Projektarbeit

Evaluierung von Algorithmen zur Berechnung fairer Pfade

Christoph Wagner
September 2005

Betreuer:
Prof. Dr. rer. nat. Klaus Schneider
Dipl. - Inf. Tobias Schüle
Arbeitsgruppe Reaktive Systeme
Fachbereich Informatik
Universität Kaiserslautern

Inhaltsverzeichnis

1 Einführung

Wie in vielen Bereichen gilt es bei der Entwicklung reaktiver Systeme Fehler zu vermeiden. Allen Gebieten gemeinsam ist dabei die Reduktion der Kosten, die durch spät entdeckte Fehler entstehen. Bei reaktiven Systemen kommt hinzu, dass sie in sicherheitskritischen Bereichen eingesetzt werden und durch Fehler einerseits schwere finanzielle Schäden entstehen, andererseits Personen körperlich zu Schaden kommen können.
Daher sind Methoden nötig, die in der Lage sind, die Fehlerfreiheit von Systemen zu gewährleisten. Im Lauf der Zeit hat sich die symbolische Modellprüfung zum Hauptwerkzeug entwickelt, um Systeme zu verifizieren. Dabei werden die zu prüfenden Systeme normalerweise durch Kripkestrukturen dargestellt, während die Spezifikation, also die einzuhaltenden Bedingungen und Eigenschaften, meist in Temporallogiken formuliert und vom Modellprüfer in μ-Kalkül Formeln übersetzt werden. Die Modellprüfung hat dann zu entscheiden, ob die Initialzustände der Kripkestruktur die Spezifikation erfüllen.
Wenn ein System eine gewünschte Eigenschaft nicht erfüllt, sollte das Modellprüfungswerkzeug Gegenbeispiele generieren anhand derer erkannt werden kann, warum das System die Spezifikation nicht erfüllt. Die Berechnung von Gegenbeispielen für Spezifikationen in LTL ist zwar immer möglich, doch im praktischen Einsatz hat sich gezeigt, dass der Berechnungsaufwand sehr groß ist. Während die generelle Vorgehensweise dabei unverändert blieb, gab es für den Teilschritt der Zerlegung der Kripkestruktur in stark zusammenhängende Komponenten[1] verbesserte Algorithmen.
Diese Arbeit stellt zunächst das Vorgehen zur Generierung von Gegenbeispielen vor und vergleicht dann die Algorithmen zur Zerlegung der Kripkestrukturen anhand ihrer an Beispielen gemessenen Laufzeiten.

[1] Der gebräuchliche englische Ausdruck lautet Strongly Connected Component, SCC.

2 Voraussetzungen

In diesem Abschnitt wird auf die Themen eingegangen, die zum späteren Verständnis der Algorithmen nötig sind, welche zum Berechnen der Gegenbeispiele verwendet werden. Zunächst werden Binäre Entscheidungsdiagramme[2] als die zu Grunde gelegten Datenstrukturen vorgestellt und die Methoden zu ihrer Manipulation kurz erläutert. Danach werden Kripkestrukturen vorgestellt, durch die das zu überprüfende System modelliert wird. Auf die Vorgänger- bzw. Nachfolgerberechnung von Zuständen einer Kripkestruktur wird dabei als zeitkritischster Schritt im besonderen eingegangen. Es wird gezeigt, wie eine Spezifikation in der Temporallogik LTL erfolgt, und wie ein Gegenbeispiel einer nicht erfüllten Spezifikation aussieht. Abschließend wird auf das Value-Change-Dump-Format eingegangen, in dem die Gegenbeispiele ausgegeben werden können.

2.1 Binäre Entscheidungsdiagramme

Binäre Entscheidungsdiagramme gehen zurück auf eine Arbeit von Bryant [2] und sind eine Form der Repräsentation aussagenlogischer Formeln. Sie basieren darauf, dass der if-then-else Operator eine Basis bildet, d.h. sich jede aussagenlogische Formel ausschließlich durch if-then-else darstellen läßt.

$$\begin{aligned}\neg\varphi &\equiv (\varphi \Rightarrow 0 \mid 1)\\ \varphi \wedge \psi &\equiv (\varphi \Rightarrow \psi \mid 0)\\ \varphi \vee \psi &\equiv (\varphi \Rightarrow 1 \mid \psi)\end{aligned}$$

Ein Entscheidungsdiagramm ergibt sich indem in einer Formel jedes Auftreten eines Operators durch if-then-else ersetzt wird. Werden die in der Formel vorhandenen Variablen geordnet und gleiche Teilbäume nur einmal gespeichert, so bilden die BDDs eine kanonische Normalform.

Definition 1 (Geordnete BDDs). *Seien die Variablen $\mathcal{V} = \{ x_1, x_2, \ldots, x_n \}$ mit einer bijektiven Funktion $\pi : \mathcal{V} \rightarrow \{ 2, \ldots, |\mathcal{V}|+1 \}$ gegeben.*
Sei degree : $\mathcal{V} \cup \mathbb{N} \rightarrow \mathbb{N}$

$$\textit{degree}(v) := \begin{cases} \pi(v) : \textit{falls } v \in \mathcal{V} \\ n : \textit{falls } n \in \mathbb{N} \end{cases}$$

so gilt mit
$x \prec_\pi y :\Leftrightarrow$ *degree(x) < degree(y)*

[2] Binary Decision Diagram, BDD

dass

$0 \prec_\pi 1 \prec_\pi x_1 \prec_\pi \ldots$

Ein geordnetes binäres Entscheidungsdiagramm (OBDD) über den Variablen $\mathcal{V}$ bezüglich der Ordnung $\prec_\pi$ ist ein Tupel G = ($\mathcal{V} \cup \{L_0, L_1\}$, v_0 , label, high, low) mit

$\forall\, v \in \mathcal{V}$.*label(*$v$*)* $\in \mathcal{V}$

- *label(*L_0*)* = 0 *und label(*L_1*)* = 1
- *high :* $\mathcal{V} \rightarrow \mathcal{V} \cup \{\ L_0, L_1\ \}$ *und*
- *low :* $\mathcal{V} \rightarrow \mathcal{V} \cup \{\ L_0, L_1\ \}$ *mit*
  - *degree(label(high(*v*))) < degree(label(*v*))*
  - *degree(label(low(*v*))) < degree(label(*v*))*
- $\exists\, v_0 \in \mathcal{V}.\ \forall\, v \in \mathcal{V}$.*high(*$v$*)* $\neq v_0 \wedge$ *low(*v*)* $\neq v_0$

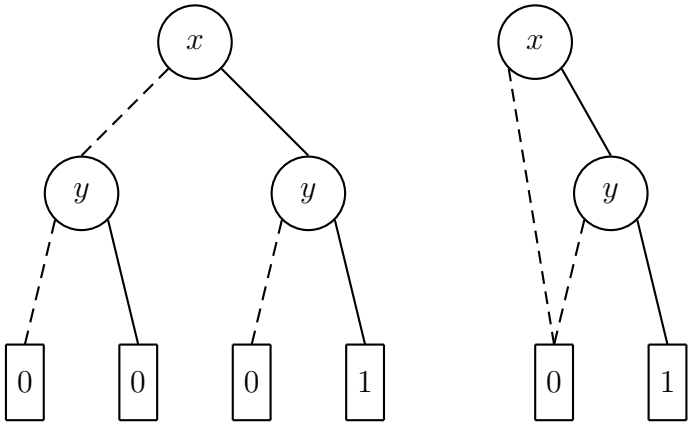

Abbildung 1: BDDs zu $x \wedge y$

Es gibt einen Wurzelknoten, v_0, von dem aus sich das Entscheidungsdiagramm aufbaut. Die Auswertung eines OBDD erfolgt indem man entsprechend der Belegung des Wurzelknotens der High bzw. Low Kante folgt, und dieses Vorgehen fortsetzt bis man an einem Blatt angekommen ist. Das Label des Blatts gibt dann die Gültigkeit des OBDDs für die Belegung an. Beide OBDDs aus Abbildung 1 beschreiben die Formel $x \wedge y$, wobei die High - Kante durch eine durchgezogene

und die Low - Kante durch eine gestrichelte Linie dargestellt sind.
Nun müssen noch gleiche Teilbäume entfernt werden.

