
Seminar Modellüberprüfung SMV

21. Januar 2002

Hauptseminar an der Universität Ulm
Fakultät für Informatik
Abteilung Künstliche Intelligenz
Prof. Dr. F. von Henke
Holger Pfeifer

Autor: Markus Nosse
MtkNr: 379040

Zusammenfassung

In dieser Seminararbeit werden die theoretischen und algorithmischen Grundlagen der symbolischen Modellüberprüfung so weit beschrieben, dass es möglich sein soll, die Funktionsweise eines automatischen Modellüberprüfers zu verstehen. Daran schließt sich die Vorstellung eines Werkzeugs zur symbolischen Modellüberprüfung – SMV, Symbolic Model Verifier – an. Das Hauptaugenmerk liegt dabei auf der Definitionssprache des Systems.

Inhaltsverzeichnis

1	Einleitung	4
2	Grundlagen	5
2.1	Binäre Entscheidungsdiagramme	5
2.1.1	Direkte Konstruktion von BDDs	6
2.1.2	BDDs als kanonische Darstellungsform für logische Formeln	7
2.1.3	Repräsentation von Kripke Strukturen	7
2.2	Symbolische Modellüberprüfung	8
2.2.1	Fixpunktdarstellung für CTL	9
2.2.2	Algorithmus zur symbolischen Modellüberprüfung	9
3	SMV	10
3.1	Überblick über SMV	11
3.1.1	Typen und Definition von Signalen	11
3.1.2	Module	11
3.1.3	Spezifikation	13
3.2	Sprachkonstrukte von extended SMV	13
3.2.1	Komplexe Typen	13
3.2.2	Bedingte Sprachkonstrukte	14
3.2.3	Iterative Sprachkonstrukte	15
3.3	Fortgeschrittene Modellierungstechniken	15
3.4	Beispiel: Gegenseitiger Ausschluss	16
4	Schlussbemerkung	16

1 Einleitung

Modellüberprüfung ist ein Verfahren zur formalen Verifikation von parallelen, zustandsbasierten Systemen. Das Verfahren prüft ob ein als Modell gegebenes System die gewünschte Spezifikation erfüllt. Von der Erarbeitung der theoretischen Grundlagen über erste Forschungssysteme bis hin zu Softwaresystemen, die Modellüberprüfung in den industriellen Designprozess eingebracht haben, sind weniger als zwei Jahrzehnte vergangen.

Systeme werden als endliche Zustandsautomaten beschrieben, während Spezifikationen mit zeitlichen Logiken wie LTL (Linear Time Logics) oder CTL (Computation Tree Logics) präzise formuliert werden können. Die Grundlage für die symbolische Modellüberprüfung legten Burch und Clarke et al in ihrem 1992 veröffentlichten Paper “Symbolic Model Checking: 10^{20} States and Beyond” [3].

Symbolisch bedeutet in diesem Fall, dass Modelle und Formeln symbolisch repräsentiert und auch manipuliert werden. Damit ist es möglich das größte Problem des Verfahrens, die explosionsartige Zunahme des Zustandsraums bei komplexeren Systemen in den Griff zu bekommen.

Gegenstand dieser Seminararbeit ist SMV, ein Tool zur symbolischen Modellüberprüfung, das zunächst als Ergebnis einer Dissertationsarbeit an der Carnegie Mellon University entstand, welches mittlerweile zu einem System weiterentwickelt worden ist, das Anforderungen im realen Einsatz erfüllt. Allerdings konnten mit Hilfe des ursprünglichen SMV bereits reale Verifikationsaufgaben bewältigt werden. Als Beispiel hierfür soll die Arbeit von Clarke und Grumberg et al dienen, die das vom IEEE spezifizierte Cache-Kohärenz Protokoll FutureBus+ mit Hilfe von SMV überprüft haben und dabei sogar Fehler nachweisen konnten [4].

In den vergangenen Jahren wurde SMV weiterentwickelt, so dass noch komplexere Systeme bewältigt werden können. Dabei ist jedoch zu bemerken, dass vielmehr Modellierungstechniken entwickelt wurden, mit deren Hilfe solche Systeme beispielsweise in kleinere Komponenten aufgeteilt werden können, so dass die Summe der erfolgreich überprüften Teile ein korrektes System ergeben oder dass die Modelle durch das Ausnutzen von Symmetrien vereinfacht werden.

Im weiteren Verlauf dieser Arbeit wird auf den Sprach- und Funktionsumfang der aktuellen Version von (extended) SMV eingegangen. Um ein besseres Verständnis der Funktionsweise von SMV zu gewährleisten, wird zunächst auf die Grundlagen der symbolischen Modellüberprüfung eingegangen. Neben der grundlegenden Datenstruktur der geordneten binären Entscheidungsdiagramme (OBDD) wird gezeigt, wie CTL Formeln als Fixpunktgleichungen ausgedrückt werden können. Diese Kombination aus OBDDs und Fixpunktcharakterisierungen ermöglicht verhältnismäßig effiziente Algorithmen für das gesamte Verfahren.

Danach werden die grundlegenden Elemente der Beschreibungssprache von SMV vorgestellt und basierend darauf komplexere Sprachkonstrukte. Zwei Beispiele verdeutlichen wie SMV Programme aussehen. Im Anschluss daran wird noch ein Ausblick auf die in extended SMV neu eingeführten Modellierungstechniken gegeben.

2 Grundlagen

2.1 Binäre Entscheidungsdiagramme

Da sich binäre Entscheidungsdiagramme leicht aus binären Entscheidungsbäumen herleiten lassen, ist es sinnvoll, diese kurz zu betrachten. Ein Entscheidungsbaum ist ein gerichteter, azyklischer Graph (V, E) mit einem ausgezeichneten Wurzelknoten, der eine boolesche Funktion $f(x_1, \dots, x_n) \rightarrow \{0, 1\}$ repräsentiert. Es gibt zwei Arten von Knoten, zum einen terminale Knoten, die entweder mit 0 oder mit 1 bezeichnet sind und zum anderen nichtterminale Knoten. Diese werden mit einer der Variablen x_i der dargestellten booleschen Funktion bezeichnet. Von nichtterminalen Knoten gehen zwei Kanten aus. Die mit 0 (bzw. 1) bezeichnete Kante weist auf den Teilbaum, der mit $low(\nu)$ (bzw. $high(\nu)$) bezeichnet wird.

