Formale Verifikation von LTL-Formeln für SystemC-Beschreibungen

Daniel Große Rolf Drechsler
Institut für Informatik, Universität Bremen, 28359 Bremen
{grosse, drechsle}@informatik.uni-bremen.de

Zusammenfassung

Es wird eine formale Verifikationsmethodik zum Nachweis von Formeln in *linear time temporal logic* (LTL) für Systeme, die in SystemC spezifiziert sind, vorgestellt. SystemC bietet als C++-Klassenbibliothek die Möglichkeit eines gemeinsamen Hard- und Softwaresystementwurfs. Für den Beweis einer LTL-Eigenschaft in SystemC kann die Erfüllbarkeit einer Booleschen Funktion betrachtet werden, die aus der Eigenschaft und der Schaltkreisbeschreibung mittels symbolischer Methoden konstruiert wird. Im Gegensatz zu simulationsbasierten Ansätzen kann dabei Vollständigkeit gewährleistet werden. Anhand einer Fallstudie eines skalierbaren Arbiters wird die Effizienz des Beweisverfahrens untersucht.

1 Einleitung

Der Entwurf einer Schaltung erfolgt in der Regel auf RT-Ebene mittels einer speziellen Hardwarebeschreibungssprache, wie z.B. VHDL oder Verilog. Da bei heutigen Systemen sowohl die Komplexität als auch der Anteil an Systemfunktionen, die in Software implementiert werden, immer mehr zunimmt, werden in der Praxis auf Systemebene häufig zunächst Referenzmodelle in simulierbaren Beschreibungssprachen, wie *C*, erzeugt. Um die entstehende Lücke zu VHDL/Verilog zu schließen, wurden verschiedene Erweiterungen von C/C++ vorgeschlagen, die es ermöglichen Hardware zu modellieren. In diesem Zusammenhang wurde die Systembeschreibungssprache *SystemC* [10] eingeführt, bei der es sich um eine C++-Klassenbibliothek handelt. SystemC ist frei verfügbar, unterstützt Beschreibungen auf hohen Abstraktionsebenen und besitzt eine hohe Simulationsgeschwindigkeit. Für die Verifikation von Systemen ist ein zyklengenauer Simulator integriert.

Die „einfache Simulation“ von Systemen reicht jedoch aus zweierlei Gründen nicht mehr aus. Einerseits nimmt die Überdeckung der Funktionalität durch die Simulation wegen der enormen Komplexität heutiger Schaltungen immer mehr ab. Andererseits steigen die Korrektheitsanforderungen an die Systeme, da diese vermehrt in sicherheitsrelevanten Bereichen eingesetzt werden. Dies führte in den vergangenen Jahren zur Entwicklung von Verifikationsansätzen basierend auf formalen Methoden. Im Äquivalenzvergleich von Schaltungen sind diese heutzutage gängige Praxis.

Um Eigenschaften für Schaltungen zu beweisen, wurden Verfahren zur Modellprüfung vorgestellt, welche sich in „klassisches“ Modellprüfen [2, 6] und *bounded model checking* (BMC) [1] einteilen lassen. BMC betrachtet dabei Eigenschaften nur bzgl. eines endlichen Zeitintervalls.

Allerdings basieren bisher alle Verifikationsansätze für SystemC lediglich auf Simulation (z.B. [8, 4]). In dieser Arbeit wird erstmals ein Ansatz für SystemC vorgestellt, der Eigenschaften, spezifiziert in *linear time temporal logic* (LTL), auf der Basis von BMC formal beweist und dabei Vollständigkeit mittels der Menge der erreichbaren Zustände garantiert. Das Verfahren betrachtet hierarchische SystemC-Designs, wobei für jedes SystemC-Modul eine Gatterbeschreibung vorliegen muss. Für ein solches Design werden die Ausgabe- und Übergangsfunktion des zu Grunde liegenden endlichen Automaten bestimmt. Anschließend erfolgt die Zustandstraversierung des Automaten. Schließlich kann der Beweis einer LTL-Eigenschaft auf ein Erfüllbarkeitsproblem zurückgeführt werden, welches auf dem „abgerollten" Schaltkreis und der Menge aller erreichbaren Zustände basiert. Mit dem vorgestellten Beweisverfahren können auch LTL-Formeln, die über unendliche Zeitintervalle argumentieren, bewiesen werden, da die Menge der erreichbaren Zustände im Beweis zur Verfügung steht.