Definition 2 (Isomorphe OBDDs). *Zwei OBDDs $\mathcal{B}_1$ und $\mathcal{B}_2$ heißen isomorph, wenn es eine bijektive Abbildung $\Theta : \mathcal{V}_1 \rightarrow \mathcal{V}_2$ gibt, so dass für jeden Knoten gilt:*
falls v ein Blatt ist, so ist auch $\Theta(v)$ ein Blatt und trägt dieselbe Markierung,
falls v kein Blatt ist, so gilt
- label(v) = label($\Theta(v)$)
- (Θ(high(v)) = high($\Theta(v)$)
- (Θ(low(v)) = low($\Theta(v)$)

Definition 3 (Reduzierte OBDDs). *Ein OBDD ist reduziert (ein ROBDD), wenn es*
- keinen Knoten v gibt mit high(v) = low(v)
- kein Paar von Knoten v_1, v_2 gibt, so dass die in v_1 und v_2 wurzelnden Untergraphen isomorph sind.

In Abbildung 1 ist zu dem OBDD zu $x \wedge y$ die reduzierte Form angegeben. In der Praxis werden nur reduzierte OBDDs eingesetzt, da sie einen geringeren Speicherplatzbedarf aufweisen. Eine weitere nützliche Eigenschaft betrifft die Überprüfung auf Gleichheit von Formeln. Da äquivalente Formeln nur einmal gespeichert werden, reduziert sich das Problem auf einen simplen Zeigervergleich.
Des Weiteren lassen sich die booleschen Operationen effizient auf BDDs ausführen. Auch Quantifizierung ist möglich, welche später für die Nachfolger- und Vorgängerberechnung benötigt wird. Ferner existieren Heuristiken für Variablenordnungen π, um die Größe der BDDs zu minimieren.

2.2 Kripkestrukturen

Kripkestrukturen sind das Mittel der Wahl, um Systeme für die Modellprüfung zu beschreiben. Sie ähneln endlichen Automaten und weisen wie diese Zustände und Transitionen auf. Die Zustände entsprechen den möglichen Variablenbelegungen des Systems und die Transitionen beschreiben die möglichen Zustandsübergänge. Eine Kripkestruktur beschreibt somit alle möglichen Berechnungsfolgen des Systems, wobei den Transitionen jedoch keine Bedingungen zu Grunde liegen.

Definition 4 (Kripkestruktur). *Eine Kripkestruktur ist ein Tupel $\mathcal{K}$ = ($\mathcal{I}$, $\mathcal{S}$, $\mathcal{R}$, $\mathcal{L}$), mit*

- *$\mathcal{S}$ einer endlichen Menge von Zuständen*
- *$\mathcal{I} \subseteq \mathcal{S}$ einer Menge initialer Zustände*
- *$\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S}$ einer Transitionsrelation*
- *$\mathcal{L} : \mathcal{S} \rightarrow 2^{\mathcal{V}}$ einer Markierungsfunktion, wobei $\mathcal{L}(s)$ der Menge der in s gültigen Variablen entspricht.*

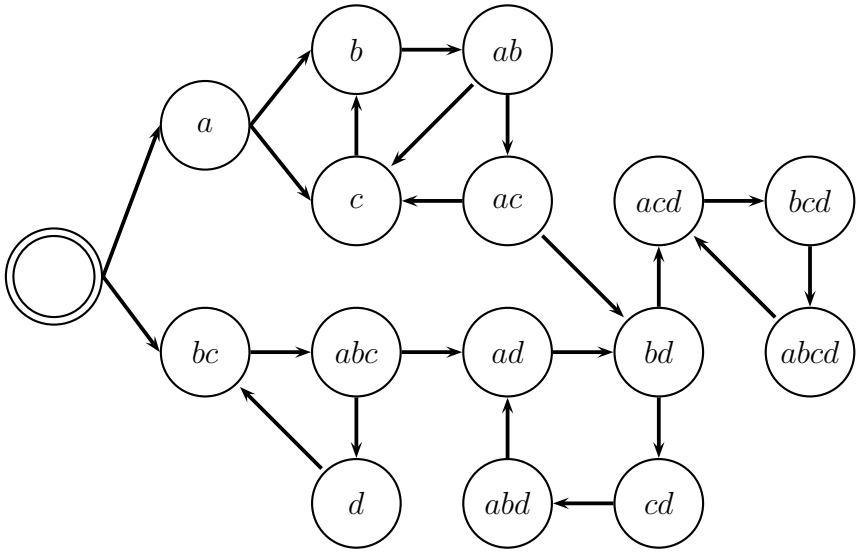

Abbildung 2: Beispiel einer Kripkestruktur mit 4 Variablen

Nutzt man die Möglichkeit, die Zustände der Kripkestruktur direkt durch die Variablen zu kodieren, so gilt $s = \mathcal{L}(s)$ und $\mathcal{S} = 2^{\mathcal{V}}$. Die Initialzustände können so direkt durch eine aussagenlogische Formel dargestellt werden. Nimmt man zu jeder Variable x_i noch x_i' hinzu um die Gültigkeit von x_i im Folgezustand anzugeben, so läßt sich auch die Transitionsrelation aussagen-logisch darstellen. Es ergibt sich so eine symbolische Darstellung der Kripkestruktur $\mathcal{K} = (\mathcal{V}, \mathcal{I}, \mathcal{R})$, für die als Datenstrukturen BDDs dienen. In Abbildung 2 ist eine beispielhafte Kripkestruktur mit 4 Variablen angegeben. Zur Einfachheit ist auch hier ein Zustand s mit der Menge der in ihm gültigen Variablen markiert. Der Initialzustand ist durch doppelt gezeichnete Kreise hervorgehoben.
Für die Berechnung von Gegenbeispielen ist die Bestimmung von Nachfolger- bzw. Vorgängermengen eines Zustandes nötig. Der Algorithmus zur Bestimmung dieser Mengen hat im Worst Case exponentielle Laufzeit in der Größe des BDDs.

Definition 5 (Vorgänger- und Nachfolgermengen). *Beschreibt die aussagenlogische Formel Φ_Q die Zustandsmenge Q, so beschreibt*

$$pre^R_\exists(Q) = \exists x'_1, \ldots, x'_n . \mathcal{R}(x_1, \ldots, x_n, x'_1, \ldots, x'_n) \wedge [\Phi_Q]^{x'_1,\ldots,x'_n}_{x_1,\ldots,x_n}$$

die Menge der Zustände, von denen aus in einem Schritt die Menge Q erreichbar ist, also die Vorgängermenge, und

$$suc^R_\exists(Q) = [\exists x_1, \ldots, x_n . \mathcal{R}(x_1, \ldots, x_n, x'_1, \ldots, x'_n) \wedge \Phi_Q]^{x_1,\ldots,x_n}_{x'_1,\ldots,x'_n}$$

entsprechend die Nachfolgermenge, die Zustandsmenge, die in einem Schritt von Q aus erreichbar ist.
Hierbei entspricht $[\Phi]^{x'}_x$ der Formel Φ, in der jedes Vorkommen von x durch x' ersetzt wird.

In der Modellprüfung wird die Vorgänger- bzw. Nachfolgerberechung als symbolischer Schritt bezeichnet. Obwohl die Menge der symbolischen Schritte eines Algorithmus nicht direkt mit dessen Laufzeit zusammenhängt liefert sie dennoch einen guten Ansatz, um die Komplexität des Algorithmus zu bestimmen.