Die Funktion $value(\nu)$ liefert den Wert eines terminalen Knoten, während $var(\nu)$ die Variable eines nichtterminalen Knoten ergibt.

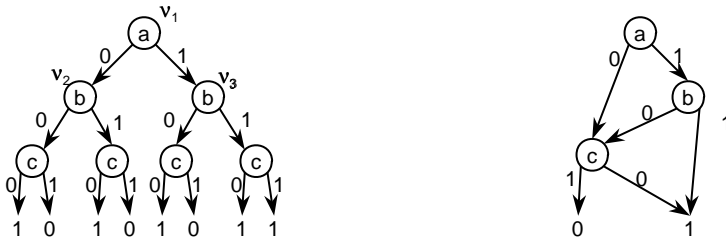


Abbildung 1: Entscheidungsbaum und -diagramm für die einfache boolesche Funktion $f = a \wedge b \vee \neg c$. Am Knoten ν_1 des Entscheidungsbaums lassen sich die oben eingeführten Bezeichnungen verdeutlichen. Es gilt $var(\nu_1) = a$, $low(\nu_1) = \nu_2$ und $high(\nu_1) = \nu_3$.

Der Wahrheitswert der Funktion für eine gegebene Variablenbelegung kann ermittelt werden indem der Entscheidungsbaum entsprechend dieser Belegung traversiert wird. An einem nichtterminalen Knoten $\nu \in V$ wird der Pfad bei $low(\nu)$ fortgesetzt, wenn $var(\nu)$ mit 0 belegt ist, andernfalls bei $high(\nu)$. Der nach $|V|$ Entscheidungen erreichte terminale Knoten ergibt den Wahrheitswert der Funktion für die gegebene Belegung.

Der Nachteil binärer Entscheidungsbäume ist, dass sie die gleiche Größe wie die Wahrheitstabelle der booleschen Funktion haben, d.h. solch ein Baum hat 2^n Blätter bzw. Pfade und insgesamt $2^n - 1$ Knoten. Bei einer genaueren Analyse von Entscheidungsbäumen fällt jedoch auf, dass es etliche gleichartige Teilbäume und terminale Knoten gibt. Dies ist auch aus dem in Abbildung 1 gezeigten Entscheidungsbaum ersichtlich.

An dieser Stelle setzt die Idee der binären Entscheidungsdiagramme an: Durch das Zusammenfassen isomorpher Teilbäume und terminaler Knoten sowie der Eliminierung überflüssiger Entscheidungen kann die Größe der Graphen drastisch reduziert werden.

Für das Zusammenfassen isomorpher Teilgraphen gelten die folgenden Regeln: Alle mit 0 bzw. 1 beschrifteten terminalen Knoten werden zu jeweils einem mit 0 bzw. 1 beschrifteten Blatt zusammengefasst, wobei alle in die bishe-

rigen Blätter eingehenden Kanten erhalten bleiben. Zwei nichtterminale Knoten $\nu_1, \nu_2 \in V$ können zu einem Knoten zusammengefasst werden, wenn ν_1 und ν_2 mit der selben Variablen beschriftet sind und in ihren beiden Kindern übereinstimmen. Formal können diese Bedingungen folgendermaßen ausgedrückt werden: (1) $var(\nu_1) = var(\nu_2)$ und (2) $low(\nu_1) = low(\nu_2) \wedge high(\nu_1) = high(\nu_2)$. Die beiden Teilbäume mit den Wurzelknoten $low(\nu_2)$ und $low(\nu_3)$ in Abbildung 1 erfüllen diese Bedingungen und sind damit isomorph.

Eine überflüssige Entscheidung an einem nichtterminalen Knoten liegt vor, wenn die beiden Kanten auf ein und den selben Teilgraphen verweisen. Das heisst, ein Knoten $\nu \in V$ kann aus dem Diagramm entfernt werden, wenn $low(\nu) = high(\nu)$. Die eingehende Kante in ν wird durch zwei Kanten zu $low(\nu)$ bzw. $high(\nu)$ ersetzt. Ein Beispiel für diesen Fall ist der Knoten $high(\nu_3)$ in Abbildung 1.

2.1.1 Direkte Konstruktion von BDDs

Ist eine boolesche Funktion als logischer Ausdruck gegeben (z.B. als Summe von Produkten o.ä.), so ist eine direkte Konstruktion des BDD möglich. Unter Verwendung der Shannon Expansion lässt sich eine Funktion $f(x_1, \dots, x_n)$ wie folgt darstellen

$$f(x_1 \dots x_n) = x_i f_1(x_1 \dots x_{i-1}, x_{i+1} \dots x_n) \vee \neg x_i f_0(x_1 \dots x_{i-1}, x_{i+1} \dots x_n)$$

wobei $f_0 = f(x_1 \dots x_{i-1}, 0, x_{i+1} \dots x_n)$
 und $f_1 = f(x_1 \dots x_{i-1}, 1, x_{i+1} \dots x_n)$.

Zur Erstellung eines BDD kann ein Algorithmus eingesetzt werden, der diese Shannon Expansion wiederholt für jede Variable der Funktion anwendet [1]. Für jede Variable wird ein Knoten ν erstellt, der mit dem Namen der Variablen bezeichnet wird. Die beiden Kinder $low(\nu)$ und $high(\nu)$ ergeben sich durch rekursive Anwendung des Algorithmus aus den Formeln f_0 und f_1 . Eine beispielhafte Anwendung des Algorithmus ist in Abbildung 2 gegeben. Die Größe des resultierenden BDD kann noch weiter reduziert werden, wenn vor jedem Rekursionsschritt geprüft wird, ob ein entsprechendes Entscheidungsdiagramm bereits erstellt wurde.

