also sowohl ein Systemnetz als auch eine LTL-Formelbüchiautomat als Büchinetz darstellen. Wenn es nun gelingt, aus beiden Netzen ein einziges Netz zu entwickeln, dessen Erreichbarkeitsgraph der Produktbildung beider zugehöriger Automaten entspricht, kann das Modelchecking direkt auf Ebene eines Petrinetzes bewerkstelligt werden.
Auch dieses Produktbüchinetz ist mit Hilfe der beiden Büchinetze konstruierbar. Realisiert wird die Transformation in ein einziges Büchinetz durch eine Kopplung beider Netze. Bei deren alternierendem Schalten kann das Formelbüchinetz praktisch kontrollieren, ob das Systembüchinetz akzeptierende Markierungen erreicht. Zur Realisierung eines alternierenden Schaltens werden zwei weitere Stellen, scPN und scBA, benötigt, die entscheiden, welches Netz wann schalten darf. Auch sie bilden eine Invariante. Dann werden die Transitionen des Systembüchinetzes mit einer eingehenden Kante von scPN und einer ausgehenden Kante zur Stelle scBA verknüpft. Für die Transitionen des Formelbüchinetzes wird genauso verfahren, jedoch mit entgegengesetzter Kantenrichtung. Die Transitionen eines Teilbüchinetzes können demnach nur dann schalten, wenn ihre zugehörige Kontrollstelle markiert ist. Da das Formelbüchinetz akzeptierende Schaltvorgänge des Systembüchinetzes erkennen soll, sind weitere Kanten erforderlich. Im Formelbüchiautomaten ist ein Zustandsübergang von der Belegung bestimmter Stellen des Systemnetzes abhängig. Diese Abhängigkeit wird auf das Produktbüchinetz übertragen, indem zwischen den betroffenen Stellen des Systembüchinetzes und den die Zustandsübergängen repräsentierenden Transitionen des Formelbüchinetzes bidirektionale Kanten, so genannte Lesekanten, eingefügt werden. Die Anfangsmarkierung des Produktbüchinetzes setzt sich aus den Anfangsmarkierungen der Teilbüchinetze und der Kontrollstelle scBA zusammen. Die Menge der akzeptierenden Stellen des Produktbüchinetzes entspricht der des Formelbüchinetzes.

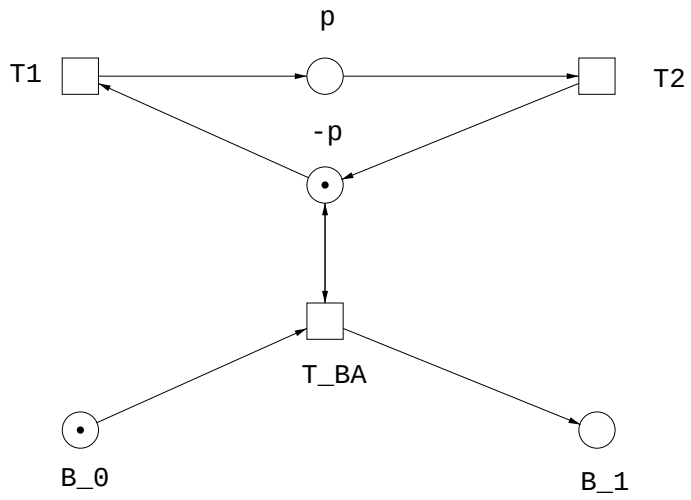


Abbildung 3.2: komplementäre Stelle mit Kopplung zum Formelbüchinetz

3.3.1 Die Symbolische Tiefensuche

Auf der Grundlage des vorliegenden Produktbüchinetzes lässt sich nun ein sehr effizientes Modelchecking-Verfahren anwenden, dass den Erreichbarkeitsgraphen schrittweise nur soweit erzeugt, wie es nötig ist, um das Leerheitsproblem der Sprache des Netzes zu entscheiden. Es wird deshalb auch als ein on-the-fly Verfahren bezeichnet.

Ähnlich einem automatenbasierten Verfahren wird auch hier mit Hilfe einer Tiefensuche nach einem akzeptierenden Kreis gesucht. Jedoch wird dabei ein Graph zugrunde gelegt, dessen Knoten ganzen Markierungsmengen entsprechen. Eine Markierung eines Petrinetzes mit n Stellen kann als n-stellige boolesche Funktion mit dem Ergebnis 1 aufgefasst werden. Zur Repräsentation der Mengen von Markierungen können dann ZBDDs verwendet werden.
Um den benötigten Graphen zu konstruieren, muss zunächst eine Übergangsrelation zwischen den einzelnen Markierungsmengen definiert werden. Man gewinnt die Nachfolgemenge einer Markierungsmenge durch das Schalten aller schaltfähigen Transitionen beider Teilbüchinetze, wobei zuerst alle Transitionen des Formelbüchinetzes schalten müssen. Da jeder Zustand einer Menge einen Nachfolgezustand haben muss, sind alle toten Zustände der Zustandsmenge in ihre Nachfolgermengen zu übertragen. So ergibt sich die Funktion *next* ([Spr01], S. 113).

Beginnend mit der Anfangsmarkierung kann nun schrittweise der Erreichbarkeitsgraph des Büchinetzes erzeugt und dabei nach einem akzeptierenden Kreis gesucht werden. Dazu werden alle erzeugten Markierungsmengen in einem Stapel gemerkt. Wird irgendwann eine Markierungsmenge erzeugt, die bereits im Stapel vorhanden ist, muss geprüft werden, ob eine der dazwischen liegenden Mengen ausschließlich aus akzeptierenden Markierungen besteht. Existiert eine solche Menge, wurde ein

akzeptierender Kreis gefunden. Dabei wurde ein Systemablauf ermittelt, der die zu überprüfende LTL-Formel nicht erfüllt, da das Formelbüchinetz aus der negierten LTL-Formel gewonnen wurde.

Taucht in einer Folge von Markierungsmengen eine Markierung mehrmals auf, muss dies leider nicht unbedingt bedeuten, dass sie auf einem Kreis im Erreichbarkeitsgraphen liegt ([Spr01], S.119 Beispiel 14).
Deshalb ist es notwendig, entstehende Markierungsmengen so zu zerlegen, dass eine Menge des Graphen entweder nur aus akzeptierenden oder nur aus nicht akzeptierenden Markierungen besteht. Unter der Vorraussetzung, dass die Menge akzeptierender Zustände F bestimmt worden ist, lässt sich nun jede durch die Funktion *next* ermittelte Zustandsmenge M in zwei disjunkte Teilmengen zerlegen, indem zum Einen die Schnittmenge von M und F und zum Anderen die Differenzmenge von M und F bestimmt werden. Diese Teilmengen werden dann als Nachfolgemengen von M in den Graphen eingefügt. Ihre Vereinigung entspricht wieder der Menge M.

Wären alle Markierungsmengen des Graphen disjunkt, würde eine Partition der Menge der Markierungen des Erreichbarkeitsgraphen des Büchinetzes vorliegen. Die maximal mögliche Größe des Graphen würde dann der Anzahl der erreichbaren Markierungen entsprechen.
Die einfache Zerlegung von Nachfolgemengen in akzeptierende und nicht akzeptierende Mengen, verhindert allerdings noch nicht, dass sich unterschiedliche Mengen des Graphen zum Teil aus gleichen Markierungen zusammensetzen. Man könnte eine Partitionierung zwar erreichen, müsste dafür beim Einfügen eines Knoten in den Graphen unter Umständen vorhandene Mengen zerlegen.

Damit aber auf dem entstehenden Graphen eine Tiefensuche durchführbar ist, muss das Zerlegen bereits existierender Mengen verhindert werden.
Deshalb werden auch Mengen in den Graphen aufgenommen, die entweder selbst Teilmenge einer vorhandenen Zustandsmenge sind, oder bei denen der umgekehrte Fall vorliegt. Ein nach diesem Vorgehen konstruierter Graph wird als Mengengraph bezeichnet. Die Zerlegung einer in den Graphen einzufügenden Zustandsmenge unter Berücksichtigung von Teilmengenbeziehungen findet sich in der Funktion *divide* wieder ([Spr01], S. 124).

3.3.2 Implementierung in EEMC

Die Klasse PetriNet stellt zur Durchführung der symbolischen Tiefensuche die beiden Funktionen *SymbolicDFS* und die Funktion *visit* bereit. Zur Repräsentation eines Knotens des Mengengraphen wurde die Struktur UElem eingeführt. Sie enthält die vier Attribute *num*, *root* , *acc* und *set*, also die Nummer des Knotens, eine Angabe, ob es sich um einen akzeptierenden Knoten handelt, eine Angabe, ob der Knoten einen Kreis begründet und die Zustandsmenge des Knotens in Form eines ZBDD. Des Weiteren wurden zwei Felder vom Typ *UElem* vereinbart, die jeweils akzeptierende und nicht akzeptierende Knoten des Graphen aufnehmen. Mit Hilfe der Funktionen *checkAccUnivers* und *checkNoAccUnivers*

kann für eine Zustandsmenge überprüft werden, ob bereits ein Knoten existiert, der sie enthält. Ist dies der Fall, wird dessen Adresse geliefert, ansonsten wird das nächste freie Feldelement mit der gegeben Zustandsmenge versehen.
Auch der für die Tiefensuche benötigte Stapel *Stack* wird als Array implementiert. Er enthält Zeiger auf Knoten des Graphen. Die Variable *Stack_count* gibt den Index des obersten Stapelelementes an.

Nach dem Aufruf der Funktion *SymbolicDFS* werden zunächst der Initialzustand, die Menge der toten und akzeptierenden Zustände und deren Schnittmenge bestimmt. Anschließend wird das erste Feldelement der nicht akzeptierenden Knoten initialisiert. Dessen Zustandsmenge besteht nur aus dem Initialzustand. Durch den Aufruf der Funktion *visit* mit dem Startknoten als Argument wird die eigentliche Tiefensuche begonnen.

Die Funktion *visit*, deren Argument ein Knoten des Graphen ist, initialisiert zunächst das *num*-Attribut mit dem Wert eines mitlaufenden Zählers und legt den Knoten auf den Stapel. Des Weiteren werden zwei lokale Variablen *min* und *tmp_min* deklariert, wobei auch *min* mit dem aktuellen Zählerwert initialisiert wird. Beide werden zur Erkennung von Kreisen benötigt.
Anschließend wird die Menge aller Folgezustände der aktuellen Zustandsmenge ermittelt [*next*]. Bei der Berechnung der Folgezustände schalten zuerst alle Transitionen des Formelbüchinetzes. Anschließend schalten die Transitionen des Systemnetzes. Von der so entstehenden Markierungsmenge ausgehend werden dann alle erreichbaren Markierungen bestimmt ([Spr01], S. 57) und in ihre akzeptierenden und nicht akzeptierenden Teilmengen zerlegt.

Da das Ziel im Auffinden akzeptierender Kreise besteht, wird im Anschluss die Menge der akzeptierenden Zustände näher betrachtet. Für die weitere Zerlegung dieser Teilmengen [*divide*] wird ein lokal deklariertes Array *tmpList* benötigt, in dem die Zeiger auf die während der Zerlegung entstandenen oder eventuell bereits vorhandenen Knoten gespeichert werden. Der BDD *tmpBddAcc* enthält zunächst die Menge der akzeptierenden Zustände. Nun wird für jeden akzeptierenden Knoten im Feld *Acc_Univers* überprüft, ob dessen Zustandsmenge mit der Menge *tmpBddAcc* übereinstimmt, insofern diese nicht leer ist. Ist dies der Fall, wird die Adresse des jeweiligen Knotens in *tmpList* vermerkt. Falls keine Übereinstimmung vorliegt, werden *tmpBddAcc* und die Zustandsmenge auf gegenseitige Teilmengenbeziehungen hin untersucht. Dazu wird die Durchschnittsmenge gebildet und geprüft, ob eine der beiden Zustandsmengen genau dieser Durchschnittsmenge entspricht.