2 Grundlagen

2.1 Modellierung von Schaltkreisen in SystemC

SystemC erlaubt die Modellierung von Schaltungen und Systemen auf unterschiedlichen Abstraktionsebenen [5, 10]. Im Rahmen dieser Arbeit werden zur Modellierung von Hardware in SystemC Grundgatter verwendet, die es ermöglichen, synchrone sequentielle Schaltkreise auf Gatterebene zu beschreiben. Als Grundgatter werden das UND-, ODER- und NICHT-Gatter, sowie das Flip-Flop modelliert. Mittels Hierarchiebildung können weitere Gatter, wie z.B. das XOR-Gatter einfach beschrieben werden. Exemplarisch sei hier der Aufbau des UND-Gatters in Abbildung 1 dargestellt: Innerhalb des SystemC-Moduls `AndGate` lassen sich die Schnittstellendefinition (die ersten drei Zeilen), die Funktionalität (`doAnd`) und die Sensitivitätsliste unterscheiden. Das `AndGate` ist sensitiv auf die beiden Eingänge `in1` und `in2`, d.h. falls während der Simulation eine Änderung an einem der beiden Eingänge stattfindet, so wird der Wert des Ausganges `out` neu berechnet, da dann der Prozess `doAnd()` aktiviert wird. Ferner wird auch die Methode `end_of_elaboration()` benutzerspezifisch überladen. Ihre Bedeutung für die Zustandstraversierung wird im Abschnitt 3 erläutert.

2.2 Simulation von SystemC-Beschreibungen

Nach der Spezifikation eines Systems in SystemC ist dessen Simulation durch den SystemC-Simulationskernel möglich. Dazu ist die Instanziierung des *Top-Level*-Moduls erforderlich. Dies geschieht in SystemC in der Funktion `sc_main()`, die die `main()`-Funktion eines typischen C++-Programms ersetzt. Darin werden die Top-Level-Signale, Clocks und Module deklariert bzw. instanziiert und anschließend die Module untereinander verbunden. Zur Validierung des Designs erstellt man eine *testbench* (ebenfalls als ein Modul) und verwendet diese als Stimuli-Generator. Vor dem Start der Simulation mit `sc_start` wird festgelegt, welche Signalverläufe aufgezeichnet werden sollen. Es kann auch ein Modul entworfen werden, welches die Ausgaben des Systems überprüft. Zur Simulation wird die Systembeschreibung mit einem Standard-C++-Compiler übersetzt und anschließend gestartet. Es ist also keine weitere Simulationsumgebung erforderlich.

```
SC_MODULE(AndGate) {
 sc_in<bool> in1;
 sc_in<bool> in2;
 sc_out<bool> out;
 void doAnd();
 SC_CTOR(AndGate) {
  SC_METHOD(doAnd);
  sensitive << in1 << in2;
 }
 void end_of_elaboration() {
  symb->reportAND(name(),out,in1,in2);
 }
};
void AndGate::doAnd() {
  out.write(in1.read() && in2.read());
}
```

Abbildung 1: UND-Gatter

2.3 Linear Time Temporal Logic

Zunächst wird LTL eingeführt. Die hier betrachtete Logik FLTL *(finite linear time temporal logic)* basiert auf LTL und ermöglicht es zusätzlich auch über endliche Intervalle auf einem linearen Zeitstrahl argumentieren zu können [8]. Im Folgenden sei V eine Menge von Variablensymbolen.

Definition 2.1 Eine *Signalfolge* $T[m \dots n]$ $(m \leq n)$ ist eine Abbildung von $\{m, \dots, n\} \rightarrow 2^V$. Sind m und n aus dem Kontext ersichtlich, so schreibt man auch einfach nur T. Die Menge aller Signalfolgen $T[m \dots n]$ $(T[0 \dots \infty])$ wird bezeichnet mit $\mathcal{T}$ $(\mathcal{T}^\infty)$.

Definition 2.2 Für zwei Signalfolgen $T'[0 \dots n']$ und $T[0 \dots n]$ mit $n' > n$ ist T' eine Erweiterung von T, falls gilt: $\forall i,\ 0 \leq i \leq n : T(i) = T'(i)$.