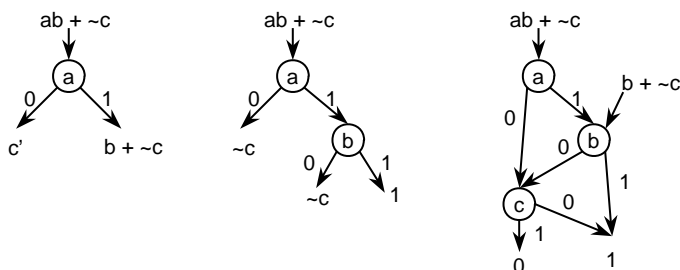


Abbildung 2: Direkte Konstruktion des BDD für die Funktion $a \wedge b \vee \neg c$. Am Knoten der Variablen b wird angedeutet, dass mit einem BDD mehrere Funktionen gleichzeitig dargestellt werden können.

2.1.2 BDDs als kanonische Darstellungsform für logische Formeln

Die Kombination aus diesem Algorithmus mit den oben beschriebenen Minimierungsregeln ergibt zwar kleinstmögliche BDDs, aber je nach dem in welcher Reihenfolge die Variablen durch die Shannon Expansion ersetzt werden, ergeben sich unterschiedliche Entscheidungsdiagramme für eine gegebene Funktion. Durch die Einführung einer totalen Ordnung auf den Variablen der Funktion kann dies vermieden werden [2].

Bryant erweitert die Definition des Funktionsgraphen um eine weitere Funktion $index : V \rightarrow \{1, \dots, n\}$. Damit wird die Bedingung für die Ordnung des Diagramms formuliert: Für einen nichtterminalen Knoten ν , dessen Nachfolger $low(\nu)$ ebenfalls kein Terminal ist, muss $index(\nu) < index(low(\nu))$ gelten. Gleichmaßen muss für einen nichtterminalen Nachfolger $high(\nu)$ von ν $index(\nu) < index(high(\nu))$ gelten. Offensichtlich kann diese Bedingung bei der Konstruktion von BDDs leicht erfüllt werden, wenn die Ersetzung der Variablen entsprechend der gegebenen Ordnung erfolgt¹.

Diese geordneten binären Entscheidungsdiagramme (OBDD) ermöglichen eine "vernünftige" Darstellung der üblicherweise auftretenden Funktionen. Zusätzlich sind minimierte OBDDs eine kanonische Darstellung für boolesche Funktionen. Der Test, ob zwei Funktionen äquivalent sind entspricht der Prüfung auf Isomorphie zwischen den OBDDs der beiden Funktionen. Ebenso reduziert sich der Test auf Erfüllbarkeit auf Prüfung von Isomorphie mit dem OBDD für die konstante Funktion 0.

Neben der Definition der Datenstruktur und deren Eigenschaften stellt Bryant in seinem Paper noch einige Algorithmen, die auf OBDDs operieren vor. Mittels *Reduce* kann ein Diagramm G mit einem Aufwand von $O(|G| \log |G|)$ auf die kanonische Form reduziert werden. Die Kombination zweier Formeln durch logische Operatoren übernimmt *Apply*. Der Aufwand ist beschränkt durch das Produkt der Größe der beiden Diagramme. Die Shannon Expansion wird von *Restrict* mit gleichem Aufwand wie *Reduce* implementiert.

Die Zeitkomplexität einer einzelnen Operation ist im schlechtesten Fall also beschränkt durch das Produkt der Größe der Graphen der Operanden. Einer der Nachteile geordneter BDDs ist allerdings, dass es schwierig sein kann eine gute Ordnung für die Variablen zu finden. Die Bestimmung einer optimalen Ordnung an sich ist coNP vollständig. Allerdings lassen sich für die meisten Probleme ausreichend gute Ordnungen finden, und unter Anwendung von Heuristiken ist es auch möglich, automatisch Ordnungen zu bestimmen, die zwar die Anzahl der BDD Knoten und damit die Ausführungszeit verringern, jedoch nicht optimal sind.

2.1.3 Repräsentation von Kripke Strukturen

Eine Relationen über dem Wertebereich $\{0, 1\}$ kann auch durch die zugehörige charakteristische Funktion f_Q beschrieben werden. Bei einer n -stelligen Relation

¹Im vorigen Abschnitt wurden Knoten und die ihnen zugeordneten Variablen unterschieden. Um also präzise zu sein, müsste der Definitionsbereich der Indexfunktion die Menge der Variablen der Funktion $\{x_1, \dots, x_n\}$ sein und jede Verwendung von $index(\nu)$ müsste konsequenterweise durch $index(var(\nu))$ ersetzt werden. Allerdings kann ein Knoten immer eindeutig einer Variablen zugeordnet werden, so dass die verwendete Notation zwar nicht präzise aber doch korrekt ist.

Q ist f_Q definiert als

$$f_Q(x_1, \dots, x_n) = 1 \iff Q(x_1, \dots, x_n).$$

Interpretiert man 0 als *falsch* und 1 als *wahr*, so kann f_Q als eine boolesche Funktion aufgefasst und folglich durch ein OBDD dargestellt werden.

Die Übergangsrelation R einer Kripke Struktur (S, R, L) mit der Zustandsmenge S und der Beschriftungsfunktion L ist im allgemeinen jedoch nicht über dem Wertebereich $\{0, 1\}$ sondern über einem beliebigen, endlichen Wertebereich D definiert. Die $|D|$ Elemente der Menge D lassen sich durch binäre Codewörter der Länge $m = \lceil \log |D| \rceil$ darstellen. Sei die Codierung durch eine bijektive Funktion $\phi : \{0, 1\} \rightarrow D$ gegeben, dann ist durch

$$\hat{R}(\vec{x}_1, \dots, \vec{x}_n) = R(\phi(\vec{x}_1), \dots, \phi(\vec{x}_n))$$

eine $m \times n$ stellige Relation \hat{R} über dem Wertebereich $\{0, 1\}$ gegeben, die äquivalent zur Übergangsrelation R ist. Stellt man die charakteristische Funktion $f_{\hat{R}}$ der codierten Übergangsrelation \hat{R} als OBDD dar, so ist dieses Diagramm eine symbolische Darstellung des Modells (S, R, L) .