Ist die Zustandsmenge eines bereits vorhandenen Knotens Teilmenge der neuen Zustandsmenge, wird die Adresse des Knoten in *tmpList* vermerkt. Wenn keine Menge in der Anderen enthalten der Schnitt aber auch nicht leer ist, wird ein neuer Knoten mit der Menge *tmpBddAcc* erzeugt. In beiden Fällen werden anschließend aus der Menge *tmpBddAcc* alle Zustände entfernt, die sich auch in der Menge des Knotens aus *Acc_Univers* befanden. Enthält *tmpBddAcc*, nachdem alle vorhandenen Knoten überprüft wurden, noch Zustände, so wird für die verbleibende Menge ein neuer Knoten erzeugt.

Bevor sich die Menge der nicht akzeptierenden Zustände schließlich der gleichen Prozedur unterziehen muss, wird jedoch die Liste der durch die Zerlegung ermittelten Knoten untersucht. Alle Knoten, deren *num*-Wert 0 ist, waren zuvor im Feld *Acc_Univers* nicht vertreten. Also wird für sie die Funktion *visit* rekursiv aufgerufen. Die Funktion *visit* liefert einen Wert zurück, der dem kleinsten *num*-Wert aller in einem Funktionsdurchlauf überprüften Knoten des Graphen entspricht. Bei der Untersuchung derjenigen Knoten, auf die die Einträge in *tmpList* verweisen, wird die Variable *tmp_min* im Falle eines neuen Knotens mit dem Ergebnis der Funktion *visit* belegt, andernfalls mit dem *num*-Wert des bereits früher erzeugten Knotens. Sollte der Wert von *tmp_min* dann kleiner als der von *min* ausfallen, wird *min* mit dem Wert von *tmp_min* aktualisiert und das *root*-Attribut des aktuellen Knotens auf true gesetzt. In diesem Falle wurde ein zu einem früheren Zeitpunkt aufgenommener Knoten gefunden, dessen Zustandsmenge einer Teilmenge des aktuellen Knotens vor der Zerlegung entspricht. Der Wert der Variablen *min* entspricht also der kleinsten Knotennummer aller bei der Zustandsmengenzerlegung, inklusive der rekursiven Aufrufe der Funktion *visit*, betrachteten Knoten. Anschließend wird mit der Menge der nicht akzeptierenden Zustände analog verfahren.

Danach kann überprüft werden, ob tatsächlich ein akzeptierender Kreis vorliegt. Dies ist der Fall, wenn nach der Zerlegung und der Aktualisierung des Wertes der Variablen *min* dieser nach wie vor mit dem *num*-Wert des aktuellen Knotens übereinstimmt, wenn also durch den rekursiven Aufruf von *visit* für neu erzeugte Knoten ein Knoten entsteht, der die Zustandsmenge des aktuellen Knotens enthält.
Wenn der aktuelle Knoten also einen Kreis begründet, sind alle Knoten des Kreises über ihn gegenseitig erreichbar. Sie bilden eine SZK, eine stark zusammenhängende Komponente.

Am wichtigsten ist nun die Beantwortung der Frage nach Knoten des Kreises mit akzeptierenden Zustandsmengen. Deshalb wird die SZK schrittweise vom Stapel entfernt und dabei jeder Knoten dahingehend überprüft, ob er akzeptierende Zustände repräsentiert. Wird ein solcher Knoten gefunden, handelt es sich also um einen akzeptierenden Kreis. Im negativen Fall werden alle Knoten des Kreises vom Stapel entfernt, und die Funktion liefert als Ergebnis den aktuellen Wert von *min*. Die Tiefensuche ist beendet, wenn keine neuen Knoten mehr entstehen.

Bezüglich der Bestimmung der Folgemarkierungen ist zu bemerken, dass bei der Ermittlung aller erreichbaren Markierungen einige Systemtransitionen schalten können, weil sie mit den Kontrollplätzen nicht verbunden sind. Im Produktbüchinetz ist normalerweise jede Transition des Netzes mit den Kontrollplätzen gekoppelt. Dann können die Transitionen des System- und Formelnetzes tatsächlich nur abwechselnd schalten.

In der konkreten Implementierung werden allerdings nur die Transitionen des Systembüchinetzes mit den Kontrollstellen verbunden, deren Schalten die Belegung der für das Formelbüchinetz relevanten Stellen verändern können. Die übrigen Transitionen können entsprechend unabhängig von der Belegung der Kontrollstellen schalten.

3.3.3 DFS am Philosophenbeispiel

Im Folgenden wird die Symbolische Tiefensuche an einem kleinen Beispielnetz durchgeführt. Das Petrinetz in Abbildung 3.3 modelliert das berühmte Philosophenproblem.

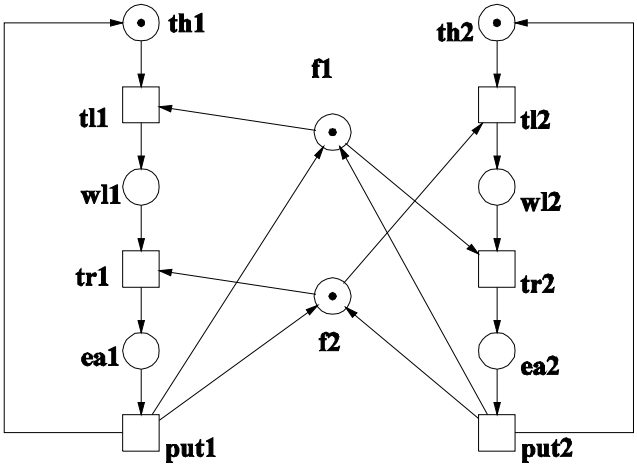
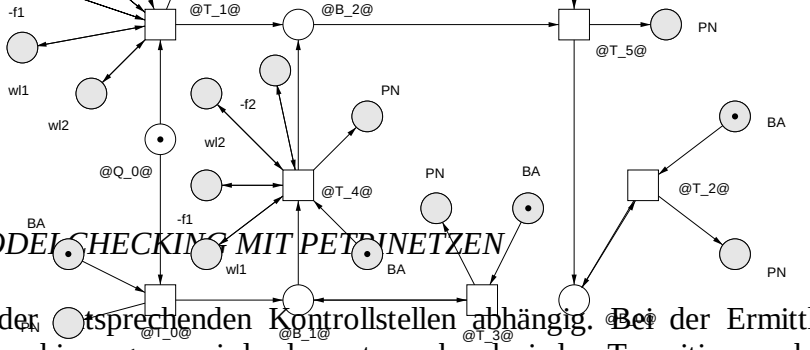


Abbildung 3.3: Petrinetz für das Philosophenproblem

Es soll geprüft werden, ob das Netz folgende LTL-Formel erfüllt.
G (! (wl1 * wl2 * (!f1) * (!f2)))
Das Netz erfüllt diese Formel, wenn für alle möglichen Markierungsfolgen gilt, dass es keine Markierung gibt, bei der die Stellen wl1 und wl2 markiert und f1 und f2 nicht markiert sind.
Ein kurzer Blick auf das Netz verrät jedoch, dass dies nicht der Fall ist.
Schalten beispielsweise die Transitionen tl1 und tl2 direkt hintereinander, wird eine entsprechende Markierung erreicht. Darüber hinaus handelt es sich um eine tote Markierung, die ein weiteres Schalten von Transitionen unmöglich macht.

Um diese Erkenntnis per Modelchecking zu erlangen, wird erst die LTL-Formel negiert. Es gilt:
Ga = ! F ! a , also ! G a = !! F ! a = F ! a $\Rightarrow$
! G (! (wl1 * wl2 * (!f1) * (!f2))) = F (wl1 * wl2 * (!f1) * (!f2)))

Aus dieser negierten LTL-Formel wird zunächst der Formelbüchiautomat konstruiert ([Spr01], S. 90). In dieser Formel sind nur elementare Aussagen negiert. Deshalb befindet sie sich bereits in Negations-Normalform. Wäre diese nicht der Fall, müsste sie vor der Konstruktion des Formelbüchiautomaten entsprechend umgewandelt werden. Anschließend werden wie beschrieben aus dem Formelbüchiautomaten ein Formelbüchinetz und letztendlich das Produktbüchinetz entwickelt.
In diesem Petrinetz ist jede Transition mit einer für die Formel bedeutsamen Stelle verbunden. Somit ist im Produktbüchinetz das Schalten jeder Transition von der



Belegung der entsprechenden Kontrollstellen abhängig. Bei der Ermittlung der Nachfolgemarkierungen wird dementsprechend jede Transitionen höchstens einmal schalten können.

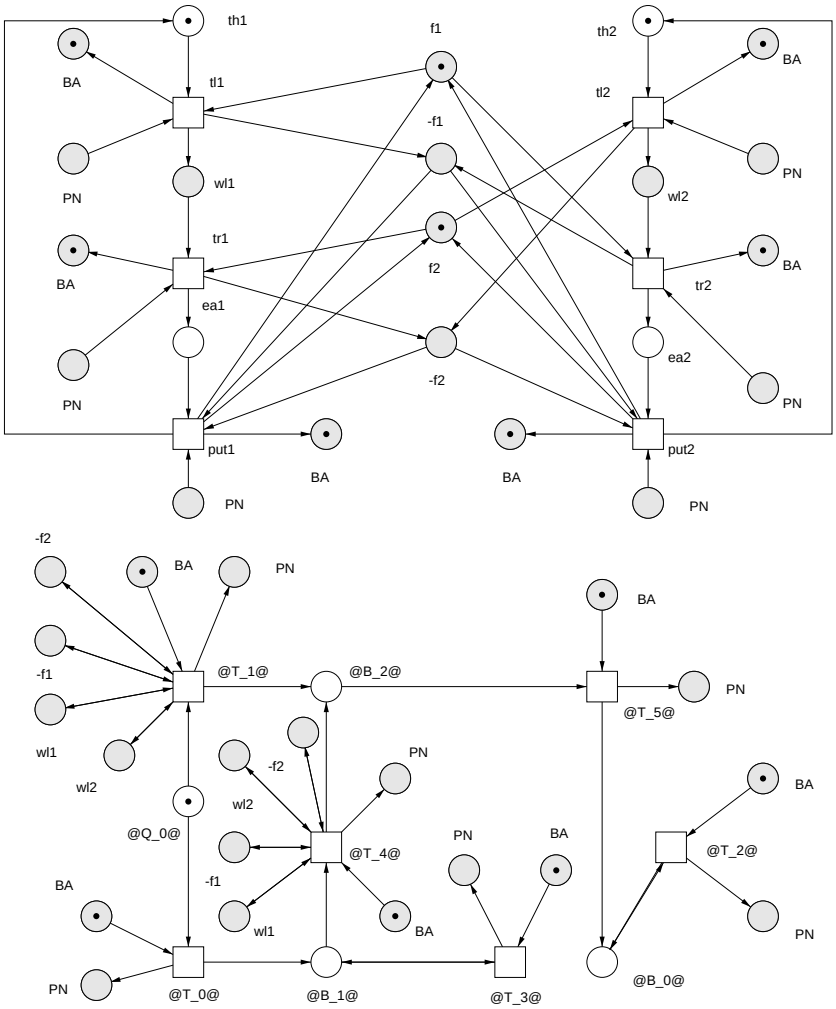


Abbildung 3.4: Produktbüchinetz

Bei der Symbolischen Tiefensuche wird nun der Erreichbarkeitsgraph schrittweise konstruiert.