Definition 2.3 Die Menge LTL von syntaktisch korrekten LTL-Formeln ist induktiv definiert:

- $V \subset LTL$
- $\neg f, f \wedge g \in LTL$ für $f, g \in LTL$
- $X_{[m]} f,\ G_{[m,n]} f,\ F_{[m,n]} f \in LTL$ für $f \in LTL$ und $m \in \mathbb{N}$ und $n \in \mathbb{N} \cup \{\infty\}$

Wird für F oder G im Index statt des Intervalls lediglich eine Zahl n angegeben, so bedeutet dies, dass $m = 0$ ist. Wird überhaupt kein Intervall bei einem temporalen Operator angeben, dann ist für F und G $m = 0$ und $n = \infty$[1], und für X $m = 1$. Die Standard-LTL-Operatoren $F,\ G$ und X sind also Spezialfälle mit $m = 0$ und $n = \infty$ bzw. $m = 1$.

LTL-Formeln werden über den eingeführten Signalfolgen interpretiert.

[1] Man spricht in diesem Fall auch von *unbounded*-Formeln.

Definition 2.4 Die Modellrelation $\models_i \subset (\mathcal{T}^\infty, LTL)$ ist definiert durch:

$$\begin{array}{lcl} T \models_i a & \Leftrightarrow & a \in T(i) \\ T \models_i \neg f & \Leftrightarrow & T \not\models_i f \\ T \models_i f \wedge g & \Leftrightarrow & T \models_i f \text{ und } T \models_i g \\ T \models_i X_{[m]} f & \Leftrightarrow & T \models_{i+m} f \\ T \models_i G_{[m,n]} f & \Leftrightarrow & \forall j,\ i+m \leq j \leq i+n : T \models_j f \\ T \models_i F_{[m,n]} f & \Leftrightarrow & \exists j,\ i+m \leq j \leq i+n : T \models_j f \end{array}$$

Die Semantik von LTL-Formeln ist festgelegt durch:

Definition 2.5 Sei f eine LTL-Formel und $T \in \mathcal{T}^\infty$ eine Signalfolge. Dann gilt:
$T \models f$ (T *erfüllt* f oder f *gilt in* T) $\Leftrightarrow\ T \models_0 f$.

Man erhält die temporale Logik FLTL, indem LTL-Formeln über endlichen Signalfolgen interpretiert werden:

Definition 2.6 Sei $T[0 \ldots n]$ eine endliche Signalfolge und f eine LTL-Formel, so gilt:

$T \models f \quad \Leftrightarrow$ für alle unendlichen Erweiterungen T' von T gilt: $T' \models f$.
$T \not\models f \quad \Leftrightarrow$ für alle unendlichen Erweiterungen T' von T gilt: $T' \not\models f$.
sonst sei $T \models f$ *schwebend (pending).*

3 Zustandstraversierung in SystemC

Um die Berechnung der Menge aller erreichbaren Zustände durchführen zu können, wird die Übergangsfunktion δ und die Ausgabefunktion λ des betrachteten Schaltkreises benötigt. Offensichtlich erfordert die Konstruktion von δ und λ aus einer SystemC-Beschreibung die Lösung zweier Aufgaben: Zum einen benötigt man die Netzliste des sequentiellen Schaltkreises, d.h. man muss in der Lage sein, die einzelnen Gatter und ihre Verbindungen untereinander in geeigneten Datenstrukturen identifizieren zu können. Zum anderen bedarf es einer topologischen Sortierung der Netzliste, um anschließend auf dieser eine symbolische Simulation ausführen zu können.

Ein sequentieller Schaltkreis wird im Rahmen dieser Arbeit ausschließlich mit den eingeführten SystemC-Grundgattern bzw. mit darauf basierenden mittels Hierarchiebildung gewonnenen Gattern beschrieben. Deshalb kann die Netzliste des betrachteten Schaltkreises gewonnen werden, indem die `end_of_elaboration()`-Methoden eines jeden Grundgatters überladen werden (siehe Abbildung 1). Dadurch ist es möglich, vor Simulationsbeginn jedes instanziierte Gatter und seine Verbindung zu anderen Gattern in geeigneten Strukturen zu referenzieren. Für das UND-Gatter in Abbildung 1 erkennt man, dass durch die `end_of_elaboration()`-Methode der Klasse `Symb` – innerhalb derer die hier vorgestellten Algorithmen und Datenstrukturen implementiert sind – die Information über ein konkretes UND-Gatter übermittelt wird.