2.3 LTL und Gegenbeispiele

Obwohl in der Modellprüfung Spezifikationen im Allgemeinen durch μ-Kalkül Formeln verifiziert werden eignet sich der μ-Kalkül nicht besonders um Spezifikationen für den Anwender gut lesbar darzustellen. Für diese Aufgabe werden Temporallogiken verwendet. Eine gute Übersicht über die Methoden der Modellprüfung ist in [4] und [6] gegeben. Es gibt drei Klassen von Temporallogiken.
CTL erlaubt Aussagen über Zustandsmengen,
LTL erlaubt Aussagen über Pfade und
CTL^* erlaubt uneingeschränkt Aussagen über Pfade und Zustände.

Definition 6 (Syntax LTL). *LTL kennt Zustandsformeln: $S ::= AP$*
und Pfadformeln: $P ::= V_\Sigma \mid \neg P \mid P \wedge P \mid XP \mid [P\ U\ P] \mid \underline{\overleftarrow{X}} P \mid [P\ \underline{\overleftarrow{U}}\ P]$

Xφ ist auf einem Pfad an der Stelle t gültig wenn an der Stelle t+1 φ gilt. φ U ψ ist auf einem Pfad an der Stelle t gültig wenn es ein u $\geq$ t gibt so dass an den Stellen x mit t $\leq$ x < u φ gilt, und wenn x = u ist ψ gilt. $\underline{\overleftarrow{X}}$ und $\underline{\overleftarrow{U}}$ sind die entsprechenden rückwärtsgerichteten Operatoren zu X und U.
Zur Modellprüfung wird zu einer gegebenen Kripkestruktur mit LTL-Spezifikation

ein ω-Automat derart bestimmt, dass der Automat bei erfüllter Spezifikation φ keinen Durchlauf akzeptiert. Dazu wird bei der Übersetzung die negierte Spezifikation $\neg\varphi$ verwendet. Dafür wird die stärkste Automatenklasse, die der nichtdeterministischen Büchi-Automaten, benötigt. Ein akzeptierender Durchlauf wäre in diesem Falle ein Pfad, auf dem unendlich oft eine gewisse Eigenschaft gültig ist. Dies wird als Fairness Eigenschaft bezeichnet, der akzeptierende Pfad ist somit ein fairer Pfad. Ein bei nicht erfüllter Spezifikation auftretender fairer Pfad entspricht somit einem Gegenbeispiel zur Spezifikation.
Faire Pfade haben einen festgelegten Aufbau. Ausgehend von einem Initalzustand geht ein Pfad, das so genannte Präfix, in einen Zyklus über auf dem dann unendlich oft die Fairness Bedingungen der Spezifikation gültig sind.
Als Beispiel betrachten wir die Kripkestruktur aus Abbildung 2. Seien dazu $F = \{cd, ad\}$ die einzuhaltenden Fairness-Bedingungen, dann ist der zugehörige faire Pfad in Abbildung 3 gegeben.

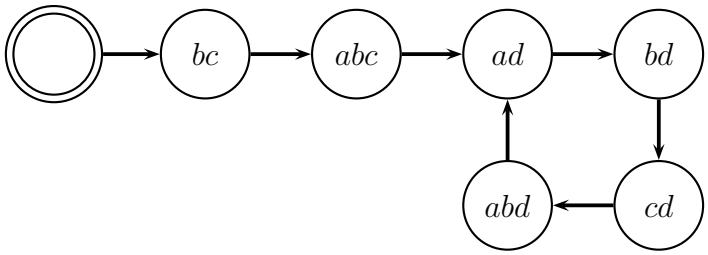

Abbildung 3: Beispiel eines Gegenbeispiels

2.4 Value Change Dump

Im IEEE Verilog Standard aus dem Jahr 2001 wurde das Value Change Dump Dateiformat beschrieben [9]. Im Rahmen dieser Arbeit wurde das VCD Format gewählt um die berechneten fairen Pfade auszugeben.
Das VCD Format bietet die Möglichkeit Variablen zu definieren und über einen bestimmten Zeitraum hinweg ihre Belegungen in einer Datei zu speichern. Mit einem entsprechenden VCD-Viewer wie zum Beispiel Dinotrace[3] kann dieses Zeit-

[3] Zu beziehen unter http://www.veripool.com/dinotrace/

liche Verhalten dann graphisch ausgegeben werden. Eine VCD Datei besteht aus einem Header in dem Informationen über die erzeugende Anwendung gespeichert werden und die in der Datei verwendeten Variablen mit ihren Typen deklariert werden. Als Typen stehen zum Beispiel integer, real, event und reg zur Verfügung. Da wir nur mit Boolschen Werten arbeiten werden alle Variablen als einbittige Register deklariert.
Darauf folgen dann die Initialbelegungen der Variablen und ihre Veränderungen, wobei nicht für jeden Simulationsschritt, oder in unserem Fall erreichten Zustand, alle Variablenbelegungen ausgegeben werden. Es genügt die Variablen zu schreiben die sich seit dem vorherigen Schritt verändert haben. Abbildung 4 zeigt das Programm Dinotrace bei der Ausgabe des Fairen Pfades aus dem Beispiel in Abbildung 3.
Als zusätzliche Ausgabe ist in die Variable $loop$ vorhanden, die auf dem Präfix $false$ ist und ihren Wert zu $true$ ändert sobald der Zyklus erreicht wird. Die Deklaration der Variablen $loop$ geschieht dabei wie folgt:
$var reg 1 *loop loop $end
Die Deklaration ist von $var und $end umschlossen. Der Typ der Variablen wird darin als erstes angegeben. Hier werden lediglich Register verwendet, welche zusammen mit ihrer Größe angegeben werden. Darauf folgt ein Datei-interner Name der Variable und der eigentliche Name der Variablen, wie er in der Ausgabe dargestellt werden sollte. Im Rumpf werden die Simulationsschritte $\#0 \ldots \#$n gefolgt von den Variablenzuweisungen aufgeführt. Variablenzuweisungen haben die Form $Wert$*$Variable$. Wenn zum Beispiel der Zyklus erreicht wird und $loop$ auf $true$ wechselt lautet die zugehörige Zuweisung:
1*loop

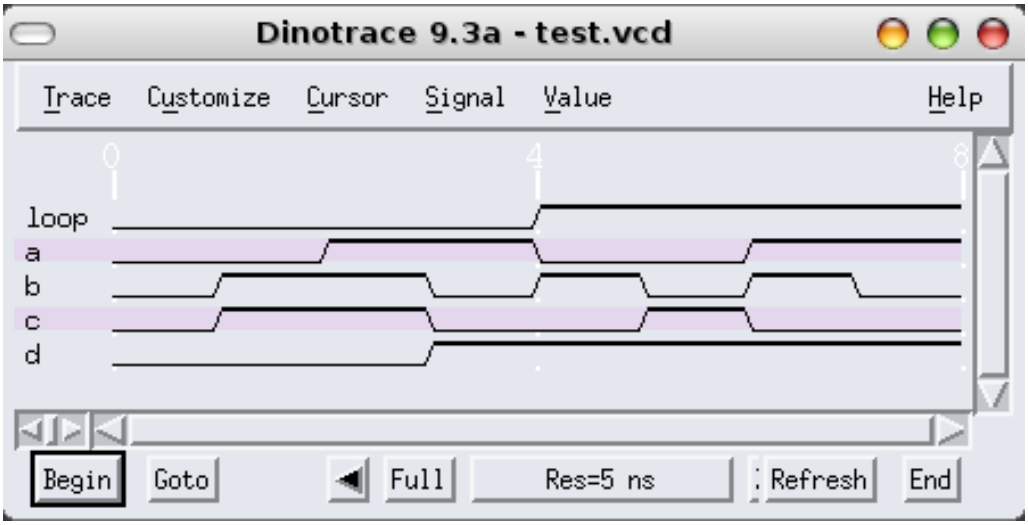

Abbildung 4: Dinotrace Ausgabe des fairen Pfades

3 Automatische Berechnung fairer Pfade

In diesem Abschnitt werden nun das Vorgehen bei der automatischen Berechnung fairer Pfade und die dazu benutzten Algorithmen vorgestellt. Die Methode geht zurück auf [3], wobei dort auch Gegenbeispiele für CTL und CTL^* besprochen werden.
Zunächst wird die Kripkestruktur in ihre stark zusammenhängende Komponenten zerlegt.