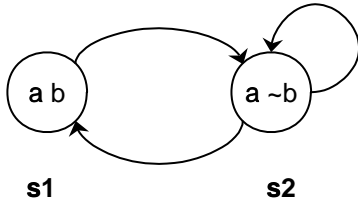


Abbildung 3: Eine einfache Kripke Struktur mit zwei Zuständen. Siehe [5].

Als Beispiel betrachte man die Kripke Struktur in Abbildung 3. Es gibt zwei Zustandsvariablen a und b . Um Folgezustände zu beschreiben, werden zwei weitere Variablen a' und b' eingeführt. Der Zustandsübergang von s_2 zu s_1 wird dann von der Konjunktion

$$(a \wedge \neg b \wedge a' \wedge b')$$

beschrieben. Die komplette charakteristische Funktion der Übergangsrelation ist gegeben durch

$$\underbrace{(a \wedge b \wedge a' \wedge \neg b')}_{s_1 \rightarrow s_2} \vee \underbrace{(a \wedge \neg b \wedge a' \wedge b')}_{s_2 \rightarrow s_1} \vee \underbrace{(a \wedge \neg b \wedge a' \wedge \neg b')}_{s_2 \rightarrow s_2}.$$

Die drei Disjunktionen in dieser Formel entsprechen den drei Zustandsübergängen der Kripke Struktur. Das OBDD für diese Formel ist eine symbolische Darstellung der Übergangsrelation des Modells.

2.2 Symbolische Modellüberprüfung

Es wird vorausgesetzt, dass der Leser mit den Grundkonzepten von CTL vertraut ist. Auf eine nähere Beschreibung der Logik oder eine formale Definition wird hier

verzichtet. Um Missverständnissen vorzubeugen wird lediglich kurz die verwendete Nomenklatur erläutert. Universelle Quantifizierung der Berechnungspfade wird mittels **A** ausgedrückt, existenzielle Quantifizierung mit **E**. Die zeitlichen Operatoren sind **X** für eine Bedingung, die zum nächsten Zeitpunkt erfüllt sein soll, **F** für eine Bedingung, die zu einem beliebigen Zeitpunkt in der Zukunft erfüllt sein soll. **G** steht für eine Bedingung die global für alle Zeitpunkte in der Zukunft gelten soll sowie die Release bzw. Until Operatoren **R** und **U**.

2.2.1 Fixpunktdarstellung für CTL

Der oben beschriebene Ansatz der Darstellung von Relationen über endlichen Wertebereichen durch OBDDs ist so flexibel, dass auch Mengen durch OBDDs beschreibbar sind. Mengen sind nichts anderes als einstellige Relationen.

Diese Beobachtung und die effizienten Algorithmen auf OBDDs lassen es interessant erscheinen Eigenschaften von Systemen symbolisch, basierend auf Mengen zu beschreiben. Eine Möglichkeit dies zu tun bieten Fixpunktcharakterisierungen von CTL Formeln. Kleinste Fixpunkte werden bezeichnet mit $\mu Z. \tau(Z)$, größte Fixpunkte mit $\nu Z. \tau(Z)$ mit $\tau : \mathcal{P}(S) \rightarrow \mathcal{P}(S)$, wobei wiederum eine Kripke Struktur $M = (S, R, L)$ zugrunde liegt. Für die grundlegenden CTL Operatoren ergeben sich die Fixpunktdarstellungen wie folgt:

- **AF** $f_1 = \mu Z. f_1 \vee \mathbf{A} \mathbf{X} Z$
- **EF** $f_1 = \mu Z. f_1 \vee \mathbf{E} \mathbf{X} Z$
- **AG** $f_1 = \nu Z. f_1 \wedge \mathbf{A} \mathbf{X} Z$
- **EG** $f_1 = \nu Z. f_1 \wedge \mathbf{E} \mathbf{X} Z$
- **A** $[f_1 \mathbf{U} f_2] = \mu Z. f_2 \vee (f_1 \vee \mathbf{A} \mathbf{X} Z)$
- **E** $[f_1 \mathbf{U} f_2] = \mu Z. f_2 \vee (f_1 \vee \mathbf{E} \mathbf{X} Z)$
- **A** $[f_1 \mathbf{R} f_2] = \nu Z. f_2 \wedge (f_1 \vee \mathbf{A} \mathbf{X} Z)$
- **E** $[f_1 \mathbf{R} f_2] = \nu Z. f_2 \wedge (f_1 \vee \mathbf{E} \mathbf{X} Z)$

Diese Fixpunktdarstellungen für CTL Formeln existieren, da CTL letztlich nur eine abkürzende Schreibweise für das μ -Kalkül [3] ist. In dem zitierten Artikel wird ein Dialekt des Kalküls definiert. Mit der Angabe eines Modellüberprüfungs Algorithmus für das μ -Kalkül und der Definition von CTL als syntaktische Abkürzung für in diesem Kalkül ausgedrückte Formeln sind die theoretischen Grundlagen für symbolische Modellüberprüfung und damit auch SMV in diesem Artikel beschrieben.

Die Berechnung kleinster bzw. größter Fixpunkte kann iterativ in einer endlichen Zahl von Schritten erfolgen. Entsprechende Algorithmen *Lfp* und *Gfp* werden in [5] als Ergebnis der Diskussion von Eigenschaften obiger Fixpunktcharakterisierungen angegeben.

2.2.2 Algorithmus zur symbolischen Modellüberprüfung

Mit den bisher beschriebenen Formalismen ist es möglich einen Algorithmus zur symbolischen Modellüberprüfung zu implementieren. Als Eingabe für den Algorithmus dient eine zu überprüfende CTL Formel, das Ergebnis ist ein OBDD, das

genau die Menge der Zustände darstellt, die die gegebene Formel erfüllen. Der Algorithmus bezieht sich auf ein Modell $M = (S, R, L)$, dessen Übergangsrelation R als OBDD gegeben ist.