Legt man dann auf Top-Level die (konstanten) Eingänge und den Startzustand des Schaltkreises fest, so steht die gesamte Information für die Zustandstraversierung zur Verfügung: Das Bestimmen einer topologischen Sortierung, die anschließende symbolische Simulation, die Berechnung der Übergangsrelation und die Fixpunktiteration der Bildberechnung können nun durchgeführt werden. Insgesamt ist somit die Menge der erreichbaren Zustände des betrachteten Schaltkreises symbolisch mittels BDDs berechnet worden. Für Details zur Zustandstraversierung in SystemC sei auf [3] verwiesen.

4 Eigenschaftsprüfung in SystemC

Der im Folgenden vorgestellte Eigenschaftsprüfer für SystemC setzt als Spezifikationssprache die temporale Logik LTL ein. Die zu verifizierende Implementierung liegt dabei als Schaltkreisbeschreibung in SystemC vor, welche auf den eingeführten Grundgattern basiert. Der Eigenschaftsprüfer verwendet für den Beweis einer LTL-Eigenschaft eine geeignete BDD-Repräsentation, die durch *Abrollen* analog zu BMC [1] erzeugt wird. Diese Repräsentation basiert auf der Übergangs- und Ausgabefunktion des Schaltkreises, die mit den Techniken aus Abschnitt 3 zur Verfügung stehen. Mit der Menge aller erreichbaren Zustände wird es schließlich möglich, die Gültigkeit der LTL-Eigenschaft zu beweisen bzw. zu widerlegen. Im Unterschied zur Logik FLTL, die LTL-Formeln nur über endlichen Signalfolgen interpretiert und deshalb beispielsweise die Gültigkeit einer Formel mit einem *unbounded* G-Operator nicht entscheiden kann, ist es mit dem hier vorgestellten Eigenschaftsprüfer ebenfalls möglich, den Beweis solcher Formeln durchzuführen.

4.1 Transformation einer LTL-Formel

Die Gültigkeit einer LTL-Formel f kann bewiesen bzw. widerlegt werden, indem f in eine Boolesche Funktion f^* überführt wird und anschließend die Erfüllbarkeit von f^* betrachtet wird. f^* wird dabei als BDD dargestellt und basiert auf der Übergangsfunktion δ und der Ausgabefunktion λ des zugrunde liegenden Schaltkreises. Zunächst werden nur temporale Operatoren mit endlichen Intervallen für f zugelassen. Das prinzipielle Vorgehen für die Transformation von f ist das Folgende:

1. Die LTL-Formel f wird geparst und dabei die Liste $list_{BDD-Op}$ erzeugt. Diese enthält BDD-Operationen, die in Schritt 3 zur Konstruktion von f^* verwendet werden. Ferner wird das Beobachtungsfenster $[t_{min}, t_{max}]$ von f bestimmt.

2. Der sequentielle Schaltkreis wird t_{max}-mal „abgerollt".

3. Es erfolgt die Konstruktion von f^* als BDD, indem die Einträge von $list_{BDD-Op}$ in aufsteigender Reihenfolge abgearbeitet werden.

Beim Parsen einer LTL-Formel f im Schritt 1 werden abhängig von dem im Rekursionsschritt aktuell betrachteten Formeloperator und der (den) abhängigen Teilformel(n) verschiedene BDD-Operationen der Liste $list_{BDD-Op}$ hinzugefügt. Beispielsweise ist für die Formel $F_{[2,3]}\ a$ eine Oder-Operation für die Variable (das Signal) a zum Zeitpunkt 2 und 3 in $list_{BDD-Op}$ zu speichern.

Das *Abrollen* des Schaltkreises im Schritt 2 ermöglicht es, dass Werte einzelner Signale des Schaltkreises zu unterschiedlichen Zeitpunkten betrachtet werden können. In Anlehnung an BMC [1] bedeutet Abrollen dabei, dass die *current state*-Variablen mit den *next state*-Variablen des vorangegangenen Zeitrahmens identifiziert werden. Abbildung 2 veranschaulicht den Abrollprozess. Die Übergangs- und Ausgabefunktion des Schaltkreises stehen als BDDs zur Verfügung, wobei I_t gerade den primären Eingängen des Schaltkreises entspricht. Bei $I_{t+1}, \ldots, I_{t_{max}}$ handelt es sich jeweils um neue BDD-Variablen für die Eingänge, da die Eingabe des Schaltkreises für jeden Folgezeitrahmen unterschiedlich sein kann. Die Werte der Ausgänge O_j zum Zeitpunkt j sind durch die Ausgabefunktion λ bestimmt, die dann gerade von den Eingängen in $I_t, I_{t+1}, \ldots, I_j$ der Zeitpunkte $t, t+1, \ldots, j$ und den Zustandsvariablen in S_t abhängt. Die aus dem Abrollprozess resultierenden BDDs sind für jeden Zeitrahmen mit Signalknoten des Schaltkreises assoziiert. Es kann also festgehalten werden, dass nun die Werte von Signalen, die innerhalb des Beobachtungsfensters $[t_{min}, t_{max}]$ einer Formel f liegen, berechnet bzw. untersucht werden können.