Definition 7 (Strongly Connected Component). *Sei $\mathcal{K}$ = ($\mathcal{I}$, $\mathcal{S}$, $\mathcal{R}$, $\mathcal{L}$). Wir schreiben s $\rightsquigarrow_{\mathcal{K}}$ s', wenn es einen endlichen Pfad in $\mathcal{K}$ gibt, der vom Zustand s zum Zustand s' führt, oder wenn s = s' gilt. $\rightsquigarrow_{\mathcal{K}}$ ist eine Quasiordnung und wir definieren die dazu gehörige Äquivalenzrelation $\approx_{\mathcal{K}}$ als:*

$$s \approx_{\mathcal{K}} s' :\Leftrightarrow s \rightsquigarrow_{\mathcal{K}} s' \wedge s' \rightsquigarrow_{\mathcal{K}} s$$

Die Äquivalenzklassen zu s $\approx_{\mathcal{K}}$ s' bezeichnen wir als SCC. Die Äquivalenzklasse zu s bezeichnen wir als SCC(s).

Offensichtlich ist eine SCC eine Menge von Zuständen mit der Eigenschaft, dass ein beliebiger Zustand jeden anderen aus dieser Menge erreichen kann. Für die Berechnung von Gegenbeispielen liegt der Nutzen darin, dass der Zyklus des fairen Pfades in einer SCC liegen muss. Algorithmen zur Zerlegung eines Graphen in SCCs existieren schon seit längerem, doch bei den klassischen Tiefensuchenalgorithmen, wie dem von Tarjan [8], müssen alle Zustände explizit aufgezählt werden, was bei Kripkestrukturen mit sehr großen Zustandsräumen unmöglich wird. Trotz der guten Laufzeit von $O(|\mathcal{K}|)$ können explizite Algorithmen nicht von der symbolischen Darstellung profitieren und sind somit für die symbolische Modellprüfung ungeeignet. Xie und Beerel stellten 1999 einen symbolischen Algorithmus vor der eine Kripkestruktur mit $O(|\mathcal{S}|^2)$ symbolischen Schritten und einer Laufzeit von $O(|\mathcal{K}||\mathcal{S}|^2)$ zerlegt. 2000 wurde der Algorithmus von Bloem, Gabow und Somenzi verbessert, sodass er mit $O(|\mathcal{S}| \log_2 |\mathcal{S}|)$ symbolischen Schritten arbeitet. Schließlich wurde 2003 von Gentilini, Piazza und Policriti ein Algorithmus vorgestellt der $O(|\mathcal{S}|)$ Schritte benötigt.
Diese Algorithmen werden in den folgenden Abschnitten genauer vorgestellt.
Hat man mit einem dieser Algorithmen also die SCCs bestimmt, überprüft man welche davon die Fairness Bedingungen der Spezifikation erfüllen.

Definition 8 (Faire SCCs und faire Zyklus). *Sei $\mathcal{F} = \{F_1, \ldots F_n\}$ eine Menge von Fairness Bedingungen.*

Die SCC SCC(s) ist fair bezüglich $\mathcal{F}$ wenn SCC(s) $\cap F_i \neq \{\}$ für alle $F_i \in \mathcal{F}$ und wenn SCC(s) eine einelementige Menge ist zusätzlich (s,s) $\in \mathcal{R}$ gilt.
Ein Tupel von Zuständen (s_0, ..., s_{m-1}) ist ein Zyklus in $\mathcal{K}$ wenn (s_i, $s_{(i+1) \bmod m}$) $\in \mathcal{R}$ für alle $i \in \{0, \ldots\ m-1\}$. Ein Zyklus in $\mathcal{K}$ ist fair bezüglich $\mathcal{F}$ wenn $\{s_0$, ..., $s_{m-1}\} \cap F_i \neq \{\}$ für alle $F_i \in \mathcal{F}$.

Der Algorithmus zum Überprüfen einer SCC auf Vorhandensein eines fairen Zyklus geht sehr geradlinig vor und prüft nacheinander das Enthaltensein der einzelnen Fairness Bedingungen ab. Der Algorithmus kann in Abbildung 5 eingesehen werden.

```
function HasFairCycle (C, F)
        fair := true;
        while (F ≠ {}) ∧ fair do
                F := selectfrom(F);
                F := F \ {F};
                fair := fair ∧ (F ∩ C ≠ {});
        end;
        if (|C| = 1) then
                {s} := C;
                fair := fair ∧ ( (s,s) ∈ R );
        end;
        return fair;
end
```

Abbildung 5: Überprüfung einer SCC auf Fairness

Sind die fairen SCCs bestimmt müssen im nächsten Schritt die erreichbaren fairen SCCs gefunden werden. Der Algorithmus aus Abbildung 6 tut dies, indem zu einer SCC schrittweise die Vorgängerzustände bestimmt werden und geprüft wird ob dabei ein Initialzustand erreicht wird. Die so berechneten Zwiebelringe der Vorgängermengen werden zurückgegeben um später den fairen Pfad zu bestimmen.
Nachdem die Menge der erreichbaren fairen SCCs bekannt ist können daraus die fairen Pfade berechnet werden. Wiederum ist das Vorgehen dabei sehr geradlinig. Ausgehend vom gefundenen Initialzustand werden dessen Nachfolger bestimmt und diese mit dem hinter dem Initialzustand liegenden Zwiebelring geschnitten. Aus der Schnittmenge wird ein beliebiger Zustand ausgewählt und entsprechend

```
function Reachable (C)
    reachable := false;
    stack fronts;
    if (C ∩ I ≠ {}) then
        push (fronts,C ∩ I);
        return (true,fronts);
    end;
    front := pre(C) \ C;
    while (¬reachable ∧ front ≠ {}) do
        if (front ∩ I = {}) then
            push (fronts,front);
            reachable := true;
        else;
            push (fronts,front ∩ I);
            front := pre(front) \ front;
        end;
    end;
    return (reachable,fronts);
end
```

Abbildung 6: Überprüfung einer SCC auf Erreichbarkeit

mit dessen Nachfolgermenge so weiter verfahren, bis die SCC erreicht ist. In der SCC muss sodann noch ein fairer Zyklus gefunden werden. Dafür wird ein ähnliches Verfahren verwendet. Innerhalb der SCC werden solange Nachfolger bestimmt, bis alle Fairness-Bedingungen mindestens einmal gültig waren und wieder der erste Zustand in der SCC erreicht wurde. Abbildung 7 zeigt das Vorgehen.

3.1 XieBeerel

Der von Xie und Beerel vorgeschlagene Algorithmus [11] beruht darauf, dass der zu s gehörende SCC dem Schnitt der Vorgänger- und Nachfolgermengen von s entspricht und nutzt dazu ihre SCC Geschlossenheit.