Der Algorithmus *Check* wird induktiv über der Struktur von CTL Formeln definiert. Sei f die zu überprüfende CTL Formel. Falls f eine atomare Aussage a ist, so ist das Ergebnis ein OBDD, das die Zustände in denen a erfüllt ist repräsentiert. Ist $f = f_1 \wedge f_2$ oder $f = \neg f_1$, so ergibt sich das Ergebnis durch die Anwendung von Bryant's *Apply* (siehe Abschnitt 2.1.2 und [2]) mit den Argumenten $Check(f_1)$ und $Check(f_2)$. Es bleiben Formeln der Form $\mathbf{EX} f$, $\mathbf{E}[f \mathbf{U} g]$ und $\mathbf{EG} f$, die von jeweils entsprechenden Prozeduren bearbeitet werden:

$$\begin{aligned} Check(\mathbf{EX} f) &= CheckEX(Check(f)), \\ Check(\mathbf{E}[f \mathbf{U} g]) &= CheckEU(Check(f), Check(g)), \\ Check(\mathbf{EG} f) &= CheckEG(Check(f)). \end{aligned}$$

Dabei ist zu bemerken, dass die Prozeduren *CheckEX*, *CheckEU* und *CheckEG* im Gegensatz zu *Check* keine CTL Formel sondern OBDDs als Eingabe erwarten. Eine Implementierung von *CheckEX* ergibt sich aus der Überlegung, dass die Formel $\mathbf{EX} f$ in einem Zustand s erfüllt ist, falls s einen Nachfolger hat, in dem f gilt. Formal gilt

$$CheckEX(f(\vec{v})) = \exists \vec{v}' [f(\vec{v}') \wedge R(\vec{v}, \vec{v}')].$$

$\mathbf{EX} f$ ist also erfüllt, wenn ein Zustand mit der Variablenbelegung \vec{v}' existiert, so dass f durch diese Variablenbelegung erfüllt wird und der Zustand von s aus in einem Schritt erreichbar ist.

Für die Prozedur *CheckEU* dient die Fixpunktdarstellung für den \mathbf{EU} Operator als Ausgangspunkt. Das Ergebnis der Fixpunktgleichung

$$\mathbf{E}[f_1 \mathbf{U} f_2] = \mu Z. f_2 \vee (f_1 \vee \mathbf{EX} Z)$$

kann mit Hilfe der Funktion *Lfp* in einer endlichen Anzahl von Schritten berechnet werden. Die Herangehensweise für die Prozedur *CheckEG* ist der eben beschriebenen sehr ähnlich, nur dass diesmal die Funktion *Gfp* zur Berechnung des größten Fixpunktes zum Einsatz kommt. Wie oben bereits erwähnt ist die Fixpunktdarstellung des \mathbf{EG} Operators

$$\mathbf{EG} f_1 = \nu Z. f_1 \wedge \mathbf{EX} Z.$$

3 SMV

Im bisherigen Verlauf dieser Arbeit sind alle für die symbolische Modellüberprüfung notwendigen Formalismen, Methoden und Algorithmen beschrieben. Bislang wurde ein Modell $M = (S, R, L)$ zugrundegelegt und eine CTL Formel f , die zu überprüfen ist. Zur Spezifikation von Modellen dienen Kripke Strukturen, oder wahlweise logische Ausdrücke, die die Übergangsrelation beschreiben wie in Abschnitt 2.1.3. Allerdings wurde durch das dort gegebene Beispiel deutlich, dass sich auf diese Weise nur sehr kleine und einfache Modelle beherrschen lassen.

Um Modellüberprüfung auch bei komplexeren Modellen anwenden zu können, ist eine Beschreibungsmöglichkeit auf einer höheren, abstrakteren Ebene notwendig. Das SMV (Symbolic Model Verifier) System [6] bietet eine solche Definitionssprache für endliche Zustandsmaschinen und Spezifikationen in CTL. Aus dem ursprünglichen Tool als Ergebnis seiner Dissertationsarbeit entwickelte McMillan in den folgenden Jahren ein mächtiges Werkzeug für symbolische Modellüberprüfung [7], [8]. Die Vorstellung der Definitionssprache von SMV wird sich an der Syntax von extended SMV orientieren, während inhaltlich hauptsächlich der Sprachumfang des ursprünglichen SMV behandelt wird. Ein Ausblick auf einige fortgeschrittene Modellierungstechniken wird im Abschnitt 3.3 gegeben.

3.1 Überblick über SMV

3.1.1 Typen und Definition von Signalen

In SMV wird die Beschreibung des Systems und die Spezifikation in einem Programm zusammengefasst. Das Modell umfasst eine Menge von Zustandsvariablen (“Signale”), die einen von drei Basistypen haben können. Die Basistypen in SMV sind Boolean, Aufzählungstypen und endliche Integer-Intervalle, darauf werden alle komplexen oder zusammengesetzten Typen zurückgeführt.

```
request: boolean;           /* Ein logisches Signal */
state:   {ready, running}; /* Ein Aufzählungstyp */
index:   0..7;              /* Ein Integer Intervall */
```

Das Verhalten des Systems wird durch eine Menge von Zuweisungen bestimmt. Konzeptionell werden alle Zuweisungen parallel vorgenommen, so dass doppelte Zuweisungen an ein Signal und zirkuläre Abhängigkeiten von SMV als Fehler bemängelt werden. Für Eindeutigkeit sorgt die explizite Zuweisung des neuen Wertes an das Signal im nächsten Zustand mittels `next(<signal>)`. Der Ausgangszustand eines Signals kann über `init(<signal>)` festgelegt werden.