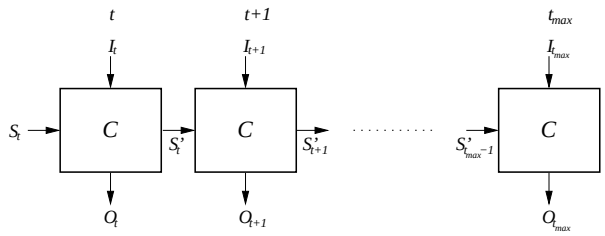

Abbildung 2: Abrollen eines sequentiellen Schaltkreises

Im Schritt 3 wird die Boolesche Funktion f^* konstruiert, indem die BDD-Operationen der Liste $list_{BDD-Op}$ ausgeführt werden. Für jede innerhalb der LTL-Formel f auftretende Variable v, auf die die BDD-Operationen in $list_{BDD-Op}$ Bezug nehmen, steht der zu ihr korrespondierende BDD-Knoten auf Grund des Abrollens zur Verfügung.

Insgesamt erhält man die Funktion f^*, repräsentiert als BDD, in deren Support höchstens die Variablen aus $I_t, I_{t+1}, \ldots, I_{t_{max}}, S_t$ liegen können. Die LTL-Eigenschaft f kann somit von einem beliebigen Zustand ausgehend betrachtet werden.

4.2 Beweis der Gültigkeit einer LTL-Formel

Der Beweis der Gültigkeit einer LTL-Formel f kann auf die Erfüllbarkeit ihrer Transformation f^* zurückgeführt werden. Es gilt die Annahme, dass in der LTL-Formel f die temporalen Operatoren F und G jeweils nur mit endlichen Zeitintervallen auftreten. Ferner sei s_0 der Startzustand des betrachteten Schaltkreises und f argumentiere von diesem Startzustand ausgehend. Dann lässt sich die Beweisaufgabe für f mittels f^* wie folgt lösen:

$$\text{Die LTL-Formel } f \text{ gilt} \Leftrightarrow (\exists I_t, I_{t+1}, \ldots, I_{t_{max}}\ \neg f^*) \wedge s_0 \equiv 0$$

f^* wird negiert, da von Interesse ist, unter welchen Belegungen f^* *nicht* gilt. Ist die linke Seite der Gleichung 0, d.h. die leere Menge, dann gilt die LTL-Formel f, denn dann gehört der Startzustand s_0 nicht zur Menge der Zustände, in denen f^* nicht gilt. Andernfalls bedeutet dies, dass ein Gegenbeispiel existiert, welches von s_0 erreichbar ist.

Betrachtet man *unbounded* LTL-Formeln der Form Gf, wobei f weiterhin nur Operatoren mit endlichen Zeitintervallen enthalten darf, dann verläuft der Beweis einer solchen Formel analog. Allerdings wird statt des Startzustandes nun die Menge der erreichbaren Zustände *Reached*, die mittels der Zustandstraversierung gemäß Abschnitt 3 berechnet wurde, in der Konjunktion verwendet, da f in allen Zuständen gelten soll. Man erhält:

$$\text{Die LTL-Formel } Gf \text{ gilt} \Leftrightarrow (\exists I_t, I_{t+1}, \ldots, I_{t_{max}}\ \neg f^*) \wedge \mathit{Reached} \equiv 0$$

Ist die Gleichung 0, also die leere Menge, dann bedeutet dies gerade, dass kein Zustand erreichbar ist, in dem f^* nicht gilt.