Definition 9 (SCC-Geschlossene Menge). *Eine Menge $U \subseteq S$ ist eine SCC geschlossene Menge wenn keine SCC sowohl U als auch das Komplement $S \backslash U$ schneidet.*

```
function FairPath (C, fronts , F)
    stack path;
    state := pop(fronts);
    push(path, state);
    while (¬ empty(fronts)) do
        back := suc(state);
        onionring := pop(fronts);
        back := back ∩ onionring;
        state := selectfrom(back);
        beginofloop := state;
        push(path, state);
    end;
    state := selectfrom(suc(state) ∩ C);
    push(path, state);
    while (F ≠ {}) do
        F := F \ (F ∩ state);
        state := selectfrom(suc(state) ∩ C);
        push(path, state);
    end;
    while (state ≠ beginofloop) do
        state := selectfrom(suc(state) ∩ C);
        push(path, state);
    end;
    return path;
end
```

Abbildung 7: Berechnung eines fairen Pfades

Lemma 1 (SCC-Geschlossenheit der Vorgänger und Nachfolger). *Sei v ein Zustand. Die Nachfolgermenge $F(v)$ und die Vorgängermenge $B(v)$ sind SCC geschlossene Mengen.*

Beweis. Da eine Kripkestruktur und die Struktur die durch Umkehrung der Transitionen entsteht die selben SCCs aufweisen genügt es den Beweis für die Vorgängermenge zu führen.
Nehmen wir an die zu v gehörende Vorgängermenge $B(v)$ sei nicht SCC geschlossen. Es gibt also eine SCC A, so dass $A \cap B(v) \neq \{\}$ und $A \setminus B(v) \neq \{\}$. Sei nun

$x \in A \cap B(v)$ und $y \in A \setminus B(v)$. Da $x, y \in A$ gilt, und da A ein SCC ist, ist auch $y \in B(x)$. Da weiter $x \in B(v)$ gilt $B(x) \subseteq B(v)$ und damit auch $y \in B(v)$. Also ist $A \cap B(v) \subseteq B(v)$ oder $A \neq \{\}$ was der Annahme widerspricht. □

Lemma 2 (SCC-Geschlossenheit der Differenz). *Die Differenz zweier SCC-Geschlossener Mengen ist SCC-Geschlossen.*

Daraus ergibt sich folgendes Vorgehen. Der Algorithmus wählt aus der Zustandsmenge einen Zustand aus und berechnet den Schnitt seiner Vorgänger- und Nachfolgermengen. Da beide Mengen SCC geschlossen sind liegt ein möglicher SCC vollständig in beiden Mengen und ist damit identisch mit dem Schnitt beider Mengen. Anschließend wird der Zustand und seine Vorgängermenge aus der Zustandsmenge entfernt. Dieses Vorgehen wird solange wiederholt bis keine Zustände mehr verbleiben.
Die Vorgänger- und Nachfolgermengen können durch einfache Fixpunktiteration bestimmt werden. Der gefundene Fixpunkt wird dabei auf eine bestimmte Menge beschränkt. Diese Aufgabe wird von den Funktionen **BackwardSet**(v, V) und **ForewardSet**(v, V) erfüllt.

```
function XieBeerel (S)
    while (S ≠ {}) do
        stack sccs;
        v := selectfrom(S);
        B := BackwardSet(v, S);
        XBRecur(v, B, sccs);
        S := S \ (v ∪ B);
    end;
    return(sccs);
end
```

Abbildung 8: Xie Beerel Top-Level Algorithmus

Zur Vereinfachung werden zu dem gewählten Zustand nach der Fixpunktberechnung diejenigen Vorgängerzustände berechnet, welche zu keinem SCC gehören, und von der weiteren Betrachtung ausgeschlossen.

Definition 10 (Vorgänger mit endlicher maximaler Distanz). *Ein Zustand u ist ein Vorgänger mit endlicher maximaler Distanz des Zustands v wenn jeder Pfad von u nach v endlich ist.*

Notwendige Vorraussetzung hierbei ist, dass u zu keinem SCC gehört, da sonst ein unendlicher Pfad vorläge. Diese Berechnung wird von der Funktion **FMDPred**(v, V) durchgeführt und wiederum auf eine Menge V eingeschränkt wird.

```
function FMDPred (v, V)
        pred := {};
        front := v;
        bound := V;
        while (front ≠ {}) do
                front := bound ∩ pre(front) \ pre(bound);
                pred := pred ∪ front;
                bound := bound \ front;
        end;
        return(pred);
end
```

Abbildung 9: Berechnung der Vorgänger mit endlicher maximaler Distanz

Weiter werden die übrig gebliebenen Vorgänger rekursiv auf das Vorhandensein von SCCs untersucht.
Da der Algorithmus in jedem Durchlauf mindestens einen Zustand aus der Menge entfernt, wird er höchstens $|\mathcal{K}|$ mal aufgerufen. Ein Durchlauf benötigt $O(|\mathcal{S}|^2)$ symbolische Operationen.

Lemma 3 (Laufzeit XieBeerel). *XieBeerel zerlegt eine Kripkestruktur in ihre SCCs in $O(|\mathcal{S}|^2)$ symbolischen Schritten mit Laufzeit von $O(|\mathcal{K}||\mathcal{S}|^2)$.*

3.2 LockStep

Im Jahr 2000 wurde von Bloem, Gabow und Somenzi der Algorithmus LockStep [1] vorgestellt der ebenso wie der von Xie und Beerel einen Zustand auswählt aber dann die Vorgänger- und Nachfolgermengen simultan berechnet. Sobald eine Iteration zum Ende kommt wird die gefundene Menge in $Converged$ gespeichert und die verbleibende Iteration auf die bereits berechnete Menge beschränkt statt sie vollständig zu berechnen. Die Rekursion arbeitet dann auch mit der bereits eingeschränkten Menge weiter.
Um die Komplexität des Algorithmus abzuschätzen wird für jeden Zustand s die Menge der symbolischen Schritte die zur Bestimmung der SCC von s benötigt

```
function XBRecur (v, B, sccs)
    F := ForewardSet(v, B);
    if (F ≠ {}) then
        push(sccs, F);
    end;
    x := F ∪ v;
    R := B \ x;
    y := FMDPred(x, R);
    R := R \ y;
    IP := pre(y ∪ x) ∩ R;
    while (R ≠ {}) do
        v := selectfrom (IP);
        B := BackwardSet (v, R);
        XBRecur (v, B, sccs);
        R := R \ (v ∪ B);
        IP := IP \ (v ∪ B);
    end;
end
```

Abbildung 10: Rekursive Funktion nach Xie Beerel

werden in $Charge(s)$ gezählt. Wenn alle SCCs bestimmt sind ergibt sich die Zahl der ausgeführten Schritte durch $\sum_{s \in \mathcal{S}} Charge(s)$. Dabei sind die folgenden Beobachtungen für das Zählen der Schritte von Bedeutung:
(1) Nach der Fixpunktiteration bilden C, S_1 und S_2 eine Partitionierung von V.
(2) Nach der Fixpunktiteration gilt $|S_1| \leq \lfloor \frac{1}{2}|\mathcal{S}| \rfloor$ oder $|S_2| \leq \lfloor \frac{1}{2}|\mathcal{S}| \rfloor$ denn sonst wäre $|S| = |C| + |S_1| + |S_2| > 1 + 2\lfloor \frac{1}{2}|\mathcal{S}| \rfloor \geq |\mathcal{S}|$.
(3) Die erste **while** Schleife hat maximal $|C| + \lfloor \frac{1}{2}|\mathcal{S}| \rfloor$ Iterationen. Die Schleife stoppt wenn entweder die Vorgänger- oder die Nachfolgermenge bestimmt ist. Betrachtet man die Fälle aus (2)

(a) gilt $|S_1| \leq \lfloor \frac{1}{2}|\mathcal{S}| \rfloor$ also $|Converged \setminus C| \leq \lfloor \frac{1}{2}|\mathcal{S}| \rfloor$ und es folgt, dass $|Converged| \leq |C| + \lfloor \frac{1}{2}|S| \rfloor$. Da $|Converged|$ die Anzahl der Iterationen der Schleife bestimmt gilt die Aussage.

(b) gilt $|S_2| \leq \lfloor \frac{1}{2}|S| \rfloor$ und damit $|\mathcal{S}| - |C| \leq \lfloor \frac{1}{2}|\mathcal{S}| \rfloor$ Die noch nicht bestimmte Menge ist Teilmenge von $C \cup S_2$ und hat maximal $|C| + |S_2| \leq |C| + \lfloor \frac{1}{2}|\mathcal{S}| \rfloor$ Elemente. Also gilt auch für diesen Fall die Aussage.