Die nächsten Zeilen zeigen einen Programmdurchlauf für das obige Beispiel, ergänzt durch einige zusätzliche Ausgaben.

DSSZLTL Version 3.0
Formula:
*(G (! (wl1 * wl2 * (! f1) * (! f2))))*
(NrP 14 NrT 12 NrMP 6)

*_1_from_0_*
1_number of states: 1
1_fire of formula net:
@T_0@
1_fire of system net:
tl2 ; tl1
1_number of accepting nodes: 0
1_number of not accepting nodes: 1
*_2_from_1_*
2_number of states: 2
2_fire of formula net:
@T_3@
2_fire of system net:
tl2 ; tl1 ; tr2 ; tr1
2_number of accepting nodes: 0
2_number of not accepting nodes: 1
*_3_from_2_*
3_number of states: 3
3_fire of formula net:
@T_3@ ; @T_4@
3_fire of system net:
put2 ; put1
3_number of accepting nodes: 1
*_4_from_3_*
4_number of states: 1
4_fire of formula net:
@T_5@
4_fire of system net:
4_number of accepting nodes: 1
*_5_from_4_*
5_number of states: 1
5_fire of formula net:
@T_2@
5_fire of system net:
5_number of accepting nodes: 1
5_number of not accepting nodes: 0
5_cycle found: 5
5_min: 5
4_number of not accepting nodes: 0
3_number of not accepting nodes: 1
*_6_from_3_*
6_number of states: 1
6_fire of formula net:
@T_3@
6_fire of system net:
6_number of accepting nodes: 0
6_number of not accepting nodes: 1

---

counter-example:
<tl2 ; tl1>
SymbolicDFS: The Formula is false
SymbolicDFS computation time: real:0.00 sec, user:0.00 sec, sys:0.00 sec
Total time: real:0.11 sec, user:0.04 sec, sys:0.07 sec
ZBDD statistic:
overall number of nodes: 228
number of nodes in nodes-array: 228
number of garbage collections: 0
fireUnion cache hits/miss/sum: 35 186 221
binOp cache hits/miss/sum: 163 494 657
state number cache hits/miss/sum: 5 34 39

Die folgende Abbildung zeigt den Erreichbarkeitsgraphen des Produktbüchinetzes, der bei der symbolischen Tiefensuche erzeugt wurde. Die roten Umrandungen stellen die Markierungsmengen dar, die sich in den Knoten des Mengengraphen befinden.

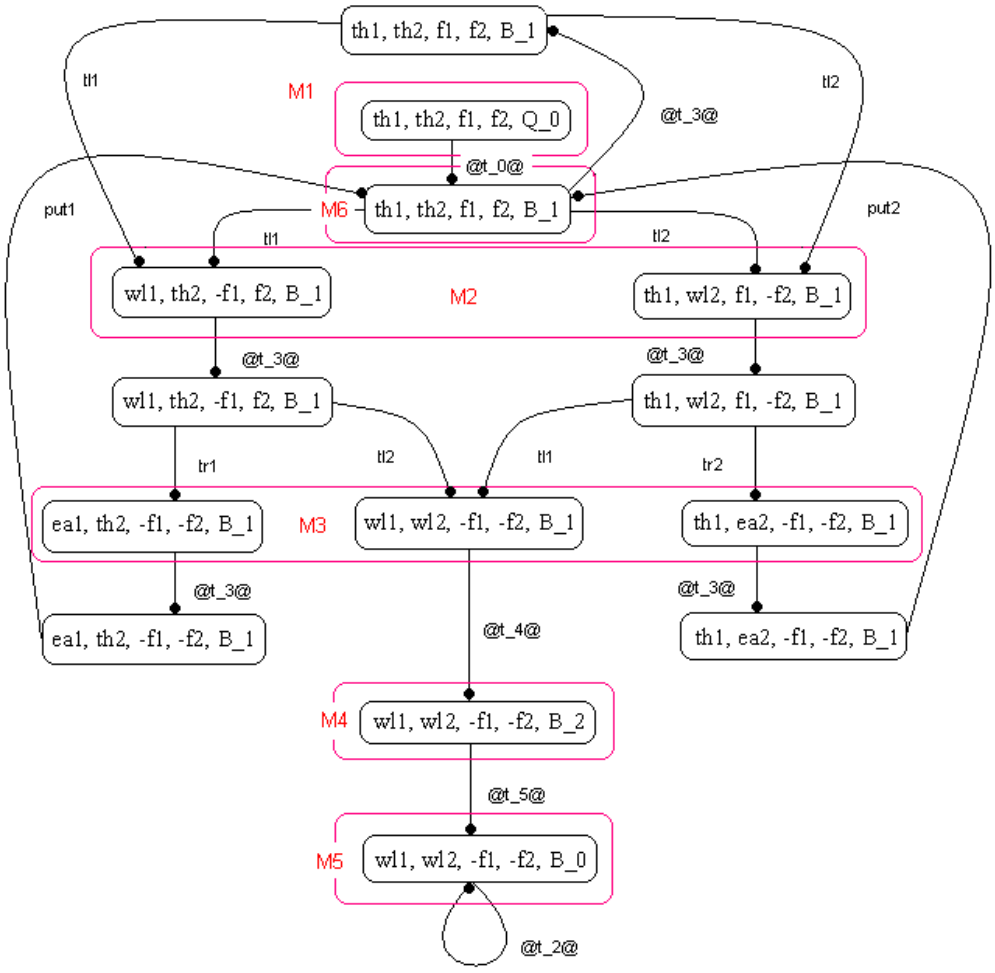


Abbildung 3.5: Erreichbarkeitsgraph des Produktbüchinetzes

3.3.4 Ermittlung eines Gegenbeispiels

Sollte bei der symbolischen Tiefensuche ein akzeptierender Kreis gefunden werden, erfüllt das System die LTL-Formel nicht. Es muss also mindestens eine Folge von Zuständen geben, die nicht in der Sprache der Formel liegt. Nun kann es zur Behebung dieses vermutlichen Designfehlers äußerst hilfreich sein, diese Zustandsfolge zu kennen. Es ist allerdings günstiger, Informationen über die geschalteten Transitionen zu besitzen, anstatt über einzelne Markierungen (Zustände).

Würden beim Modelchecking nur einzelne Zustände betrachtet werden, befände sich im Falle eines gefundenen Kreises das Gegenbeispiel direkt auf dem Stapel. Man müsste aus der Markierungsfolge nur noch diejenigen Transitionen bestimmten, die für die einzelnen Übergänge verantwortlich waren.

Im Falle der symbolischen Tiefensuche von Jochen Spranger ist dies leider nicht auf diesem Wege möglich. Auf dem Stapel befinden sich nicht nur einzelne Markierungen, sondern ganze Mengen. Darüber hinaus besteht der Übergang von zwei auf dem Stapel befindlichen, benachbarten Markierungsmengen nicht aus einer Menge von Transitionen, die jeweils nur einmal geschaltet haben. Bei der Bestimmung aller erreichbaren Markierungen entstehen einzelne Markierungen durch das Hintereinanderschalten mehrer verschiedener Transitionen.

Um also eine Transitionsfolge als Gegenbeispiel angeben zu können, müssen Transitionsfolgen gefunden werden, die eine einzelne Markierung einer Menge des Stapels in eine Markierung ihrer Folgemenge auf dem Stapel überführen. Reiht man diese Transitionsfolgen zu einer gültigen Folge zusammen, die zu einer Markierung der letzten auf dem Kreis liegenden Markierungsmenge führt, hat man eine bzgl. der LTL-Formel ungültige Schaltfolge gefunden. Die zugehörige Markierungsfolge muss in der entsprechenden Reihenfolge aus jeder Stapelmenge eine konkrete Markierung enthalten.

Realisierung

Zur Bestimmung eines Pfades von der Anfangsmarkierung hin zu einer Markierung einer akzeptierenden Markierungsmenge auf dem Kreis wird folgendermaßen verfahren.

Zunächst wird wie bei einer Tiefensuche ein Pfad bestimmt, der bei der Anfangsmarkierung beginnt und zu einer Markierung der nächsten Menge auf dem Stapel führt. Die Abbruchbedingung bei dieser rekursiven Suche besteht entweder im Finden einer entsprechenden Markierung oder einer Übertretung der maximalen Pfadlänge, die während der eigentlichen Tiefensuche bestimmt werden muss. Wurde eine Markierung gefunden, lässt man die Transitionen des Formelnetzes schalten. Sollte die dadurch entstandene Markierung in der Zielmenge liegen, war die Suche erfolgreich. Andernfalls wird nun nach einer Folge von Transitionen des Systemnetzes gesucht, die in eine Markierung der wiederum nächsten Stapelmenge führt.

Um die Anzahl der Transitionen einzuschränken, die in einer Rekursionsebene schalten können, wird bei der eigentlichen Tiefensuche festgehalten, welche Transitionen geschaltet haben. Dafür wird ein Zähler benötigt, der die Anzahl der

Schaltebenen repräsentiert. Weiterhin wird zu jeder Markierungsmenge des Stapels die Länge der längsten Schaltsequenz bei der Bestimmung der erreichbaren Markierungen vermerkt. Sie wird als oben erwähntes Abbruchkriterium benötigt.

Die drei Felder *f3*, *f4* und *f5* beinhalten die Informationen über die zu schaltenden Transitionen. Das erste Feld enthält die Nummern der Transitionen, ist aber ohne den Inhalt der beiden übrigen Felder nutzlos. Das Feld *f4* beherbergt im i–ten Feldelement die Anzahl der in der i-ten Schaltebene geschalteten Transitionen. Das Feld *f5* enthält im i-ten Feldelement den Index bzgl. des Feldes *f3* der ersten geschalteten Transition der i-ten Schaltebene. Die Struktur *UElem* wurde um die Attribute *offset* und *fired* erweitert; *fired* ist die erwähnte Länge der längste Schaltfolge und *offset* gibt den Wert der 0-ten Schaltebene bei jedem Aufruf von *visit* an.