Mit der Beziehung $Ff \equiv \neg G \neg f$ kann die Gültigkeit einer LTL-Formel der Form Ff bewiesen werden. In diesem Fall ist zu zeigen, dass es mindestens einen erreichbaren Zustand gibt, von dem aus f gilt:

$$\text{Die LTL-Formel } Ff \text{ gilt} \Leftrightarrow (\exists I_t, I_{t+1}, \ldots, I_{t_{max}}\ f^*) \wedge \mathit{Reached} \not\equiv 0$$

Zusammenfassend kann festgehalten werden, dass mit dem vorgestellten Beweisverfahren, das einerseits auf einer geeigneten Transformation der LTL-Formel und andererseits auf der Verwendung von symbolischen Techniken und der Erreichbarkeitsanalyse beruht, die Gültigkeit einer LTL-Formel im Gegensatz zu den simulationsbasierten Verfahren

```
class Symb {
...
public:
  // Aufrufen in sc_main()
  void startSymb() {
    ...
    // die Erreichbarkeitsanalyse wurde bereits durchgeführt
    // δ, λ und Reached stehen zur Verfügung
    // Eigenschaftsprüfung:
    t_min = t_max = 0; clearList(list_BDD-Op);
    parseLTL(f, t_min, t_max, list_BDD-Op);
    // list_BDD-Op und das Intervall [t_min, t_max] sind berechnet worden
    unrollCircuit(δ, λ, t_max);
    prove(f);
  }
};
```

Abbildung 3: Pseudocode des Eigenschaftsprüfers innerhalb der Klasse `Symb`

vollständig bewiesen oder widerlegt werden kann. Im Falle der *unbounded* Operatoren F und G wird dabei die Menge der erreichbaren Zustände des zugrunde liegenden Schaltkreises benötigt.

4.3 Implementierung des Eigenschaftsprüfers

Die Algorithmen für den Eigenschaftsprüfer wurden als C++-Bibliothek zur Verfügung gestellt und innerhalb der C++-Klasse `Symb` implementiert. Diese wird zusammen mit der Schaltkreisbeschreibung übersetzt. Auf Top-Level (in `sc_main()`) wird der Eigenschaftsprüfer nach der Instanziierung des Schaltkreises mittels `startSymb()` gestartet. Zuvor wird mit der Methode `sc_assertLTL(f)` dem Eigenschaftsprüfer mitgeteilt, dass die LTL-Formel f bewiesen bzw. widerlegt werden soll. Mit dem Aufruf von `startSymb()` erfolgt dann zunächst die Erreichbarkeitsanalyse und anschließend der Beweis der LTL-Eigenschaft. In Abbildung 3 ist der Eigenschaftsprüfer als Pseudocode skizziert. Er wurde am Ende der Methode `startSymb()` angefügt – also nach den Verfahren zur Zustandstraversierung – da für den Beweis von LTL-Formeln mit *unbounded*-Operatoren die Menge der erreichbaren Zustände des Schaltkreises benötigt wird. Als Ausgabe des Eigenschaftsprüfers erhält man, ob die zu beweisende Formel gilt oder nicht. Als BDD-Paket wurde CUDD von Fabio Somenzi [9] eingesetzt.

5 Fallstudie: Ein skalierbarer Arbiter

Die Experimente wurden auf einem AMD Athlon mit 800 MHz und 512 MB Hauptspeicher durchgeführt. Als Schaltkreis wurde ein skalierbarer *Arbiter* mit den vorgestellten Grundgattern modelliert, der oft für Experimente im Rahmen der formalen Verifikation herangezogen wird [6, 8]. Im linken Teil der Abbildung 4 ist eine einzelne Arbiter-Zelle zu sehen, wobei W und T Flip-Flops bezeichnen. Im rechten Teil dieser Abbildung ist die Verbindung n solcher Zellen untereinander dargestellt. Für den skalierbaren Arbiter, bestehend aus n Zellen, sollten die folgenden drei Eigenschaften gelten:

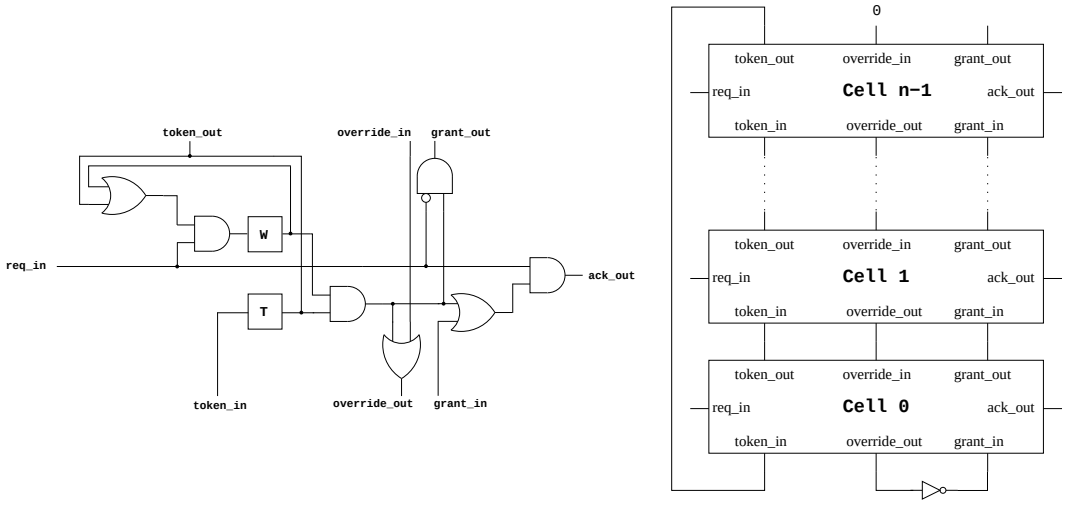

Abbildung 4: Ein skalierbarer Arbiter

1. *Wechselseitiger Ausschluss*: Keine zwei Ausgänge des Arbiters sind gleichzeitig 1:

$$G(\bigwedge_{i \neq j}^{n} \neg(ack_i \wedge ack_j))$$

2. *Lebendigkeit*: Jede Anforderung req_i wird innerhalb von $2n$ Zeitrahmen durch ack_i bestätigt:

$$G(req_i \rightarrow F_{[0,2n-1]} ack_i)$$

3. *Conservativeness*: ack_i wird nur gesetzt, wenn auch eine Anforderung req_i erfolgte:

$$G(ack_i \rightarrow req_i)$$

In Tabelle 1 sind die Ergebnisse für den Beweis der Eigenschaften 1 und 3 zusammengefasst. Die erste Spalte gibt die Anzahl der Arbiter-Zellen an. In der Spalte *Zustände* findet man die Anzahl der erreichbaren Zustände, berechnet mit dem Verfahren aus Abschnitt 3. Die Spalten *Zeit EA* und *Zeit Beweis* geben jeweils die benötigte Laufzeit für die Erreichbarkeitsanalyse bzw. Eigenschaftsprüfung in CPU-Sekunden an. Für die Durchführung der Eigenschaftsprüfung benötigt der BDD-Manager gegenüber der alleinigen Erreichbarkeitsanalyse mehr Speicherplatz. Dieser zusätzliche Speicherbedarf in MByte ist in den Spalten *Platzzuwachs* zu sehen. Während des Aufbaus der BDDs wurde sowohl für die Erreichbarkeitsanalyse als auch während der Eigenschaftsprüfung *sifting* [7] zugelassen[2]. Man erkennt, dass die beiden lokalen Eigenschaften *wechselseitiger Ausschluss* und *conservativeness* problemlos auch für große Instanzen des Arbiters bewiesen werden können.

Die *Lebendigkeits*-Eigenschaft lässt sich in der Form $G(req_i \rightarrow F_{[0,2n-1]} ack_i)$ für den Arbiter nicht beweisen. Für ihren Beweis ist eine *Umgebungsannahme* zu formulieren: Eine Anforderung req_i muss so lange auf 1 gehalten werden, bis die zugehörige Bestätigung ack_i erfolgt. Als LTL-Eigenschaft erhält man nun insgesamt:

[2]Das BDD-Paket ruft selbständig sifting auf, was den nicht monotonen Speicherplatzzuwachs bei der Eigenschaft *wechselseitiger Ausschluss* in Tabelle 1 erklärt.

Tabelle 1: Resultate beim Beweis *wechselseitiger Ausschluss* und *conservativeness*

Zellen	Zustände	Zeit EA	*wechsels. A.* Zeit Beweis	*wechsels. A.* Platz-zuwachs	*conserv.* für $i=0$ Zeit Beweis	*conserv.* für $i=0$ Platz-zuwachs
2	8	0.01	0.01	<0.01	0.01	<0.01
3	24	0.01	0.01	<0.01	0.01	<0.01
4	64	0.01	0.01	<0.01	0.01	<0.01
5	160	0.04	0.01	<0.01	0.01	<0.01
6	384	0.06	0.01	<0.01	0.01	<0.01
7	896	0.09	0.01	<0.01	0.01	<0.01
8	2048	0.15	0.01	<0.01	0.01	<0.01
9	4608	0.13	0.01	<0.01	0.01	<0.01
10	10240	0.24	0.01	<0.01	0.01	<0.01
11	22528	0.25	0.07	<0.01	0.01	<0.01
12	49152	0.26	0.01	<0.01	0.01	<0.01
20	$2.10 \cdot 10^7$	1.26	0.25	0.05	0.01	<0.01
50	$5.63 \cdot 10^{16}$	38.04	4.85	0.23	0.01	<0.01
100	$1.27 \cdot 10^{32}$	618.49	415.21	0.60	0.01	<0.01
150	$2.14 \cdot 10^{47}$	3424.10	1199.93	10.71	0.01	<0.01
200	$3.21 \cdot 10^{62}$	13751.07	12832.90	5.34	0.01	<0.01