(4) Die zweite **while** Schleife bestimmt die SCC C und hat damit maximal $|C|$

```
function LockStep (V, sccs)
    if (V = ∅) then
        return;
    end;
    v := selectfrom (V);
    F := {v};
    FFront := {v};
    B := {v};
    BFront := {v};
    while (FFront ≠ ∅ ∧ BFront ≠ ∅) do
        FFront := suc (FFront) \ F;
        BFront := pre (BFront) \ B;
        F := FFront ∪ F;
        B := BFront ∪ B;
    end;
    if (FFront = ∅) then
        Converged := F;
    else
        Converged := B;
    end;
    while (FFront ∩ B ≠ ∅ ∧ BFront ∩ F ≠ ∅) do
        FFront := suc (FFront) \ F;
        BFront := pre (BFront) \ B;
        F := FFront ∪ F;
        B := BFront ∪ B;
    end;
    C := F ∩ B;
    push(sccs, C);
    S1 := Converged \ C;
    S2 := V \ Converged;
    LockStep (S1, sccs);
    LockStep (S2, sccs);
end
```

Abbildung 11: Der Algorithmus LockStep

Iterationen, da in jeder Iteration mindestens ein Zustand des SCCs gefunden wird. (5) Vor den rekursiven Aufrufen werden maximal $3|C| + 2\lfloor\frac{1}{2}|\mathcal{S}|\rfloor$ symbolische Schitte ausgeführt. Zwei werden in der ersten Schleife ausgeführt und einer in der zweiten Schleife. Das ergibt 2 ($|C| + \lfloor\frac{1}{2}|\mathcal{S}|\rfloor$) + $|C|$ Schritte.

Mit diesen Feststellungen können nun die $Charge(s)$ bestimmt werden. Wir nennen S_{small} := S_1 wenn $|S_1| \leq \lfloor\frac{1}{2}|\mathcal{S}|\rfloor$ und S_{small} := S_2 anderenfalls.

Für die Zustände $s \in C$ gilt $Charge(s)$:= $Charge(s)$ + 3
Für die Zustände $s \in S_{small}$ gilt $Charge(s)$:= $Charge(s)$ + 2
Für die Zustände $s \in \mathcal{S} \setminus (C \cup S_{small})$ ändert sich $Charge(s)$ nicht.

Nun muss noch bedacht werden wie oft diese Berechnungen für einen Zustand s ausgeführt werden können.
Gehört s zu der SCC die gerade berechnet wird, so wird $Charge(s)$ um 3 erhöht und sonst nicht weiter verändert, da die Zustände der SCC aus den Mengen für die rekursiven Aufrufe entfernt werden. Ist s nicht im SCC so wird er in einem der beiden rekursiven Aufrufe verwendet. Gilt $s \in \mathcal{S} \setminus (C \cup S_{small})$ so ändert sich $Charge(s)$ nicht und der Fall muss nicht weiter beachtet werden. Gilt hingegen $s \in S_{small}$ wird $Charge(s)$ um 2 erhöht, was nur $log_2\ |\mathcal{S}|$ mal auftreten kann, da $|S_{small}| \leq \lfloor\frac{1}{2}|\mathcal{S}|\rfloor$.
Für den Zustand s ergibt sich damit $Charge(s) \leq 2log_2\ |\mathcal{S}|$ + 3, und $\sum_{s\in\mathcal{S}} Charge(s)$ hat als obere Grenze O ($|\mathcal{S}|\ log_2\ |\mathcal{S}|$).

Lemma 4 (Laufzeit LockStep). *Der Algorithmus LockStep berechnet die SCCs einer Kripkestruktur mit O ($|\mathcal{S}|\ log_2\ |\mathcal{S}|$) symbolischen Schritten. Die Laufzeit beträgt O ($|\mathcal{K}|\ |\mathcal{S}|\ log_2\ |\mathcal{S}|$).*

Auch LockStep kann, wie XieBeerel, um die Funktion FMDPred erweitert werden. Die Verwendung, auch für SpineSCC wird in [7] empfohlen. Da vor der Ausführung nicht bekannt ist welche Fixpunktiteration zuerst beendet wird, und in welchem rekursiven Aufruf entsprechend die Vorgängermenge der SCC behandelt wird, muss eine Fallunterscheidung vorgenommen werden, da die Nutzung von FMDPred nur auf den Vorgängern der SCC sinnvoll ist. Entspricht $Converged$ der Vorgängermenge, ändert sich S_1 zu $Converged \setminus C \setminus$ FMDPred(C, $Converged \setminus C$). Entspricht $Converged$ der Nachfolgermenge so muss entsprechend S_2 zu $V \setminus Converged \setminus$ FMDPred(C, $V \setminus Converged$) verändert werden.

3.3 SpineSCC

Gentilini, Piazza und Policriti schlugen 2003 einen Algorithmus vor, der das Verfahren zur Zerlegung in SCCs weiter verbessert [5]. Die Verbesserung beruht darauf dass die Berechnung der Nachfolgermengen durch die Zwiebelringe eine Ordnung auf den Nachfolgern bestimmt. Dazu werden Spines und Skeletons berechnet um die Reihenfolge der zu berechnenden SCCs zu bestimmen.

Definition 11 (Spine). *Sei $S_{spine} \subseteq \mathcal{S}$. Das Paar ($S_{spine}$, v) ist ein Spine genau dann wenn $v \in S_{spine}$ und eine bijektive Abbildung existiert*
$f : \mathcal{S} \rightarrow \{1, \ldots, |S_{spine}|\}$, so dass
(1) $f(v) = |S_{spine}|$
(2) $\forall\, u, w \in S_{spine}.((u,w) \in \mathcal{R} \Rightarrow f(w) = f(u) + 1 \vee f(w) \leq f(u))$
(3) $\forall\, u, w\, .(f(w) = f(u) + 1 \Rightarrow (u, w) \in \mathcal{R})$

Die Bijektion f bezeichnen wir als die Zertifizierungsfunktion.

Lemma 5 (Eigenschaften eines Spines). *Sei $S_{spine} = \{s_1, \ldots, s_n\}$ und (S_{spine}, s_n) ein Spine mit der Zertifizierungsfunktion $f(s_i) = i$.*
(1) ($\{s_1, \ldots\ s_k\}$, s_k) ist ein Spine, und f ist beschränkt auf $\{s_1, \ldots\ s_k\}$
(2) ($\{s_k, \ldots\ s_n\}$, s_n) ist ein Spine, und f ist beschränkt auf $\{s_k, \ldots\ s_n\}$ mit $f(s_j) = j - k + 1$
(3) Die Zertifizierungsfunktion eines Spines ist eindeutig bestimmt.
(4) Ist ($s_1, \ldots\ s_n$) der kürzeste Pfad von s_1 nach s_n, so ist (($s_1, \ldots\ s_n$), s_n) ein Spine.
*(5) **pre**($\{s_n\}$) $\cap$ $\{s_1, \ldots, s_n\}$ = $\{s_{n-1}\}$*
(6) $S_{spine} \cap$ SCC(s_1) = $\{s_1, \ldots\ s_k\}$ für ein k
(7) $S_{spine} \cap$ SCC(s_n) = $\{s_l, \ldots\ s_n\}$ für ein l
(8) entweder ist $s_n \in$ SCC(s_1) oder ($S_{spine} \setminus$ SCC(s_1), s_n) ist ein Spine
*(9) entweder ist $s_1 \in$ SCC(s_n) oder ($S_{spine} \setminus$ SCC(s_n), s_l) ist ein Spine, wobei **pre**($S_{spine} \cap$ SCC(s_n)) $\cap$ ($S_{spine} \setminus$ SCC(s_n)) = $\{s_l\}$ gilt*
(10) SCC(s_n) = FW($\{s_n\}$) $\cap \bigcup_{i=1}^{n}$ SCC(s_i), wobei FW(t) die Nachfolgermenge des Zustandes t beschreibt.