Mit diesen Informationen und den Markierungsmengen auf dem Stapel lässt sich nun durch die Funktionen *search1* und *search2* der gesuchte Pfad bestimmen. Bei der Erzeugung des Pfades wird jede Nummer einer geschalteten Transition auf dem dafür eingerichteten Stapel *TStack* abgelegt. Sollte ein Pfad verworfen werden, werden die Transitionen vom Stapel entfernt.

```
func search1 ( Cur, Next , S , num) ≡
    f:= -1;
    newStates := Ø;
    forall t∈ FBN do
        newStates := newStates ∪ fire( S , t);
    od;
    if newStates ∩ M_cycle.set ≠ Ø then //M_cycel entspricht der Zielmenge
        return 1;
    fi
    if Cur.fired = 0 then
        h :=search1 ( Next, Stack[num+2], newStates, num + 1);
        if h = 1 then
            return h;
        fi
        return -1;
    fi
    h := search2 ( Cur, Next , newStates, 0 , num);
    if h = -2 then
        h := search1(Next, Stack[num +2], newStates, num +1);
    fi
    return h;
end search1;
```

```
func search2 (Cur, Next, S , count, num) ≡
      h := -1;
      cnt :=0;
        if count > Cur.fired then
            return h;
      fi
      forall i< f4[Cur.offset+count] do
            t := f3[f5[Cur.offset+count]+i];
            afterFire := fire ( S , t );
            if afterFire ≠ Ø then
                cnt++;
                TStack := TStack ∪ t ;
                if afterFire ∩ Next.set ≠ Ø then
                        h := search1( Next, Stack[num+2], afterFire, num +1);
                else    h: = search2(Cur, Next , afterFire, count +1, num);
                        fi
                if h < 0 then
                        TStack := TStack / t;
                else return 1;
                fi
      od;
      if cnt =0 then
            return -2;
      fi
      return h;
end search2;
```

Nach Beendigung der DFS wird die Funktion search1 aufgerufen. Sie erhält als Argumente die ersten beiden Knoten des Stapels, die Anfangsmarkierung und den Wert 0. Der ermittelte Pfad befindet sich anschließend auf dem Stapel *TStack*. Da während der DFS der zugehörige Stapel wieder abgebaut wird, ist es erforderlich, dessen Inhalt beim Auffinden eines gesuchten Kreises zu sichern.

Dieser Ansatz stellt jedoch keine befriedigende Lösung des Problems dar. Für das Beispiel aus 3.3.3 lässt sich zwar ein Gegenbeispiel ermitteln, dies ist aber nur aufgrund der geringen Größe und der Struktur des Petrinetzes möglich.
Im konstruierten Produktbüchinetz sind alle Transitionen des Systembüchinetzes mit den Kontrollstellen verbunden. Bei der Ermittlung der erreichbaren Markierungen während der Bestimmung der Nachfolgemarkierungen, kann keine Transition des Systembüchinetzes schalten. Daher beträgt die maximale Pfadlänge zwischen den Markierungsmengen auf dem Stapel 1.

Bei anderen Beispielen wie pusher oder closed_system, können sich Werte bis zu einer Größenordnung von 100 ergeben. In Verbindung mit einer Anzahl von Transitionen jenseits der 200 ergibt sich eine derart große Vielzahl von zu überprüfenden Schaltfolgen, so dass die Ermittlung eines Gegenbeispiels auf diese

Weise als nicht realisierbar angesehen werden muss. Darüber hinaus benötigen die zusätzlichen Felder zum Festhalten der geschalteten Transitionen unnötig viel Speicher. Die Veränderung bei der Ermittlung der erreichbaren Markierungen zur Bestimmung der maximalen Pfadlänge und zur Identifizierung der geschalteten Transitionen, durch die Verwendung der Funktion *fire* anstatt von *fireUnio* verlangsamt die eigentliche Tiefensuche entscheidend ([Spr01], S. 60-61). Mit Hilfe des Stapelinhaltes lässt sich demnach kein Gegenbeispiel erzeugen, zumindest nicht in vertretbarer Zeit.

4 Inhibitorkanten

Inhibitorkanten sind spezielle Kanten, die nur von einer Stelle zu einer Transition führen. Die Besonderheit einer solchen Kante liegt darin, dass die betroffene Transition nur dann schalten kann, wenn die mit ihr verbundene Stelle nicht markiert ist. Auf diese Weise lässt sich die Unmarkiertheit einer Stelle sehr einfach ausdrücken. Zunehmende Bedeutung gewinnen Inhibitorkanten in der Bio-Informatik. Mit ihrer Hilfe lassen sich Reaktionen modellieren, deren Stattfinden nur bei Nicht- Vorhandensein bestimmter Stoffe möglich ist.

4.1 Veränderungen an EEMC

EEMC ist bisher in der Lage, Petrinetze mit zweierlei Arten von Kanten zu verarbeiten.
Neben der Nutzung einfacher gerichteter Kanten besteht die Möglichkeit, so genannte Lesekanten zu verwenden, die wie Inhibitorkanten von einer Stelle zu einer Transition führen. Ihre Besonderheit besteht darin, dass die zu ihr gehörende Stelle beim Schalten der Transition markiert bleibt.
Im Folgenden soll kurz skizziert werden, wie ein Petrinetz innerhalb des Programms EEMC repräsentiert wird.
Das Programm besteht aus den drei Komponenten LTL2BA, BAPN2BN und BNEEMC.

Die Namen der einzelnen Teilprogramme lassen schon auf ihre Funktionalität schließen. Die Komponente LTL2BA liest eine LTL-Formel aus einer Datei, konstruiert daraus einen entsprechenden Büchiautomaten und stellt eine Funktion zur Verfügung, um den Automaten ausgeben zu können. Diese wird zusammen mit der das Petrinetz beschreibenden APNN-Datei vom Teilprogramm BAPN2BN eingelesen. Aus dem Büchiautomaten und dem Systemnetz wird dann das für das Modelchecking benötigte Produktbüchinetz entwickelt. Wird die Ausgabe des Netzes in eine Datei vorgenommen, kann es schließlich von BNEEMC eingelesen werden. Das Teilprogramm BNEEMC ist der eigentliche Modelchecker und bietet dem Nutzer drei verschiedene Verfahren zur Prüfung der Systemeigenschaften hinsichtlich der spezifizierten LTL-Formel.
Innerhalb von BAPN2BN und BNEEMC, beide Teilprogramme sind in der Programmiersprache C++ implementiert, wird demnach eine interne

Darstellungsform für Petrinetze benötigt. Die dafür nötigen Datenstrukturen und Funktionen befinden sich in den Dateien net.h und net.cc.

Zur Repräsentation der unterschiedlichen Knoten eines Petrinetzes wurden zunächst die Strukturen *PElem_* und *TElem_* eingeführt. Sie enthalten neben Attributen zur eindeutigen Identifizierung eine Reihe von Zeigern auf andere Netzelemente, so dass sich die einzelnen Elemente zu einer Netzstruktur verketten lassen. Sowohl die Transitionen als auch die Stellen bilden jeweils eine einfach verkettete Liste. Darüber hinaus besitzt jede Transition eine Liste ihrer Vorstellen und eine Liste ihrer Nachstellen. Für eine Stelle gilt das Analoge. Diese Listen bestehen jedoch nicht direkt aus Zeigern auf Stellen bzw. Transitionen, sondern aus Zeigern auf die entsprechenden Kanten. Zur Darstellung einer Kante wurde die Struktur *EdgeElem_* eingeführt. Sie enthält neben einer Angabe des Kantentyps jeweils eine Instanz des Platzes und der Transition, die sie verbindet. Außerdem besitzt jede Kante vier Zeiger auf andere Kanten. Sie repräsentieren die jeweils benachbarten Kanten. Eine Kante, die beispielsweise von einer Stelle zu einer Transition führt, enthält einen Zeiger auf eine Kante (somit auf eine Kantenliste), die zu einer Nachstelle der Transition führt und entsprechend mit *nextPostPlaces* benannt ist.

Darüber hinaus werden eine Hashtabelle für die Stellen des Netzes und eine Hashtabelle für die Gesamtheit der Netzknoten geführt, um ausschließen zu können, dass zwei identische Stellen bzw. Knoten mehr als einmal auftreten.
Da die Netze zunächst aus einer Datei eingelesen werden müssen, stellen beide Programme Funktionen zur Verfügung, um den Inhalt der entsprechenden Dateien zu parsen. Bei diesem Vorgang werden mit Hilfe der Funktionen *mk_place*, *mk_trans* und *mk_arc* die jeweiligen Netzelemente erzeugt und in die interne Netzdatenstruktur eingegliedert. Da in Zukunft auch Inhibitorkanten verarbeitet werden sollen, wird im Folgenden die Funktion *mk_arc* kurz betrachtet. Im Detail gibt es zwischen der Funktion innerhalb von BAPN2BN Unterschiede zu der von BNEEMC. Das grundlegende Vorgehen der beiden Funktionen ist aber identisch. Die Funktion erhält als wichtigste Argumente den Namen der Kante, ihre Quelle und ihr Ziel, ihre Kapazität und eine Angabe über ihren Typ. Da es sich um sichere Petrinetze handelt, sind Kapazitäten ungleich dem Wert „Eins“ nicht zulässig.

Zunächst wird überprüft, ob der angegebene Kantentyp überhaupt erlaubt ist und ob die durch die Kante zu verbindenden Netzknoten bereits existieren. Anschließend wird festgestellt, ob die Kante von einer Stelle zu einer Transition führt oder ob der umgekehrte Sachverhalt vorliegt. Nur für den ersten Fall sind beispielsweise Lesekanten zulässig. In der internen Struktur des Netzes, darf zwischen einer Stelle und einer Transition nur eine Kante existieren. Sollten also zwei einfachen Kanten eine Transition und eine Stelle verbinden, müssen sie durch eine Lesekante ersetzt werden. Die entsprechende Stelle wird mit Hilfe dieser Kante später als eine Vor- und Nachstelle identifiziert. Jede Stelle, die in der Umgebung einer Transition liegt, muss hinsichtlich dieser Transition genau klassifiziert werden. Diese Klassifizierung, die zunächst nur Vorstellen, Nachstellen und Vor- und Nachstellen unterscheidet, ist unter anderem bei der Realisierung des Schaltens einer Transition unverzichtbar.

Bevor also die betrachtete, von einer Stelle ausgehende Kante dem Netz hinzugefügt wird, werden alle ausgehenden Kanten der Transition durchmustert und dahingehend untersucht, ob sie zu der betreffenden Stelle zurückführen. Ist dies der Fall, wird die neue Kante dem Netz als Lesekante durch entsprechende Zeigermanipulation hinzugefügt, und im gleichen Zuge die überflüssigen Kanten entfernt. Eine als solche ausgewiesene Lesekante wird direkt in das Netz integriert.

Führt eine Kante von einer Transition zu einer Stelle, so darf es sich nur um eine einfache Kante handeln. Dennoch müssen auch in diesem Fall alle Kanten zu den Vorstellen der Transition überprüft und im gegebenen Fall durch Lesekanten ersetzt werden, was das Einfügen der aktuellen Kante unterbinden würde.
Die Funktionen mk_place und mk_trans sind an dieser Stelle nicht von Interesse.
Diese zwei Funktionen genügen allerdings nur dem Teilprogramm BNEEMC, um die gesamte Netzstruktur zu konstruieren, da es diese, bis auf die angesprochene Kantenreduzierung, nicht weiter verändert.

Im Falle von BAPN2BN werden zwei weitere Funktionen benötigt. Die Funktion BAPN2BN konstruiert aus einem vorliegenden Systempetrinetz und einem Büchiautomaten in Form eines Büchinetzes ein Produktbüchinetz. Dazu ist es nötig, durch zusätzliche Stellen und Kanten beide Netze zu koppeln. Demnach werden zwei Funktionen zum Einfügen komplementärer Stellen und der Kontrollstellen des Produktbüchinetzes inklusive der Erzeugung der erforderlichen Transitionen benötigt.

Die Markierungen des Erreichbarkeitsgraphen eines Petrinetzes im Falle unseres Modelchecking Problems eines Produktbüchinetzes werden mit Hilfe von ZBDDs repräsentiert. So ist es auch möglich, ganze Mengen von Markierungen mit einem ZBDD darzustellen. Ebenso lässt sich der Übergang von einer Markierung bzw. Markierungsmenge in eine andere durch das Schalten einer Transition direkt auf der Ebene eines ZBDD und mit Hilfe dafür geeigneter Operation realisieren. Das eingangs kurz vorgestellte ZBDD-Paket von Andreas Noack liefert dafür die drei Funktionen *fire_*, *revFire_* und *fireUnio_*. Soll der Modelchecker auch Inhibitorkanten verarbeiten können, ist es selbstverständlich notwendig, das Schaltverhalten einer Transition mit solch einer eingehenden Kante dahingehend anzupassen. Ein Transition wird durch eine Instanz der Struktur *Transition* repräsentiert. Sie verfügt über eine Identifikationsnummer, Angaben über die Anzahl ihrer Vorstellen, ihrer Nachstellen und deren Summe und über ein Feld *places*, das ihre Umgebungsstellen enthält.

