SystemC Verifikation mittels symbolischer Simulation einer Zwischensprache

Hoang M. Le, Universität Bremen, 28359 Bremen, Deutschland
Daniel Große, solvertec GmbH, 28359 Bremen, Deutschland
Vladimir Herdt, Universität Bremen, 28359 Bremen, Deutschland
Rolf Drechsler, Universität Bremen und DFKI GmbH, 28359 Bremen, Deutschland

Kurzfassung

Beim Entwurf elektronischer Systeme ist die funktionale Verifikation eine zentrale Aufgabe. Insbesondere gilt es die Korrektheit verschiedener Modelle bereits auf hohen Abstraktionsebenen sicherzustellen, welche häufig in der Systembeschreibungssprache SystemC beschrieben werden. Nur so ist es möglich Fehler frühzeitig zu entdecken (und zu korrigieren) und so enorme Folgekosten zu vermeiden, die sonst durch das „Weitertragen“ eines Fehlers und seiner Auswirkungen (beispielsweise bei Verfeinerungsschritten) entstehen.
Die meisten Verifikationsmethoden für SystemC Modelle auf Transaktionsebene (TLM) basieren auf Simulation mit konkreten Werten. Sie können deshalb nur einen kleinen Teil der Funktionalität abdecken, d.h. es können weder alle Fehler entdeckt noch kann die Korrektheit nachgewiesen werden. Darüberhinaus gibt es eine überschaubare Anzahl von formalen Ansätzen, die zwar den Suchraum vollständig explorieren, aber größtenteils für komplexe Modelle nicht skalieren.
In dieser Arbeit stellen wir einen neuen Ansatz zur symbolischen Simulation für SystemC vor. Die Basis bildet eine kompakte Zwischensprache, auf die SystemC Modelle abgebildet werden können. Der entwickelte symbolische Simulator integriert zudem verschiedene Such- und Optimierungsstrategien zur Performanzsteigerung. Durch die vollständige Suchraumexploration ist der Simulator in der Lage, alle vorhandenen Fehler (u.a. Zusicherung- und Speicherzugriffsverletzungen) zu finden. Experimentelle Ergebnisse bestätigen die Überlegenheit unseres Verfahrens im Vergleich zu *State-of-the-Art* Ansätzen.

1 Einleitung

Elektronische Systeme bestehen heutzutage zunehmend aus einer Vielzahl von Hardware- und Softwarekomponenten. Um diese aufkommende Komplexität zu bewältigen, werden Systembeschreibungssprachen eingesetzt [1]. Sie ermöglichen die Systemmodellierung auf einer hohen Abstraktionsebene. Damit können verschiedene Entwurfsalternativen einfacher exploriert oder frühzeitig mit der Softwareentwicklung und Integration begonnen werden [1]. Die Systembeschreibungssprache SystemC [11] ist der Quasi-Standard zur Modellierung auf Systemebene. Ein SystemC Programm repräsentiert das Systemmodell und stellt damit die Grundlage des Entwurfablaufs dar. Es wird iterativ bis zu einem synthetisierbaren Modell verfeinert. Deshalb ist es wichtig, die korrekte Funktionalität des initialen SystemC Programms sicherzustellen, denn nicht entdeckte Fehler werden weiter propagiert [1]. Die frühzeitige Erkennung eines Fehlers im Entwicklungsablauf reduziert den nötigen Korrekturaufwand (Zeit und Kosten) erheblich.
Kommunikation und Synchronization auf der Systemebene sind in Form von abstrakten Operationen (Transaktionen) und Events modelliert. Dabei spielen Modellierungskonstrukte der niedrigeren Abstraktionsebenen wie Signale oder Leitungen keine Rolle. Deshalb lassen sich Verifikationsmethoden für diese Ebenen (siehe z.B. [15, 7]) nicht weiterverwenden. Auf der Systemebene existieren bereits verschiedene Ansätze zur Fehlersuche in SystemC Programmen. Die meisten Ansätze sind simulationsbasiert, z.B. [3, 5, 16]. Sie betrachten nur eine einzige der vielen vorhandenen Ausführungsfolgen der SystemC Prozesse und benötigen konkrete Eingabewerte. Aufgrund dieser Limitierungen eignen sie sich nur bedingt zur Fehlersuche und können die Abwesenheit von Fehlern nicht garantieren. Ein stärkerer Ansatz zur Fehlersuche wurde in [13, 2] vorgestellt. Im Vergleich zu den gerade genannten Simulationsverfahren werden alle Ausführungsfolgen der SystemC Prozesse betrachtet. Dabei werden sowohl statische als auch dynamische Optimierungsverfahren genutzt, um die redundanten Ausführungsfolgen zu reduzieren. Diese Ansätze arbeiten jedoch nach wie vor mit konkreten Werten. Sie benötigen damit repräsentative Eingabewerte und können damit nicht alle Spezialfälle abdecken. Deshalb können auch sie die Abwesenheit von Fehlern nicht garantieren. Um die Korrektheit ei-