Definition 12 (Skelett der Nachfolgermenge). *Sei $\mathcal{K}$ = ($\mathcal{S}$, $\mathcal{I}$, $\mathcal{R}$, $\mathcal{L}$) eine Kripkestruktur und $s_1 \in \mathcal{S}$. Sei weiter ($s_1, \ldots\ s_n$) $\in$ FW($\{s_1\}$ ein kürzester Pfad mit maximaler Länge von in FW($\{s_1\}$ dann ist der Spine (($s_1, \ldots\ s_n$), s_n) ein Skelett von FW($\{s_1\}$).*

```
function SpineSCC (V, s, node, sccs)
    if (V = ∅) then
        return;
    end;
    if (s = ∅) then
        node := selectfrom(V);
    end;
    (fw, news, newnode) := SkeletonForeward(V, node);
    scc := node;
    while (((pre(scc) ∩ fw) \ scc) ≠ ∅) do
        scc := scc ∪ (pre(scc) ∩ fw);
    end;
    push(sccs, scc);
    V' := V \ fw;
    s' := news;
    node' := pre(scc ∩ s) ∩ (s \ scc);
    SpineSCC(V', s', node', sccs);
    V' := fw \ scc;
    s' := news \ scc;
    node' newnode \ scc;
    SpineSCC(V', s', node', sccs);
end
```

Abbildung 12: Der Algorithmus SpineSCC

Der Algorithmus berechnet zu einem Knoten v die Nachfolgermenge zusammen mit einem Skelett der Nachfolgermenge und bestimmt dann die Knoten der Nachfolgermenge, die einen Pfad zu v besitzen. Dabei nutzt der Algorithmus die Eigenschaften der Spines um ein geordnetes Vorgehen bei der Bestimmung der SCCs zu erreichen. Die lineare Laufzeit ergibt sich dadurch, dass ein Skelett für die Bestimmung mehrerer SCCs benutzt werden kann und sich so die symbolischen Schritte die für das Skelett nötig waren auf mehrere SCCs verteilen.

Lemma 6 (Laufzeit SpineSCC). *Der Algorithmus SpineSCC berechnet die stark zusammenhängenden Komponenten einer Kripkestruktur in $O(|\mathcal{S}|)$ symbolischen Schritten.*

Ein Beweis dieses Lemmas ist in [6] zu finden.

```
function SkeletonForward (V, v)
    stack Stack;
    Level := v;
    fw := ∅;
    while (Level ≠ ∅) do
        push(Stack, Level);
        fw := fw ∪ Level;
        Level := suc(Level) \ fw;
    end;
    Level := pop(Stack);
    s := selectfrom(Level);
    n := s;
    while (Stack ≠ ∅) do
        Level := pop(Stack);
        s := s ∪ selectfrom(pre(s) ∩ Level);
    end;
    return(fw, s, n);
end
```

Abbildung 13: Skelett Berechnung

Der Algorithmus SpineSCC kann ebenfalls von der Elimination der Vorgänger mit endlicher maximaler Distanz profitieren [7]. Dazu genügt es im ersten rekursiven Aufruf V' zu $V \setminus fw \setminus$ FMDPred(scc, $V \setminus fw$) zu ändern, um FMDPred auf die Vorgänger anzuwenden.

3.4 Implementierung und experimentelle Ergebnisse

Im Rahmen dieser Arbeit wurden die vorgestellten Algorithmen in das Verifikations-Werkzeug Averest[4] integriert. Averest bietet einen Compiler für die synchrone Sprache Quarz, in der zu den beschriebenen Systemen eine temporallogische Spezifikation angegeben werden kann. Die Spezifikation lässt sich dann mit dem Modellprüfungswerkzeug Beryl überprüfen. Beryl wurde um die Fähigkeit Gegenbeispiele zu berechnen erweitert, wobei alle drei Algorithmen sowohl ohne, als auch mit der Berechnung der Vorgänger mit endlicher maximaler Distanz imple-

[4]Informationen sind unter http://www.averest.org/ erhältlich.

mentiert wurden. In den Tabellen verweist xb auf den Algorithmus von Xie und Beerel, ls auf LockStep und spine auf SpineSCC. Die Erweiterungen mit den Vorgängern endlicher maximaler Distanz sind durch das angehängte f gekennzeichnet. Getestet wurde anhand von Beispielen die im Averest Paket enthalten sind. Dabei wurde die Laufzeit der Algorithmen zur Zerlegung in SCCs gemessen, und die Anzahl boolscher Operationen und symbolischer Schritte gezählt.
Die Tests wurden auf einem Dual Xeon Rechner mit 3,06 Ghz durchgeführt, wobei 4 GB Arbeitsspeicher zur Verfügung standen.

Am Beispiel der Kripkestruktur aus Abbildung 2 ergaben sich die folgenden Messwerte.

Algorithmus	Zeit (sec)	Operationen	Schritte
xb	0	180	33
ls	0	162	33
spine	0	339	67
xb-f	0	228	47
ls-f	0	232	42
spine-f	0	338	73

Hier zeigt sich, dass die Bestimmung der Vorgänger mit endlicher maximaler Distanz bei diesem Beispiel eher zu einer Verschlechterung führt, auch wenn das nicht an der Laufzeit wahrnehmbar ist. Betrachtet man die zugrundeliegende Kripkestruktur in Abbildung 2, so ist leicht zu erkennen dass es so gut wie keine Zustände gibt, die zu keiner SCC gehören, und so FMDPred zu keiner Verbesserung führt.
Dass dem nicht immer so ist zeigt sich am Beispiel des Abros. Dieses Beispiel ist bei der Übersetzung skalierbar durch die Variable $numevents$. Betrachten wir zunächst XieBeerel ohne FMDPred.

XieBeerel					
$numevents$	Variablen	SCCs	Zeit (sec)	Operationen	Schritte
2	8	5	0,01	4863	675
3	11	9	0,14	44026	5380
4	12	17	0,43	92703	10911
5	17	33	42,98	3582126	361056
6	18	65	to	-	-

Bei LockStep und SpineSCC, die beide ohne die Erweiterung um FMDPred vorgestellt wurden, zeigt sich ein ähnliches Bild.

LockStep					
$numevents$	Variablen	SCCs	Zeit (sec)	Operationen	Schritte
2	8	5	0,01	4045	643
3	11	9	0,14	36585	4998
4	12	17	0,35	76741	10027
5	17	33	55,98	2976147	318912
6	18	65	78,52	6171880	638548
7	19	129	190,51	12776817	1277838
8	20	257	to	-	-

SpineSCC					
$numevents$	Variablen	SCCs	Zeit (sec)	Operationen	Schritte
2	8	5	0,01	6138	666
3	11	9	0,2	59003	5224
4	12	17	0,49	126382	10566
5	17	33	44,28	5176403	337666
6	18	65	81,78	10861705	679006
7	19	129	230,8	22714259	1363148
8	20	257	to	-	-

Wobei to für einen Timeout steht und der Algorithmus nach einer Stunde ohne Ergebnis abgebrochen wurde. Entsprechend bezeichnet mo einen Memoryout.
Xie und Beerel stellten ihren Algorithmus mit der Verbesserung der Elimination der Vorgänger endlicher maximaler Distanz vor, dessen Ergebnisse sich in diesem Beispiel stark von denen der Algorithmen ohne FMDPred abhob.

XieBeerel	mit FMDPred				
$numevents$	Variablen	SCCs	Zeit (sec)	Operationen	Schritte
2	8	5	0	260	52
3	11	9	0,01	503	95
4	12	17	0,02	964	178
5	17	33	0,1	2045	347
6	18	65	0,63	4094	683
7	19	129	4,79	8255	1355
8	20	257	0,1	16700	2698
9	29	513	0,29	37961	5387
10	30	1025	0,66	76874	10763
15	35	32769	30,82	2621523	344076
20	56	?	mo	-	-

Wegen der deutlich spürbaren Unterschiede wurde FMDPred ebenfalls in LockStep und SpineSCC integriert.