Im Folgenden soll die Funktion *fire_* exemplarisch erläutert werden. Sie wird erstmalig von der Funktion *fire* aufgerufen. Diese erhält als Argumente eine Referenz des zu schaltenden BDD, einen Zeiger auf das *places* Feld der zu schaltenden Transition und die Nummer der Transition. Sie bestimmt mit Hilfe der Funktion *fire_* rekursiv einen BDD, der die Menge aller Markierungen nach dem Schalten der Transition enthält. Die Funktion *fire_* erzeugt dabei in jedem Rekursionsschritt einen neuen Knoten und fügt ihn mit Hilfe der Funktion *insert* in das Knotenfeld *nodes* ein, sofern er noch nicht darin enthalten ist.

Die Funktion *fire_* erhält als Argumente die Nummer des Wurzelknoten des ZBDD, in tieferen Rekursionsebenen den Wurzelknoten des entsprechenden Teil-BDD und die Liste der Vor– und der Nachstellen der aktuell schaltenden Transition. Im Folgenden werden drei Fälle unterschieden. Sollte die Variable des aktuellen Wurzelknotens kleiner sein als die Variable des ersten Listenelements, wird überprüft, ob der gesuchte BDD-Knoten nicht schon in einem eigens dafür vorgesehenen Cache aufzufinden ist. Ist dies nicht der Fall, wird ein neuer Knoten gebildet, dessen Low- und High-Nachfolgeknoten rekursiv durch Anwendung der Operation *fire_* bestimmt werden. Das Ergebnis wird im Cache vermerkt. Es wird ein Knoten erzeugt, dessen Variablenbeschriftung unverändert bleibt, dessen Low– und High–Kanten allerdings auf die Knoten zeigen, die das Ergebnis der *fire_*-Operation, angewandt auf die Nachfolgeknoten des aktuellen Wurzelknotens, sind.

Sollte die Variable des aktuellen Knoten allerdings mit dem Wert des ersten Listenelements übereinstimmen, so ist ein Knoten gefunden worden, der in der Umgebung der zu schaltenden Transition liegt und dessen Belegung sich entsprechend verändern kann. Um dies zu entscheiden, muss überprüft werden, ob der Knoten eine Vorstelle, eine Nachstelle oder beides der aktuellen Transition ist.

Vorstelle

In diesem Fall würde ein Knoten erzeugt werden, dessen Low–Knoten das Resultat der *fire_* Operation bzgl. des High–Knotens des aktuellen Knotens und des nächstfolgenden Listenelementes wäre. Die High–Kante des neuen Knotens würde auf den 0-Knoten zeigen. Ein BDD-Knoten repräsentiert stets eine Entscheidung, die über ihre ausgehenden Kanten bestimmt wird. Zur Bestimmung der Folgeknoten nach dem Schalten wird also für einen Knoten, der eine Vorstelle repräsentiert, derjenige Knoten herangezogen, auf den die High-Kante des aktuellen Knoten zeigt. Dies bedeutet eine Belegung dieser Stelle. Nach einem erfolgreichen Schalten der Transition würden alle Knoten mit der entsprechenden Variablenbeschriftung mit ihrer High-Kante auf den 0- Knoten zeigen, was einer Nichtbelegung der entsprechenden Stelle gleichkommt. Da es sich allerdings um einen Zero-surpressed BDD handelt, wird an dieser Stelle von der entsprechenden Reduktionsregel Gebrauch gemacht. Die High–Kante des neuen Knotens würde auf den 0–Knoten zeigen und kann demnach eliminiert werden. Anstatt also einen neuen Knoten zu erzeugen, wird direkt das Ergebnis der *fire_* Operation bzgl. des High–Knotens zurückgegeben. Kann eine Transition nicht schalten, weil mindestens eine Vorstelle nicht belegt gewesen ist, ergibt sich als Ergebnis der ZBDD, der nur aus dem 0-Knoten besteht.

Vor– und Nachstelle

Ein solcher Knoten repräsentiert eine Stelle, die mit der aktuellen Transition durch eine Lesekante verbunden ist. Der Unterschied zum oberen Fall besteht allein darin, dass hier wirklich ein neuer Knoten mit der entsprechenden Variablen erzeugt werden muss, da auch nach dem Schalten die entsprechende Stelle belegt ist. Also wird im Gegensatz zu einer Vorstelle der Transition ein Knoten erzeugt, dessen Low-Kante auf den 0–Knoten zeigt und dessen High–Kante auf einen

Knoten zeigt, der das Ergebnis der Operation *fire_* mit den gleichen Argumenten wie bei einer Vorstelle ist. Zur Berechnung des Folgeknotens nach dem Schalten einer Transition wird also auch hier der High–Nachfolgeknoten herangezogen, was eine Belegung der entsprechenden Stelle im Netz bedeutet.

Nachstelle
Eine Nachstelle einer schaltfähigen Transition ist stets unbelegt. Es wird hier davon ausgegangen, dass jede Vor– und Nachstelle einer Transition durch eine entsprechende Lesekante mit ihr verbunden ist. Nach dem Schalten der Transition muss die entsprechende Nachstelle belegt sein. Handelt es sich also um einen Knoten, dessen Variable eine Nachstelle der Transition repräsentiert, wird ein neuer Knoten erzeugt, dessen Low–Kante auf den 0–Knoten verweist und dessen High–Kante auf den Knoten zeigt, der als Ergebnis der Operation *fire_* bzgl. des Low–Nachfolgeknoten des aktuellen Knotens und dem nächsten Listenknoten vorliegt.

Sollte die Variable des aktuellen Knotens größer sein als die des ersten Listenelements, so ist die entsprechende Stelle in keiner der durch den ZBDD repräsentierten Menge von Markierungen belegt. Sollte sie also zu den Vorstellen bzw. Vor- und Nachstellen der zu schaltenden Transition gehören, kann diese nicht schalten und es wird in diesen beiden Fällen der 0-Knoten als Ergebnis geliefert. Handelt es sich allerdings um eine Nachstelle, so hängt das Schalten der Transition nicht von ihr ab. Als Ergebnis wird ein neuer Knoten erzeugt, dessen Low-Kante auf den Low-Knoten verweist und dessen High–Nachfolgeknoten das fire_–Ergebnis desselben Knotens und dem nächsten Eintrag der Liste darstellt.
([Spr01], S. 62)

Im Vorangegangenen fiel des Öfteren der Begriff einer Liste der Vor– und Nachstellen der aktuellen Transition. In der Implementierung handelt es sich dabei um ein Array vom Typ *unsigned*. Es enthält zweierlei Informationen. Hinter den Feldelementen mit geradem Index verbergen sich die Identifikationsnummern der entsprechenden Stellen. Die ungeraden Feldelemente beherbergen eine Angabe des Typs der Stelle bzgl. der Transition. Eine solche Angabe bezieht sich stets auf die Stelle des unmittelbaren Vorgängerfeldelementes. Das Array ist entsprechend doppelt so groß wie die Anzahl der Stellen der Umgebung der betrachteten Transition. Um die Bedeutung der einzelnen Stellen für das Schalten der Transition zu unterscheiden, wurde der Enumerator Typ *PP_PrePost* vereinbart, der zunächst die Werte *PP_PRE* , *PP_PREPOST* und *PP_POST* (Vorstelle, Vor– und Nachstelle, Nachstelle) aufführt. Jede Instanz einer Transition besitzt solch ein *places*-Feld ihrer Umgebungsstellen.

Interessant ist nun, an welcher Stelle des Programms das für das Schalten einer Transition unverzichtbare Feld ihrer Umgebungsstellen initialisiert wird.
Dafür stellt die Klasse PetriNet die Funktionen *countTransPlaces* und *initTransPlaces* zur Verfügung. Beide Funktionen erhalten als Argumente einen Zeiger und eine Referenz auf die aktuelle Transition. Die erste Funktion geht alle ein- und ausgehenden Kanten durch und zählt dabei die Anzahl der Vor– und Nachstellen und die gesamte Anzahl der Stellen ihrer Umgebung auf und

vermerkt sie in den entsprechenden Attributen der Transition. Wenn auf diese Weise die Anzahl der Umgebungsstellen der Transition bestimmt worden ist, wird das Array zur Speicherung der Vor– und Nachstellen angelegt. Dann können mit Hilfe der zweiten Funktion die Feldelemente initialisiert werden.

Das Vorgehen ähnelt dem der Zählfunktion. Es werden alle ein– und ausgehenden Kanten der Transition auf ihren Typ hin überprüft. Die Identifikationsnummer der Stelle, die durch die Kante mit der Transition verbunden ist (stimmt nicht mit der des Feldes der Stellen überein, wenn die Stellen umgeordnet wurden - Variablenordnung), wird in einem Feldelement mit geradem Index gespeichert, die Angabe über den Stellentyp in Feldelementen mit ungeradem Index. Dazu stehen, wie erwähnt, die Werte des Enumerators *PP_PrePost* zur Verfügung. Ist beispielsweise eine Transition durch eine eingehende Kante mit einem Platz verbunden, so wird an der entsprechenden Stelle des Feldes der Typ *PP_PRE* vermerkt, falls es sich um eine einfache Kante gehandelt hat. Im Falle einer Lesekante muss der Typ *PP_PREPOST* eingetragen werden.

Bei jedem neuen Eintrag wird ein Zähler inkrementiert. Das Feld besitzt 2*(Anzahl der Umgebungsstellen) +1 Einträge. Der letzte Eintrag entspricht genau der Anzahl aller Stellen in der Umgebung der Transition. Nachdem das *places*-Feld einer Transition initialisiert wurde, ist nicht sichergestellt, dass die Reihenfolge der Stellen der Variablenordnung der ZBDDs entspricht. Deshalb werden die Einträge des Feldes von der Funktion *sortTransPlaces* sortiert, die als Argument das entsprechende Feld erhält.

Veränderungen

Vor diesem Hintergrund ist es nun ein Leichtes, EEMC zu erweitern, so dass auch Netze mit Inhibitorkanten eingelesen und verarbeitet werden können. Neben den drei angesprochenen, sind dafür in weiteren Funktionen größtenteils nur sehr kleine Veränderungen vorzunehmen.
Die folgenden Veränderungen wurden nach dem Vorbild des Pakets dssz-ctl-2.0 vorgenommen.

Veränderungen am Programm BAPN2BN

net.h :
Erweiterung des Enumerators *arctype* um die Einträge *Inhibitor* und *InhibitorPost;*
net.cc :
mk_arc
Da zunächst geprüft wird, ob es sich um eine zulässige Kantenart handelt, muss der Typ „inhibitor„ berücksichtigt werden. Beim Einfügen einer Kante von einer Stelle zu einer Transition vom Typ *Inhibitor* müssen Kanten zu Nachplätzen der Transition untersucht und im Falle einer zurückführenden Kante mit dem Typ *InhibitorPost* versehen werden. Das verbleibende Einfügen der Kante gestaltet sich anschließend analog zu den beiden übrigen Kantenarten.
Soll eine Kante der umgekehrten Richtung eingefügt werden, so darf sie nicht den

Typ *ReadArc* oder *Inhibitor* besitzen. Wurde dies ausgeschlossen, werden alle Kanten zu Vorplätzen der Transition untersucht. Ist eine Inhibitorkante von der gleichen Stelle aus dabei, erhält die sie den Typ *InhibitorPos*t. Wie bei einer Lesekante handelt sich also um eine bidirektionale Kante. Ihre zugehörige Transition kann nur bei Nichtbelegung der jeweiligen Stelle schalten. Nach dem Schalten ist die Stelle markiert.