Es ist möglich, einem Signal einen Wert, der aus einer vorgegebenen Menge nichtdeterministisch ausgewählt wird, zuweisen zu lassen. Damit ist es möglich, Details der Implementierung des Systems zu verbergen, wie zum Beispiel bei einem laufenden Prozess, der entweder unterbrochen, terminiert oder weiterhin ausgeführt wird. Wann eine Unterbrechung stattfindet ist oft unerheblich, allein die Tatsache, dass der Prozess irgendwann unterbrochen wird, genügt.

```
init(request) = 0;           /* Angabe des Ausgangszustandes */
next(index) = index + 1     /* Einfache Zuweisung */
next(state) = {running, ready}; /* Nichtdeterministische Zuweisung */
```

3.1.2 Module

Um komplexe bzw. große Systeme beschreiben zu können, kann das Modell in Module aufgeteilt werden. Module können mehrfach verwendet (instanziiert) werden, was komponentenorientiertes Modellieren ermöglicht. Instanzen, die durch das Schlüsselwort `process` eingeleitet werden definieren eine Menge paralleler Prozesse.

```

MODULE main() {

gate1: process inverter(gate3.out);
gate2: process inverter(gate1.out);
gate3: process inverter(gate2.out);

liveness:
  assert G F gate1.running & G F gate2.running & G F gate3.running;

spec:
  assert ((G (F gate1.out)) & (G (F !gate1.out)));

using liveness prove spec;
assume liveness;
}

MODULE inverter(inp) {

INPUT inp : boolean;
out: boolean;

init(out) := 0;
next(out) := ~inp;
}

```

Abbildung 4: Modellierung eines Rings aus drei Invertern in extended SMV. Angelehnt an [6].

Ein Prozess wird zu einem gegebenen Zeitpunkt entweder ausgeführt oder nicht und zu einem Zeitpunkt wird immer genau ein Prozess ausgeführt. Die zeitlichen Intervalle zwischen zwei Ausführungen eines Prozesses sind beliebig. Zuweisungen an den nächsten Wert eines Signals innerhalb eines Prozesses werden nur dann durchgeführt, wenn der Prozess ausgeführt wird. Anwendungen für diese Technik sind zum Beispiel Modelle von Kommunikationsprotokollen, asynchronen Schaltungen oder auch parallelen Programmen.

Die Moduldefinition kann eine formale Parameterliste umfassen. Die Parameter übernehmen die Funktion von Eingängen in und Ausgängen aus dem Modul. Parameter sind global sichtbar, beim Zugriff erfolgt die Dereferenzierung durch einen Punkt.

Die Syntax zur Definition von Modulen wird aus dem in Abbildung 4 dargestellten Beispiel ersichtlich. Formale Parameter eines Moduls können mit den Schlüsselwörtern `input` bzw. `output` explizit als Eingabe- bzw. Ausgabeparameter definiert werden. Die Spezifikation eines Eingabeparameters impliziert eine Zuweisung vom aktuellen Parameter an den formalen Parameter. Umgekehrt bedeutet die Definition eines Ausgabeparameters eine Zuweisung vom formalen an den aktuellen Parameter.

3.1.3 Spezifikation

Die Spezifikation von Eigenschaften des Systems erfolgt durch die Angabe von **assertions**, die optional mit einem Namen versehen werden können. In extended SMV stehen die temporalen Operatoren **X**, **G**, **F** und **U** zur Verfügung wobei zu beachten ist, dass alle temporalen Formeln implizit universell quantifiziert werden. Mittels des **using ... prove ...**-Konstrukts können für den Beweis einer Eigenschaft andere Eigenschaften als Voraussetzung festgelegt werden. Nichtbeweisbare Eigenschaften wie zum Beispiel Fairness-Bedingungen können mittels **assume ...** als wahr angenommen werden.

3.2 Sprachkonstrukte von extended SMV

3.2.1 Komplexe Typen

Neben den in Abschnitt 3.1.1 vorgestellten Basistypen kennt extended SMV auch Arrays und strukturierte Typen. Diese beiden Typen stellen jedoch nur syntaktische Abkürzungen für äquivalente Definitionen aus Basistypen dar.

Die Syntax für eine Arraydeklaration sieht wie folgt aus:

```
<signal> : array <expr> .. <expr> [of <type>];
```

Da Arrayelemente jeden beliebigen Typ annehmen können, ist auch die Konstruktion mehrdimensionaler Arrays möglich. Wird die Typangabe weggelassen, handelt es sich um die Definition eines generischen Arrays, dessen Elemente beliebige Werte auch unterschiedlicher Typen zugewiesen werden können.

```
bitsLsb : array 7..0 of boolean;  
bitsMsb : array 0..7 of boolean;  
x : array 0..1 of array 0..1 of boolean;  
generic : array 0..5;
```

Wie aus den beiden ersten Zeilen ersichtlich ist, gibt es keine Bedingung, dass die untere Array Grenze kleiner sein muss als die obere Grenze. Das Array **x** ist zweidimensional mit zwei Elementen pro Dimension, während die letzte Definition ein Beispiel für die Definition eines generischen Arrays ist.

Strukturierte Typen werden in extended SMV durch Module abgebildet. In Abschnitt 3.1.2 wurde dies bereits angedeutet. Einen Sonderfall stellen Module ohne formale Parameter dar, die mittels des **struct**-Konstrukts abgekürzt definiert werden können. Intern wird das Programm vor der Verifikation "geglättet", d.h. jede Instanz eines Moduls oder Struktur wird ausgeschrieben. Demzufolge sind die beiden folgenden Definitionen äquivalent:

```
typedef hands struct {  
    left, right: boolean;  
}  
x : hands;  
  
y.left : boolean;  
y.right: boolean;
```

3.2.2 Bedingte Sprachkonstrukte

Das einfachste bedingte Sprachkonstrukt in extended SMV ist die `if`-Anweisung. Zusätzlich gibt es `default`-Blöcke und eine `case`- sowie eine `switch`-Anweisung. Die Syntax der `if`-Anweisung sieht wie folgt aus:

```
if (<condition>) <stmt1> else <stmt2>
```

`<stmt1>` bzw. `<stmt2>` können entweder einzelne Anweisungen sein oder eine Folge von Anweisungen, die durch geschweifte Klammern zusammengefasst sind. Durch bedingte Zuweisungen an Signale kann der Fall eintreten, dass einem Signal beispielsweise bei erfüllter Bedingung ein Wert zugewiesen wird, ansonsten aber bleibt das Signal undefiniert. Die Regel in extended SMV für diesen Fall ist es, dem Signal nichtdeterministisch einen Wert aus dem Wertebereich des Typs des Signals zuzuweisen.