LockStep	mit FMDPred				
$numevents$	Variablen	SCCs	Zeit (sec)	Operationen	Schritte
2	8	5	0	255	46
3	11	9	0,01	490	84
4	12	17	0,02	923	154
5	17	33	0,14	1960	300
6	18	65	1,09	3913	588
7	19	129	8,33	7882	1164
8	20	257	0,12	15939	2314
9	29	513	0,36	36436	4620
10	30	1025	0,97	73813	9228
15	35	32769	79,01	2523228	294924
20	56	?	mo	-	-

SpineSCC	mit FMDPred				
$numevents$	Variablen	SCCs	Zeit (sec)	Operationen	Schritte
2	8	5	0	367	56
3	11	9	0	755	103
4	12	17	0,02	1427	188
5	17	33	0,11	3281	367
6	18	65	0,99	6614	719
7	19	129	7,49	13467	1423
8	20	257	0,16	27523	2828
9	29	513	to	-	-
10	30	1025	1,36	140498	11279
15	35	32769	102,11	4980975	360464
20	56	?	mo	-	-

Betrachtet man die Anzahl der symbolischen Schritte und der boolschen Operationen so zeigt sich, dass LockStep bei beidem weniger braucht als XieBeerel, während SpineSCC etwas über XieBeerel liegt.
War bei den Algorithmen ohne FMDPred noch mal XieBeerel und mal LockStep der schnellste, so ist mit FMDPred XieBeerel immer der schnellste. SpineSCC zeigt sich trotz seiner theoretisch linearen Laufzeit als der langsamste der getesteten Algorithmen.
Weiter wurde noch am Beispiel des Bubblesort-Algorithmus getestet, welcher ebenfalls skalierbar ist.

$size$	Variablen	SCCs	Zeit (sec)	Operationen	Schritte
XieBeerel	mit FMDPred				
3	20	4096	40,29	7911234	1048423
5	30	?	to	-	-
LockStep	mit FMDPred				
3	20	4096	581,76	23156826	2487602
5	30	?	to	-	-
SpineSCC	mit FMDPred				
3	20	4096	1437,64	63760958	5345036
5	30	?	to	-	-

Auch bei diesem Beispiel ist XieBeerel, trotz seiner hohen Komplexität, mit großem Abstand der schnellste der Algorithmen. LockStep liegt nicht so nahe an dessen

Ergebnissen wie es im Abro Beispiel der Fall war. Generell lässt sich feststellen, dass bei großen Kripkestrukturen alle drei Algorithmen entweder auf Grund eines Timeout oder eines Memoryout ohne Ergebnis abgebrochen werden mussten. Für den tatsächlichen praktischen Einsatz sind die Verfahren daher noch nicht geeignet.
Der deutlichste Gewinn wurde durch die Einführung der Elimination der Vorgänger ohne SCC-Zugehörigkeit erreicht, während die Senkung der Komplexität bei diesen Beispielen eher eine Verschlechterung bedeutete. Nun wurde FMD-Pred speziell im Rahmen des XieBeerel Algorithmus entwickelt und die direkte Verwendung in LockStep und SpineSCC hat auch dort zu einer deutlichen Verbesserung geführt. Es stellt sich die Frage ob für LockStep und SpineSCC weitere Optimierungen gefunden werden können, die sich direkter an den speziellen Eigenarten des jeweiligen Algorithmus orientieren, um ihre Laufzeiten zu senken. Oder ob weitere Eigenschaften der Kripkestrukturen existieren die für die SCC-Zerlegung ausgenutzt werden können. Auch könnte von Interesse sein, auf welchen Kripkestrukturen die jeweiligen Algorithmen ihre besten Ergebnisse zeigen, und ob im praktischen Einsatz dynamisch der optimale Algorithmus gewählt werden kann.

4 Zusammenfassung

Für die symbolische Modellprüfung existiert ein Vorgehen um zu einer nicht erfüllten LTL Spezifikation Gegenbeispiele zu berechnen. Dabei wird die Kripkestruktur zunächst in ihre stark verbundenen Komponenten zerlegt, unter denen sich dann eine oder mehrere faire SCCs befinden. Diese werden mit den Fairness Bedingungen der Spezifikation identifiziert und auf ihre Erreichbarkeit geprüft. Danach wird ein Pfad von einem Initialzustand zu einer erreichbaren fairen SCC bestimmt und ein Zyklus in der SCC gewählt, so dass die Fairness Bedingungen auf dem Zyklus mindestens einmal wahr werden. Der so gefundene faire Pfad wird anschließend im Value-Change-Dump-Format zur späteren Betrachtung gespeichert.

Im praktischen Einsatz ist bei diesem Vorgehen die Zerlegung der Kripkestruktur der aufwändigste Schritt. Es existieren dafür mehrere symbolische Algorithmen, von denen der von Xie und Beerel auf den hier verwendeten Beispielen am schnellsten arbeitet, obwohl er die höchste Komplexität aufweist. Es zeigte sich, dass das Verhalten der Algorithmen stark von der zu zerlegenden Kripkestruktur abhängig ist.

Von dem Algorithmus von Xie und Beerel stammt eine Erweiterung um die Vorgänger eines Zustandes von der weiteren Betrachtung auszuschließen, die nicht zu einer SCC gehören können. Diese Elimination kann abhängig von der zu Grunde liegenden Kripkestruktur eine große Verbesserung in der Laufzeit aller Algorithmen bewirken. Es stellt sich die Frage ob für die Algorithmen LockStep und SpineSCC spezielle Erweiterungen möglich sind, die ihnen zu besserem Laufzeitverhalten verhelfen können, oder ob weitere Verbesserungen für alle drei Verfahren möglich sind, damit sie in der Praxis eingesetzt werden können.

Literatur

[1] R. Bloem and H.N. Gabow and F. Somenzi: *An Algorithm for Strongly Connected Component Analysis in n log n Symbolic Steps.* Conference on Formal Methods in Computer Aided Design (FMCAD), Springer, 2000, Seiten 37-54.

[2] Randal E. Bryant: *Graph-Based Algorithms for Boolean Function Manipulation.* IEEE Transactions on Computers, Vol C-35, No. 8, August, 1986, Seiten 677 - 691.

[3] E.M. Clarke and O. Grumberg and K.L. McMillan and X. Zhao: *Efficient Generation of Counterexamples and Witnesses in Symbolic Model Checking.* Design Automation Conference (DAC), San Francisco, California, USA, 1995, Seiten 427-432.

[4] E. Clarke, O. Grumberg, D. Peled: *Model Checking.* The MIT Press, Cambridge, Massachusetts, London, England, 2000.

[5] R. Gentilini and C. Piazza and A. Policriti: *Computing Strongly Connected Components in a Linear Number of Symbolic Steps.* Symposium on Discrete Algorithms (SODA), ACM/SIAM, 2003, Seiten 573-582.

[6] K. Schneider: *Verification of Reactive Systems - Formal Methods and Algorithms.* Texts in Theoretical Computer Science (EATCS Series). Springer, 2003.

[7] K. Schneider: *Verification of Reactive Systems - Formal Methods and Algorithms.* Erweiterungen zum Buch, http://rsg.informatik.uni-kl.de/research/VeReSys/extended/FairStates.pdf

[8] R. Tarjan: *Depth first search and linear graph algorithms.* SIAM Journal on Computing. Nummer 2, 1972, Seiten 146-160.

[9] IEEE: *IEEE Standard Verilog.* 2001.

[10] A. Xie and P.A. Beerel: *Implicit Enumeration of Strongly Connected Components.* Conference on Computer Aided Design (ICCAD), IEEE Computer Society, 1999, Seiten 37-40.

[11] A. Xie and P.A. Beerel: *Implicit Enumeration of Strongly Connected Components and an Application to Formal Verification.* IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems, Ausgabe 10, IEEE Computer Society, 2000, Seiten 1225-1230.