Veränderungen am Programm BNEEMC
net.h :
net.cc :
Siehe BAPN2BN
pn.cc :
initTransPlaces
Bei der Initialisierung des Feldes *places* einer Transition muss zusätzlich geprüft werden, ob eine Kante den Typ *Inhibitor* oder *InhibitorPost* besitzt, dann wird sie dem jeweiligen Feldelement *PP_INH* bzw. *PP_INHPOST* zugewiesen.
SafeQ:
Diese Funktion überprüft ein Netz darauf hin, ob es sicher ist. Dabei wird für jede Transition die Menge der Markierungen ermittelt, bei der diese schaltfähig ist. Wenn bei keiner dieser Markierungen eine Nachstelle markiert ist, gilt das Netz als sicher. Ausgehend vom gesamten Zustandsraum wird dementsprechend mit Hilfe der Funktion *subset1* die Menge der Markierungen ermittelt, bei der Stellen mit Einträgen *PP_PRE* bzw. *PP_PREPOST* in *places*, markiert sind. Hier muss für eine Stelle mit dem Eintrag *PP_INH* in places die Funktion *subset0* verwendet werden, da Inhibitorkanten nur bei Nichtbelegung einer Stelle schalfähig sind. Bei der Prüfung, ob derart ermittelte Markierungen auch Nachstellen markieren, wird mit Hilfe von *subste1* die Teilmenge gebildet, bei der eine Stelle mit dem Eintrag *PP_POST* markiert ist. Hier ist die Überprüfung auf Stellen mit dem Eintrag *PP_INHPOST* in *places* auszuweiten.

BwdCycleCheck/
FwdCycleCheck/
SymbolicDFS:
Innerhalb dieser Funktionen wird zunächst die Menge der toten Zustände bestimmt.
Dabei wird für jede Transition das *places* Feld durchmustert. Für jede Stelle, die für eine Transition Vorstelle oder Vor –und Nachstelle ist, werden mit Hilfe der Funktion *subset0* die Markierungen ermittelt, bei denen die Stelle nicht markiert ist. Sie werden zu einer Menge *tmp* vereinigt. Die Menge der toten Stellen repräsentiert zunächst den gesamten Zustandsraum. Nach Abarbeitung jeder Transition ergibt sich die Menge der toten Zustände aus der Durchschnittsbildung von *tmp* und den bisherigen toten Zuständen.
An dieser Stelle müssen auch die Stellen berücksichtigt werden, die im Feld *places* den Eintrag *PP_INH* besitzen. Da sie bei einer toten Markierung belegt sein müssen, ist hier die Funktion *subset1* zu verwenden.

Print:
An dieser Stelle muss die Switch–Anweisung bzgl. der Typinformation im Feld *places* um zwei weitere Fälle für *PP_INH* und *PP_INHPOST* mit den erforderlichen Ausgaben „INH“ und „INHPO“ erweitert werden.

zbdd.h :
Erweiterung des Enumerators *PP_PREPOST* um die Einträge *PP_INH* und *PP_INHPOST*
zbdd.cc :
fire_:
Anfangs wurde die Funktion *fire_* etwas näher erläutert. Damit auch Transitionen mit eingehenden Inhibitorkanten schalten können, sind folgende Veränderungen vorzunehmen: Im Falle einer Übereinstimmung der aktuellen Variablenbeschriftung und des ersten Listenelements müssen zwei weitere Unterscheidungen berücksichtigt werden. Handelt es sich um eine Stelle mit dem Eintrag *PP_INH*, wird ähnlich einer Stelle mit dem Eintrag *PP_PRE* verfahren.
Der Unterschied besteht einzig und allein darin, dass der erste Parameter des erneuten Funktionsaufrufs von *fire_* der Low-Nachfolgeknoten ist. Es muss von einer Unmarkiertheit der Stelle ausgegangen werden. Beim Eintrag *PP_INHPOST* wird wie bei *PP_POST* vorgegangen.
Sollte die Variablenbeschriftung des aktuellen Knotens größer sein als der Wert des ersten Listenelements, und sollte der Eintrag *PP_INH* vorgefunden werden, wird die Funktion *fire_* mit dem gleichen Knoten, aber mit dem nächsten Listenelement aufgerufen.
revFire_:
Die Veränderungen an der Funktion *revFire_* gleichen denen an der Funktion *fire_*, da eine Stelle mit ausgehender Inhibitorkante auch nach dem Schalten der entsprechenden Transition bzgl. ihrer Belegung keine Veränderung erfährt.
fireUnio_:
Die Veränderungen orientieren sich an denen von *fire_*.

4.2 Einsparung komplementärer Stellen

Jochen Spranger gibt in seiner Dissertation einige Verbesserungen an ([Spr01], S.99), mit denen sich beispielsweise die Anzahl der benötigten Stellen reduzieren lässt. Da es bisher unmöglich war, das Schalten einer Transition von der Unmarkiertheit einer Stelle abhängig zu machen, war es nötig, für in der LTL-Formel negierte Stellen komplementäre Stellen einzufügen, die dann über eine Lesekante mit den betroffenen Transitionen des Formelbüchinetzes verbunden werden.

Mit den Inhibitorkanten steht nun ein Werkzeug zur Verfügung, mit dessen Hilfe sich diese zusätzlichen Stellen im Produktbüchinetz einsparen lassen. Soll eine Stelle nicht markiert sein, so kann man sie über eine Inhibitorkante mit der entsprechenden Transition verbinden. Diese wird nur dann schalten können, wenn sich auf der Stelle keine Marke befindet. Die dafür nötigen Veränderungen beschränken sich lediglich auf eine Funktion.

Die Funktion *printBN* in der Datei ba.cc des Programms BAPN2BN gibt das Formelbüchinetz inklusive der Transitionen zu entsprechenden Stellen des Systembüchinetzes im APNN- Format aus. Dabei wird für alle komplementären Stellen und die betroffenen Transitionen eine einfache Hin- und Rückkante ausgegeben. Diese Kanten werden durch eine einzige Inhibitorkante vom ursprünglichen Platz zur Transition ersetzt. In der *main* –Funktion kann der Aufruf der Funktion *InitAPSIDs* entfernt werden. Diese Funktion veranlasst, wenn erforderlich, das Einfügen komplementärer Stellen durch den Aufruf der Funktion *InsertNegPlaces*, die ebenfalls überflüssig wird.

Anstatt die Funktion *printBN* zu verändern, wurde jedoch eine zweite Funktion *printBN_Inhibitor* eingeführt, die der Funktion *printBN* bis auf die besagten Veränderungen entspricht. Es ist nun möglich, durch die zusätzliche Angabe der Option –k beim Programmaufruf das Produktbüchinetz mit komplementären Stellen zu konstruieren. Standardmäßig werden nun im Falle negierter Aussagen Inhibitorkanten verwendet, um eine Unmarkiertheit der jeweiligen Stellen auszudrücken.