Bleibt der nächste Wert eines Signals in einem Fall der bedingten Anweisung undefiniert, so wird als nächster Wert der aktuelle Wert angenommen. Benötigt werden diese sogenannten Default-Regeln (oder einfach nur Defaults), da der Zustand des Systems vom Wert aller Zustandsvariablen – also allen Signalen des Systems – bestimmt wird. Sollten die eingebauten Default-Regeln nicht ausreichen, so ist es mittels des `default...in...`-Konstrukts möglich, solche Regeln festzulegen.

```
default <stmt1> in <stmt2>
```

Die erste Anweisungs(-folge) `stmt1` legt die Default-Zuweisungen an Signale fest, denen in der zweiten Anweisungs(-folge) `<stmt2>` kein Wert zugewiesen wurde. Für den Fall, dass ein Signal in beiden Blöcken einen Wert zugewiesen bekommt, besitzt die Zuweisung aus dem `in` Block die höhere Priorität und wird ausgeführt. Es ist erlaubt, `default` Blöcke zu schachteln.

case und switch Anweisung Die `case` Anweisung ist eine abkürzende und übersichtlichere Schreibweise für eine Folge von `ifs`. Eine `case` Anweisung hat die Form

```
case {
  <cond_1> : <stmt_1>
  ...
  <cond_n> : <stmt_n>
  [default: <default_stmt>]
}
```

Eine Folge von n über `else` verbundene `if` Abfragen ist äquivalent zu obigem Code. Für den Fall, dass keine der Bedingungen zutrifft und auch die `default` Bedingung nicht angegeben wurde wird keine Anweisung ausgeführt.

Die `switch` Anweisung entspricht weitgehend dem aus C bekannten Sprachkonstrukt. Die Syntax der Anweisung sieht wie folgt aus:

```

switch(<expr>) {
  <cond_1> : <stmt_1>
  ...
  <cond_n> : <stmt_n>
  [default : <default_stmt>]
}

```

Die Semantik der `switch` Anweisung kann auf eine `case` Anweisung zurückgeführt werden, deren Bedingungen sich aus `<expr>` in `<cond_i>` ergeben. Statt einer einfachen Gleichheitsprüfung wird der Ausdruck `<expr>` auf Mengenzugehörigkeit in `<cond_i>` getestet. Damit sind auch mengenwertige Ausdrücke für die Bedingungen `<cond_i>` erlaubt, wie zum Beispiel 3..7.

3.2.3 Iterative Sprachkonstrukte

Extended SMV bietet auch zwei schleifenartige Sprachkonstrukte an, die allerdings wie alle komplexen Konstrukte zur Übersetzungszeit geglättet werden, was im Falle von Schleifen ein Ausrollen bedeutet. Die einfachere Schleife ist die `for` Schleife.

```

for(<var> = <init_expr>; <cond>; <var> = <next_expr>) <stmt>

```

Vor der ersten Iteration wird die Indexvariable `<var>` mit `<init_expr>` initialisiert. Solange die Bedingung `<cond>` erfüllt ist, wird die Anweisung(-folge) `<stmt>` ausgerollt, wobei jedes Vorkommen von `<var>` durch den aktuellen Wert ersetzt wird. Danach wird `<var>` auf den von `<next_expr>` bestimmten Wert gesetzt. Die Indexvariable `<var>` taucht nach dem Ausrollen der Schleife durch den Compiler nicht mehr im Programm auf und beeinflusst somit den Zustand des modellierten Systems nicht.

Neben der `for` Schleife gibt es noch die `chain` Schleife. Beim Ausrollen einer solchen Schleife werden geschachtelte `default` Blöcke erzeugt, so dass entweder die erste oder die letzte Iteration am höchsten priorisiert wird, abhängig von der Laufrichtung des Indexes. Die Syntax einer `chain` Schleife sieht wie folgt aus:

```

chain(<var> = <init_expr>; <cond>; <var> = <next_expr>) <stmt>

```

Es gelten die gleichen Regeln wie beim Ausrollen der `for` Schleife, nur dass die Anweisungs(-folge) `<stmt>` in `default` Blöcke verpackt wird. Somit sind die Anweisungen der vorigen Iteration Defaults für die Anweisungen der Aktuellen.

3.3 Fortgeschrittene Modellierungstechniken

In diesem Abschnitt soll noch kurz auf einige der fortgeschrittenen Modellierungstechniken von extended SMV eingegangen werden. Wie bereits erwähnt ermöglichen diese Techniken die Handhabung komplexerer Systeme, allerdings bedeutet die Verwendung von komplizierten programmiersprachlichen Konzepten auch eine erhöhte Fehlerwahrscheinlichkeit in den damit erstellten Programmen. Die Beispiele und Anwendungsmöglichkeiten der vorgestellten Modellierungstechniken ist [7] entnommen.

Systeme können in extended SMV durch die Einführung von Schichten in verschiedenen Detaillierungsgraden beschrieben werden. Eine höhere Schicht übernimmt dabei die Rolle der Spezifikation für das Verhalten einer tieferen Schicht, was gleichbedeutend damit ist, dass die tiefere Schicht die Implementation dieser Spezifikation übernimmt. Beispielsweise kann die Modellierung eines Switch oder eines ähnlichen Gerätes auf der Ebene von Taktzyklen gegeben sein und es soll verifiziert werden, ob das Gerät der abstrakten Beschreibung des Ende zu Ende Datenverkehrs genügt. Diese abstrakte Beschreibung, die durch `layer` Blöcke angegeben wird, kann so einfach sein, dass sie lediglich fordert, dass ankommende Datenpakete den Ausgehenden entsprechen.