$$G(G_{[0,2n-1]}(req_i \rightarrow (\neg ack_i \rightarrow X req_i)) \rightarrow (req_i \rightarrow F_{[0,2n-1]} ack_i))$$

Beim Beweis dieser Eigenschaft mit $i = 0$ für die verschiedenen Instanzen des Arbiters ergeben sich die in Tabelle 2 dargestellen Resultate. Es fällt auf, dass auf Grund der Komplexität dieser Eigenschaft die benötigte Laufzeit und der Speicherplatzzuwachs für den Beweis höher als bei den beiden anderen Eigenschaften ist. Deshalb ist es derzeit nicht möglich, mit den gegebenen Systemressourcen den Beweis dieser Eigenschaft für größere Instanzen des Arbiters durchzuführen.

Tabelle 2: Resultate beim Beweis der *Lebendigkeits*-Eigenschaft

Zellen	Zustände	Zeit EA	Zeit Beweis	Platz-zuwachs
2	8	0.01	0.01	0.01
3	24	0.01	0.01	0.08
4	64	0.02	0.61	0.18
5	160	0.06	7.07	1.12
6	384	0.08	98.46	4.78
7	896	0.09	1600.19	33.52
8	2048	0.14	8360.09	92.84
9	4608	0.19	57276.80	354.06

Mit Hilfe des vorgestellten Eigenschaftsprüfers können Spezifikationseigenschaften (gegeben als LTL-Formeln) für Schaltkreise, die in SystemC beschrieben sind, formal bewiesen werden.

6 Zusammenfassung und Ausblick

In dieser Arbeit wurde ein erster formaler BMC-basierter Verifikationsansatz zur Eigenschaftsprüfung für SystemC-Beschreibungen von Schaltkreisen vorgestellt. Die Eigenschaften werden dabei in LTL spezifiziert. Diese können im Gegensatz zu den bisher beschriebenen simulationsbasierten Verfahren vollständig bewiesen werden. Experimentell konnte gezeigt werden, dass das beschriebene Verfahren auf einem parametrisierten Arbiter mit verschiedenen zu überprüfenden Eigenschaften erfolgreich eingesetzt werden kann.

Es ist noch zu untersuchen, wie sich der Eigenschaftsprüfer auf anderen Schaltkreisen verhält. Ebenfalls lässt sich das vorgestellte BDD-basierte Beweisverfahren durch Verwendung alternativer Beweistechniken, wie z.B. eines SAT-Solvers, beschleunigen. Außerdem ist die Erweiterung des unterstützten SystemC-Sprachschatzes geplant.

Literatur

[1] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for the Constuction and Analysis of Systems*, volume 1579 of *LNCS*. Springer Verlag, 1999.

[2] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[3] R. Drechsler and D. Große. Reachability analysis for formal verification of SystemC. In *EUROMICRO*, pages 337–340, 2002.

[4] F. Ferrandi, M. Rendine, and D. Scuito. Functional verification for SystemC descriptions using constraint solving. In *Design, Automation and Test in Europe*, pages 744–751, 2002.

[5] T. Grötker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[6] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publisher, 1993.

[7] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Int'l Conf. on CAD*, pages 42–47, 1993.

[8] J. Ruf, D. W. Hoffmann, T. Kropf, and W. Rosenstiel. Simulation-guided property checking based on multi-valued AR-automata. In *Design, Automation and Test in Europe*, pages 742–748, 2001.

[9] F. Somenzi. *CUDD: CU Decision Diagram Package Release 2.3.1*. University of Colorado at Boulder, 2001.

[10] Synopsys Inc., CoWare Inc., and Frontier Design Inc., http://www.systemc.org. *Functional Specification for SystemC 2.0*.