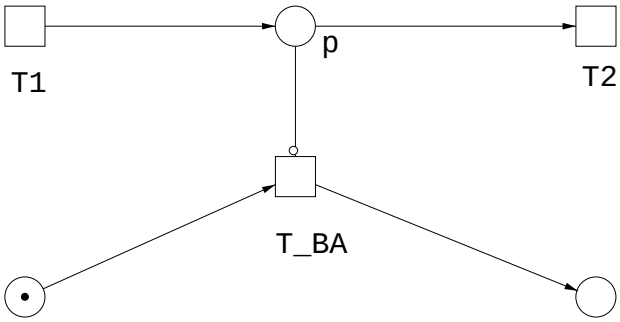


Abbildung 4.1: Unmarkiertheit mit Hilfe einer Inhibitorkante

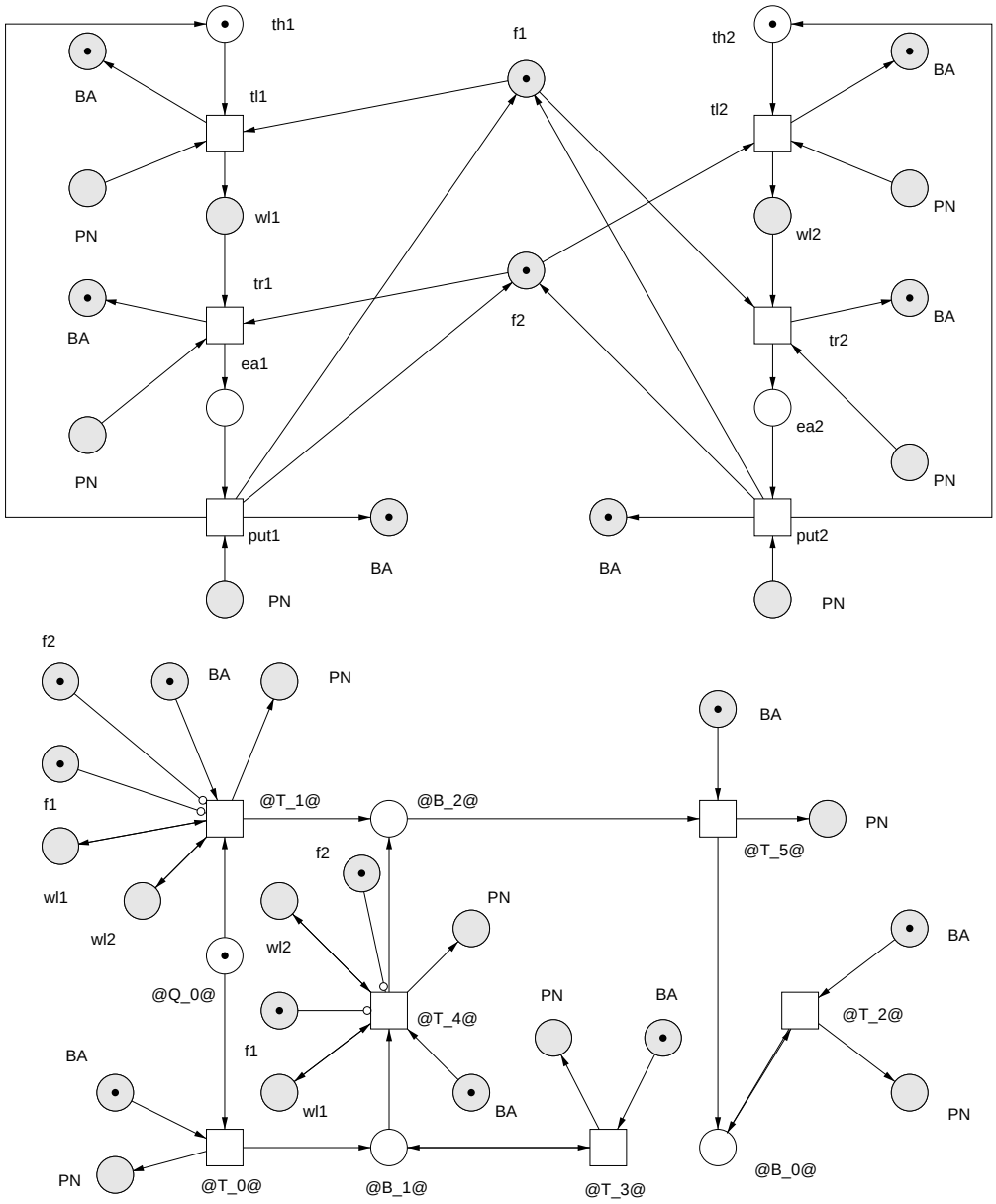


Abbildung 4.2: Produktbüchinetz aus 3.3.3 ohne komplementäre Stellen

5 Zusammenfassung

Der LTL-Modelchecker EEMC basiert auf der Verwendung von ZBDDs zur Repräsentation von Zustandsmengen. Die dafür benötigte Datenstruktur entwickelte Andreas Noack im Rahmen seiner Studienarbeit. Diese ließ sich auf einfache Art und Weise um Funktionen erweitern, die eine angestrebte Visualisierung der sonst dem Programmnutzer verborgenen BDD-Struktur realisieren. Die BDD-Struktur wird durch die Knotenanzahl einer jeden Ebene des BDD charakterisiert. Diese Daten lassen sich nun in ein geeignetes Format exportieren und dann mit Hilfe des Programms VisualizeSize von Dirk Beyer und Michael Vogel graphisch auf dem Bildschirm darstellen. Nach diesem Vorbild wird ein direkter Export der Visualisierungsdaten in das PostScript Format implementiert. Auch das Programm VisualizeSize wurde dahingehend erweitert. Beim Programmstart von EEMC stehen dem Anwender nun Optionen (Siehe Anhang) zur Verfügung, durch die die Visualisierungsdaten wichtiger, beim Modelchecking entstehender, Zustandsmengen in ein gewünschtes Format ausgegeben werden. Darüber hinaus lässt sich das Umordnen der Stellen und Transitionen, das in der Regel eine Verringerung der Knotenanzahl des BDD bzw. eine Beschleunigung des Schaltens von Transitionen bewirkt, deaktivieren. So können beispielsweise die Veränderungen der BDD-Struktur durch Verzicht einer geeigneten Umordnung der Stellen sichtbar gemacht werden.
Da dem CTL-Modelchecking-Paket dssz-ctl-2.0 die gleiche ZBDD-Datenstruktur zugrunde liegt, sollte diese BDD-Visualisierung ohne weiteres auf das Paket übertragen werden können.

Die von Jochen Spranger entwickelte Symbolische Tiefensuche ist ein on-the-fly Verifikationsverfahren mit linearem Aufwand bzgl. der Größe des zugrundeliegenden Graphen. Der bei der Tiefensuche benötigte Stapel enthält im Falle eines negativen Resultats des Modelcheckings Informationen, die zur Ermittlung eines Gegenbeispiels herangezogen werden können. Findet man eine Zustandsfolge, die aus jeder der auf dem Stapel befindlichen Zustandsmengen einen konkreten Zustand erhält, so entspricht diese Sequenz einem Gegenbeispiel. Die für diese Zustandsfolge verantwortliche Transitionssequenz ist dann eine geeignete Form, um das Gegenbeispiel anzugeben. Das entscheidende Problem bei der Ermittlung einer solchen Transitionsfolge stellt das Finden eines Überganges von einem einzelnen Zustand einer Menge des Stapels zu einem Zustand der nächsten Menge dar. Ähnlich einer Tiefensuche kann man, beim Initialzustand beginnend, mögliche Transitionsfolgen so lange verfolgen, bis ein Zustand der ersten Menge (eigentlich der zweiten Menge, da der Initialzustand selbst auch auf dem Stapel liegt) gefunden wird. Von ihm aus wird dann auf die gleiche Weise weiter verfahren, um zu einem Zustand der nächsten Zustandsmenge zu gelangen. Dieses Verfahren führt jedoch nur bei einem kleinen Petrinetz (Beispiel aus 3.3.3) zu einem Ergebnis. Für größere Netze explodiert durch die enorme Anzahl der möglichen Transitionsfolgen die Anzahl der zu überprüfenden Zustände. Auf der Grundlage des Stapelinhalts lässt sich auf

diesem Wege kein bzgl. der Laufzeit befriedigendes Ergebnis erzielen.

EEMC war bislang in der Lage, Petrinetze mit einfachen Kanten und Lesekanten zu verarbeiten. Da Inhibitorkanten in zunehmendem Maße bei der Systemmodellierung mit Petrinetzen an Bedeutung gewinnen, war es wünschenswert, auch Petrinetzmodelle mit Inhibitorkanten mit Hilfe von EEMC verifizieren zu können. Für diese Anpassung mussten die Datenstrukturen für die interne Darstellung des Netzes erweitert werden. Inhibitorkanten zeichnen sich dadurch aus, dass sie eine Transition nur bei Nichtbelegung der entsprechenden Stellen schalten lassen. Dieses besondere Schaltverhalten wurde ebenfalls auf die entsprechenden Datenstrukturen übertragen.

Mit Hilfe von Inhibitorkanten kann man nun das Schalten von Transitionen auch von der Unmarkiertheit von Stellen abhängig machen, was vorher stets zusätzliche Stellen erforderte. Dadurch lassen sich bei der Konstruktion des Produktbüchinetzes aus dem System- und dem Formelbüchinetz, das die Grundlage aller in EEMC realisierten Modelchecking-Verfahren bildet, in Abhängigkeit von der konkreten LTL-Formel unter Umständen zusätzliche Stellen einsparen.

Hinsichtlich der Aufgabenstellung wurden die Visualisierung der Struktur von BDDs und die Erweiterung des Kantenspektrums vom EEMC um Inhibitorkanten umgesetzt. Die Ermittlung eines Gegenbeispiels bei der Symbolischen Tiefensuche unter Verwendung der erzeugten Zustandsmengen des Stapels erwies sich jedoch als nicht zufrieden stellend realisierbar.

6 Anhang

Benutzung des Programms

Der Modelchecker wird mit dem Kommando
dsszltl [-d|-b|-f] [-s|-m|-l|-s]
(-k) (-p) (-t)
(-B Export-Datei | -D Export-Datei | -P Export-Datei)
-i Petrinetz-Datei –c LTL-Formel-Datei
aufgerufen.

Es stehen drei Modelchecking-Verfahren zur Verfügung:
-d SymbolicDFS
-b BwdCycleCheck
-f FwdCycleCheck

Weitere Optionen sind:
-s kleines Netz
-m mittleres Netz
-l großes Netz
-x sehr großes Netz

-k Konstruktion des Produktbüchinetzes mit komplementären Stellen
-p keine Variablenordnung
-t keine Umordnung der Transitionen

-P Export der BDD-Visualisierungsdaten in PostScript
-D Export der BDD-Visualisierungsdaten in dat (VisualizeSize)
-B Export der BDD-Visualisierungsdaten in beide Formate

Anmerkung zur Visualisierung:

Im Falle einer gewünschten Visualisierung werden die BDD-Visualisierungsdaten folgender Zustandsmengen in das gewählte Format exportiert:
BwdCycleCheck
- Menge aller vom Initialzustand aus erreichbaren Zustände (ReachableSet)
- Zustandsmengen der ermittelten Fixpunkte
FwdCycleCheck
- Zustandsmengen der ermittelten Fixpunkte
SymbolicDFS
- Zustandsmengen der beim Finden eines akzeptierenden Kreises auf dem Stapel befindlichen Knoten

Beispiele für die BDD-Visualisierung

Abbildungen A1-A2
Verfahren: BwdCycleCheck
Petrinetz: closed_system1
LTL-Formel:
! (G ((arm1_release_angle && arm1_release_ext) -> (press_stop && ! press_at_upper_pos)))

Abbildungen A3-A4
Verfahren: FwdCycleCheck
Petrinetz: closed_system1
LTL-Formel:
! (G ((arm1_release_angle && arm1_release_ext) -> (press_stop && ! press_at_upper_pos)))

Abbildungen A5-A6
Verfahren: BwdCycleCheck
Petrinetz: closed_system5
LTL-Formel:
! (G ((arm1_pick_up_angle && arm1_pick_up_ext && arm1_magnet_on) -> (arm1_magnet_on U (arm1_release_angle && arm1_release_ext))))

Abbildungen A7-A8
Verfahren: SymbolicDFS
Petrinetz: closed_system1
LTL-Formel:
! (G ((arm1_release_angle && arm1_release_ext) -> (press_stop && ! press_at_upper_pos)))