Neben dem Schichtenkonzept kann eine weitere Vereinfachung von Modellen durch das Ausnutzen von Symmetrien erfolgen. Betrachte man als Beispiel wieder den Switch, der unter Verwendung einer bestimmten Komponente jedes Bit vom Eingang auf einen Ausgang schaltet. Bei einer Wortbreite von n Bits werden n dieser Komponenten verwendet, allerdings ist es für die Korrektheit des gesamten Systems ausreichend, diese Komponente nur einmal zu verifizieren. Diese Art von Symmetrie kann über Arrays mit dem speziellen Indextyp `scalarset` in extended SMV ausgedrückt werden. Bei der Verifikation eines solchen Arrays wählt SMV exemplarisch ein Element des Arrays aus, benutzt es für die Verifikation und schließt davon auf die restlichen Arrayelemente.

Allerdings können Symmetrieeigenschaften noch weitgehender ausgenutzt werden. Die Idee orientiert sich dabei an Induktionsbeweisen: Eine Eigenschaft wird für einen Basisfall und für jeden Nachfolger bewiesen. Für diese Art von Beweisen bietet extended SMV den speziellen Indextyp `ordset` für Arrays an. Durch diesen Datentyp ist es möglich unendliche Familien endlicher Zustandssysteme zu verifizieren. Auf diese Weise kann zum Beispiel eine ALU mit beliebiger Wortbreite von SMV verifiziert werden.

3.4 Beispiel: Gegenseitiger Ausschluss

Das etwas umfangreichere Beispiel in Abbildung 5 auf Seite 17 illustriert die Verwendung einiger der vorgestellten komplexeren Sprachkonstrukte. Allerdings ist die Lösung des gestellten Problems sehr simpel gehalten: Signalisieren beide Prozesse den Wunsch, in die kritische Region eintreten zu wollen, d.h. ist der Zustand der beiden Prozesse `trying`, dann darf der als nächstes ausgeführte Prozess eintreten.

4 Schlussbemerkung

Mit der Entwicklung von SMV im Rahmen seiner Dissertation hat McMillan sicherlich für einen Durchbruch der symbolischen Modellüberprüfung gesorgt, vor allem auch weil das System OBDDs souverän einsetzt. Mit der Verfügbarkeit eines Tools und damit einer geeigneten Definitionssprache für Modelle und Spezifikationen war es dann möglich, eine Art Proof of Concept der symbolischen Modellüberprüfung durchzuführen, was mit der Verifikation des FutureBus+ eindrucksvoll gelungen ist.

```

typedef stateSet {noncritical, trying, critical};

MODULE main() {
  proc: array 0..1;

  for(i=0; i<2; i = i+1)
    proc[i] : process proc(proc[(i+1) mod 2].otherState,
                          proc[(i+1) mod 2].state);

  fairness:  assert G F proc[0].running & G F proc[1].running;
  proc0_fair: assert G F ~(proc[0].state = critical);
  proc1_fair: assert G F ~(proc[1].state = critical);

  mutex_violation: assert G F ~((proc[0].state = critical) &
                                (proc[1].state = critical));

  using fairness, proc0_fair, proc1_fair
    prove mutex_violation;

  assume fairness, proc0_fair, proc1_fair;
}

MODULE proc(state, otherState) {
  output state:      stateSet;
  input  otherState: stateSet;

  init(state) := noncritical;
  case {
    (state = noncritical) :
      next(state) := {trying, noncritical};
    (state = trying) & (otherState = noncritical) :
      next(state) := critical;
    (state = trying) & (otherState = trying) :
      next(state) := critical;
    (state = critical) :
      next(state) := {critical, noncritical};
  default:
    next(state) := state;
  };
}

```

Abbildung 5: SMV Code für gegenseitigen Ausschluss mit zwei Prozessen. (Siehe [5].)

Folgerichtig wurde bei Cadence mit “FormalCheck” ein kommerzielles Produkt für symbolische Modellüberprüfung unter Mitarbeit von Kenneth McMillan und vermutlich basierend auf extended SMV entwickelt. Die Verfügbarkeit eines solchen Tools ist als Indiz für den industriellen Einsatz von Modellüberprüfung zu werten.

Allerdings deutet die Vielzahl der in der Literatur zu findenden Beispiele an SMV Programmen darauf hin, dass das Hauptanwendungsgebiet eher im Hardwarebereich zu suchen ist. Das Problem bei der Modellierung paralleler Programme beginnt bei der Wahl einer geeigneten Abstraktion und allen Modellen gemeinsam ist die große Zahl der Zustände, was zu einer ineffizienten Verifikation führt.

Literatur

- [1] Akers, Sheldon B. *Binary Decision Diagrams*. IEEE Transactions on computers, C-27(6): 509-516, Juni 1978.
- [2] Bryant, Randal E. *Graph-Based Algorithms for Boolean Function Manipulation*. IEEE Transactions on Computers, C-35(8): 677-691, August 1986.
- [3] Burch, J. R., Clarke, Edmund M., McMillan, Kenneth L., Dill, D. L. und Hwang, L. J. *Symbolic Model Checking: 10²⁰ States and Beyond*. Information and Computation 98(2): 142-170, Juni 1992.
- [4] Clarke, Edmund M., Grumberg, Orna, Hiraishi, H., Jha, S., Long, D. E., McMillan, Kenneth M. und Ness, L. A. *Verification of the FutureBus+ cache coherence protocol*. In: *Proceedings of the 11th International Symposium on Computing Hardware Description Languages and their Applications*. Hrsg. Claesen, L. North Holland, 1993.
- [5] Clarke, Edmund M., Grumberg, Orna und Peled, Doron A. *Model Checking*. The MIT Press, Cambridge, MA, USA 1999.
- [6] McMillan, Kenneth L. *Symbolic Model Checking*. Kluwer Academic Press, Boston, MA, USA 1993.
- [7] McMillan, Kenneth L. *Getting started with SMV*. Fassung vom 23. März 1999. Quelle: <http://www-cad.eecs.berkeley.edu/~kenmcmil/tutorial.ps> am 29.11.2001.
- [8] McMillan, Kenneth L. *The SMV language*. Fassung vom 23. März 1999. Quelle: <http://www-cad.eecs.berkeley.edu/~kenmcmil/language.ps> am 29.11.2001.