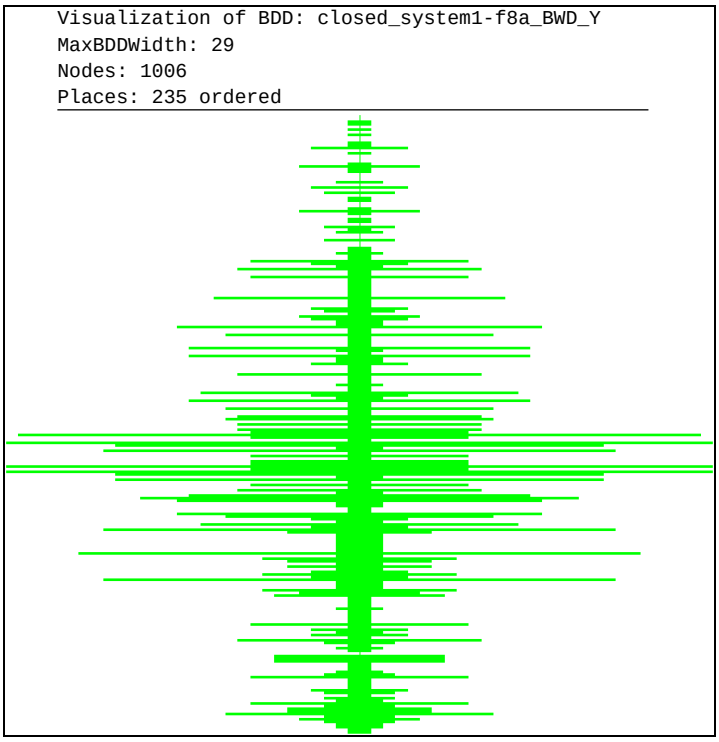


Abbildung A1

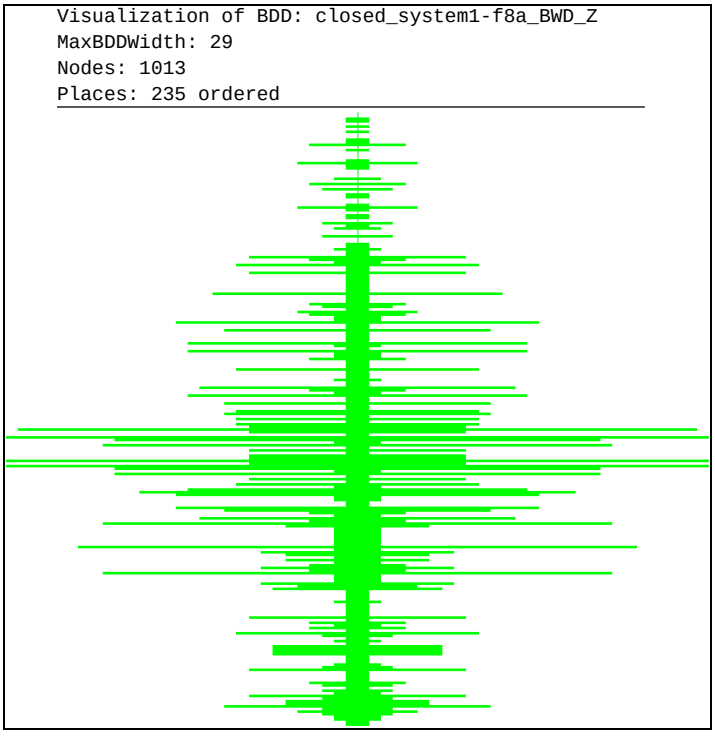


Abbildung A2

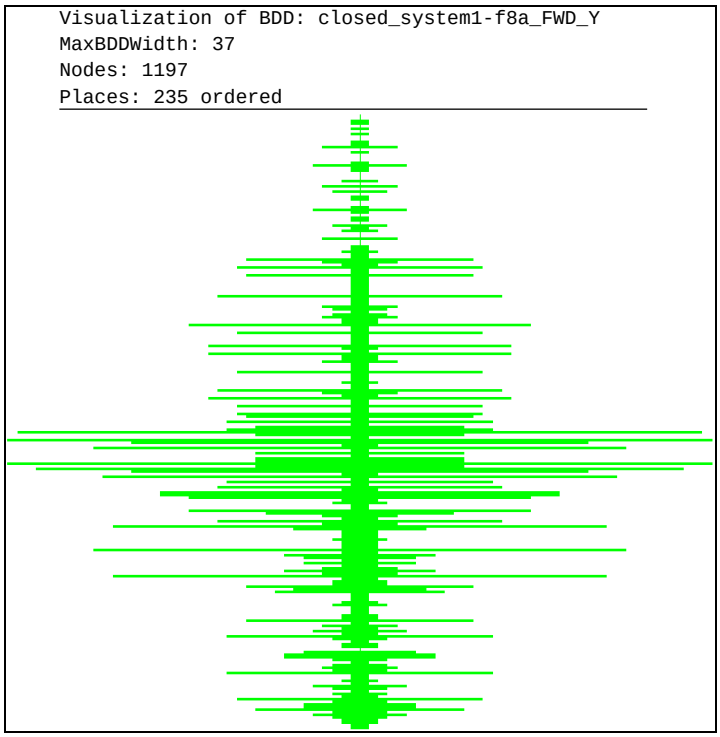


Abbildung A3

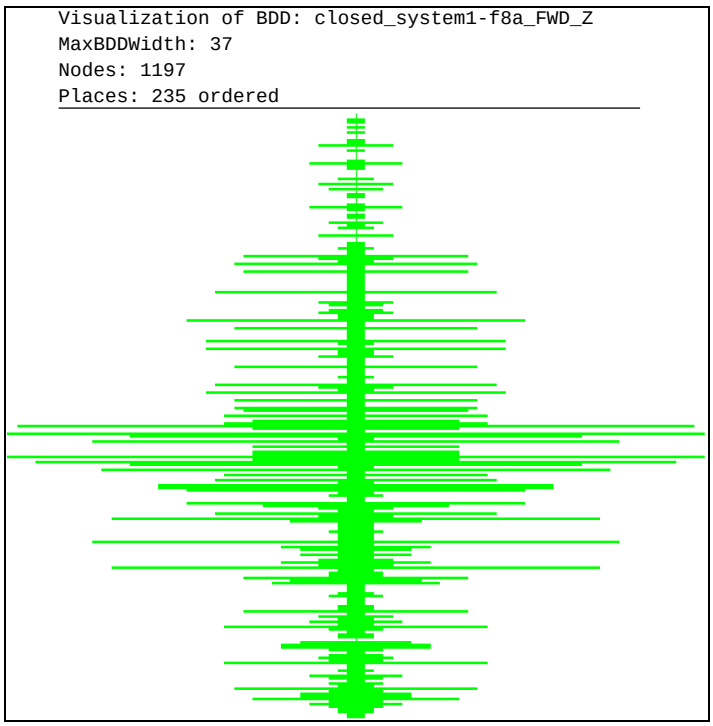


Abbildung A4

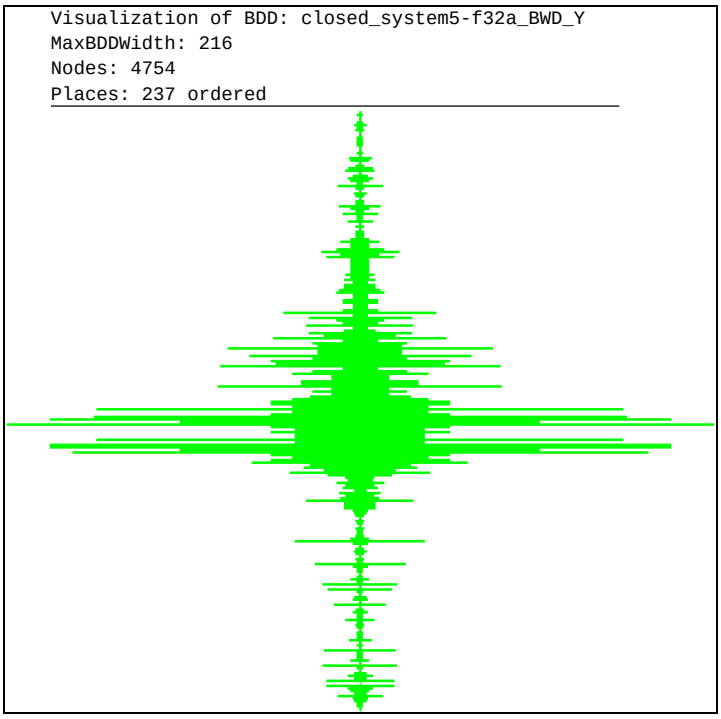


Abbildung A5

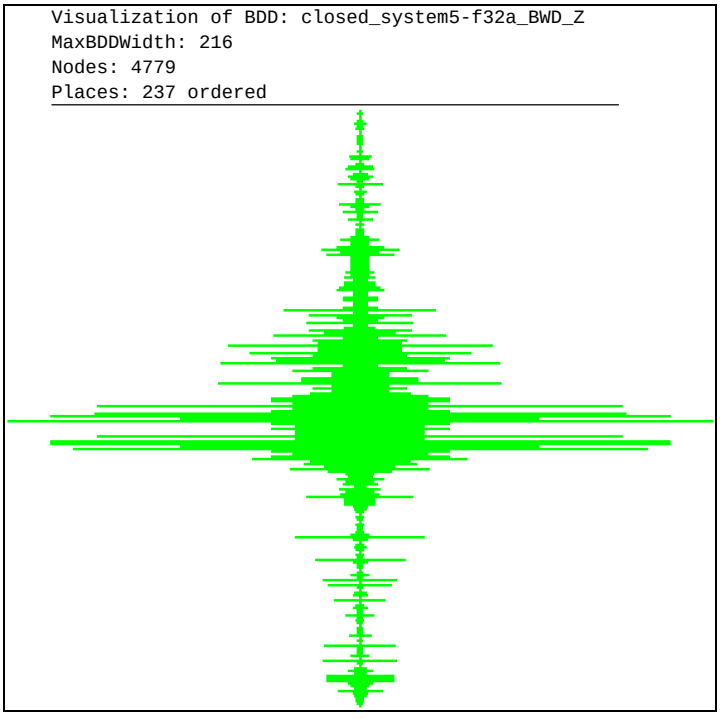


Abbildung A6

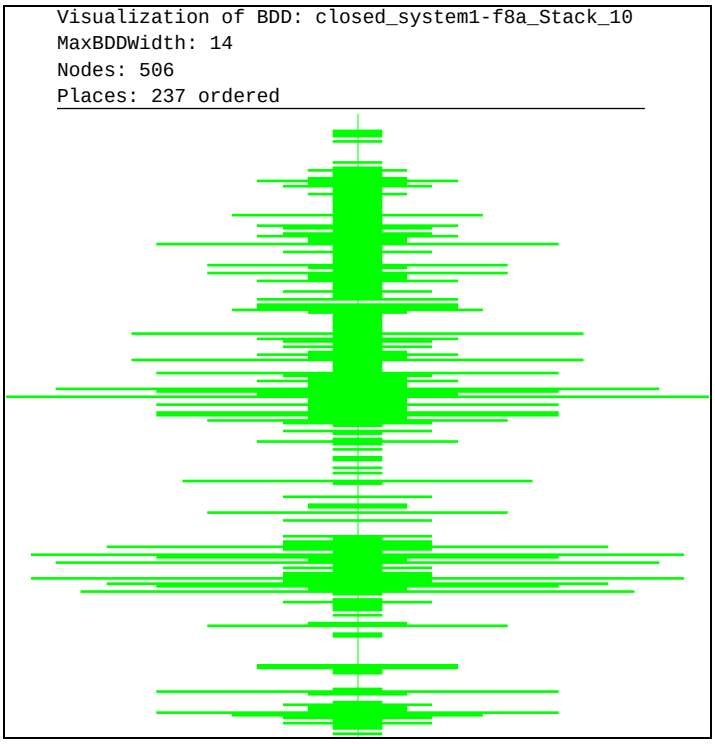


Abbildung A7

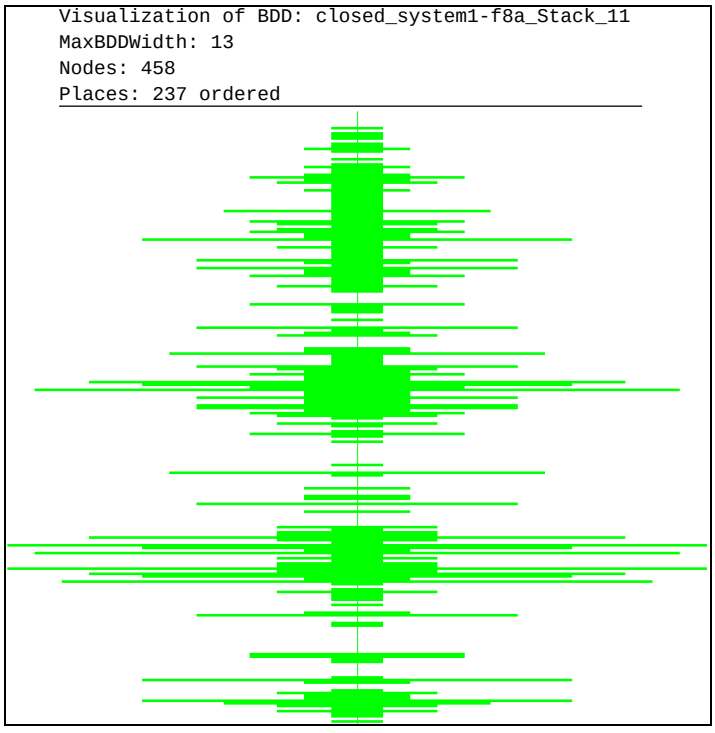


Abbildung A8

7 Literaturverzeichnis

[CGP01] Clarke, E. M.; Grumberg, Jr. O.; Peled, D.A.: Model Checking, Cambridge, Massachusettes
London. England
1999, S. 32

[Noa99] Noack, A.: Ein ZBDD-Paket für effizientes Model Checking von Petrinetzen / Brandenburgische Technische Universität Cottbus. Institut für Informatik. Lehrstuhl für Datenstrukturen und Softwarezuverlässigkeit.
1999

[Spr01] Spranger, J.: Symbolische LTL-Spezifikation von Petrinetzen, Brandenburgische Technische Universität Cottbus. Institut für Informatik.
2001