Diese Arbeit wurde zum Teil vom Bundesministerium für Bildung und Forschung (BMBF) im Rahmen des SANITAS-Projekts unter der Vertragsnummer 16M3088 und von der Deutschen Forschungsgemeinschaft (DFG) im Rahmen des Reinhart Koselleck-Projekts DR 287/23-1 gefördert.

nes Programms zu zeigen, werden formale Methoden benötigt. Formale Verifikation von SystemC Modellen ist jedoch schwierig [18]. Die anfänglichen Ansätze [14, 12, 17, 10] skalieren nicht mit komplexen SystemC Programmen. Den *State-of-the-Art* im Verifikationsbereich von SystemC repräsentieren die Arbeiten [8, 4]. Das Tool Kratos in [4] nutzt ein symbolisches Model Checking Verfahren, welches auf Abstraktionsverfeinerung basiert und den SystemC Scheduler explizit integriert. In [8] werden Bounded Model Checking und induktionsbasierte Verfahren auf der Abstraktionsebene von C zum Nachweis *temporallogischer Eigenschaften* verwendet.

In dieser Arbeit wird ein neuer Ansatz zur symbolischen Simulation für SystemC Modelle auf hohen Abstraktionsebenen vorgestellt. Sie stellt einen Beitrag zur Sicherstellung der korrekten Funktionalität eines SystemC Programms dar. Dazu wurden im Rahmen dieser Arbeit ein *symbolischer* Simulator und eine Zwischensprache entwickelt. Ein SystemC Programm wird zunächst in ein Programm der Zwischensprache übersetzt, welches anschließend vom Simulator *symbolisch* ausgeführt wird. Die Zwischensprache bietet den Vorteil, dass der Simulator von C++ entkoppelt ist. Sie besitzt einen vergleichsweise kleinen Sprachumfang und erleichtert damit die Implementierung des Simulators. Ihr Sprachumfang setzt sich aus grundlegenden C++ und SystemC Konstrukten zusammen. Die Syntax und Semantik sind also bereits an SystemC angelehnt, wodurch eine Übersetzung erleichtert wird. Durch die Betrachtung aller möglichen Ausführungsfolgen und die Unterstützung symbolischer Ausdrücke ist eine vollständige Zustandsraumexploration mit dem entwickelten Simulator möglich.

Aufgrund der Größe des Zustandsraums sind Optimierungen ein wichtiger Bestandteil des Simulators. Unsere Zustandsraumoptimierungen versuchen die Anzahl der betrachteten Ausführungsfolgen der Threads und damit den durchsuchten Zustandsraum zu reduzieren, ohne das Ergebnis der Simulation zu verändern. Der Simulator implementiert zudem verschiedene Suchverfahren, um den Zustandsraum zu explorieren (Tiefensuche, Breitensuche, iterative Tiefensuche und Zufallssuche). Damit ergeben sich verschiedene Einstellungsmöglichkeiten mit denen der Simulator gestartet werden kann. Die besten Einstellungen werden anschließend als Vergleich mit dem *State-of-the-Art* im Bereich der formalen Verifikation von SystemC herangezogen.

Der Rest dieser Arbeit ist wie folgt gegliedert: Zunächst wird in Abschnitt 2 die entwickelte Zwischensprache vorgestellt. Aufbauend darauf wird beispielhaft ein SystemC Programm in Zwischensprache übersetzt, um ein besseres Verständnis zu vermitteln, welche Sprachkonstrukte unterstützt und wie sie abgebildet werden. Abschnitt 3 widmet sich der symbolischen Simulation. Hier geht es insbesondere um die Architektur und Funktionsweise des entwickelten Simulators, dazu gehören auch die angedeuteten Optimierungen. Darauf folgt eine Evaluation des entwickelten Simulators in Abschnitt 4. Abschließend werden die Ergebnisse der Arbeit zusammengefasst.

2 Zwischensprache

Die Zwischensprache wurde mit dem Ziel entwickelt, SystemC Programme darauf abbilden zu können und damit im Endeffekt SystemC Programme auf dem Simulator auszuführen. Sie bietet den Vorteil, dass der Simulator von der Sprache C++ entkoppelt ist. Somit können der Simulator und ein automatischer Übersetzer von SystemC zur Zwischensprache getrennt entwickelt werden. Die Zwischensprache ist von der Komplexität und vom Sprachumfang deutlich einfacher als C++. Damit wird ihre Analyse und Ausführung vereinfacht. So werden ausschließlich bedingte und unbedingte Sprünge zur Steuerung des Kontrollflusses verwendet, anstelle von Schleifen und If-Then-Else Anweisungsblöcken. Außerdem werden außer Funktionsblöcken keine weiteren Blöcke unterstützt. Der Vorteil dabei ist, dass ein einzelner Index ausreicht, um sich zu merken, welche Anweisung als nächstes ausgeführt werden soll. Der Index wird als *Instruktionszeiger* bezeichnet. Diese Entscheidung vereinfacht es, zwischen der Ausführung der einzelnen Threads zu wechseln.

Ein Programm in der Zwischensprache repräsentiert ein „aufgebautes“ SystemC Design (also nach Abschluss der Elaborationsphase). Es werden C++ Basisdatentypen, Funktionen, Arrays und in eingeschränkter Form auch Zeiger unterstützt. Funktionen können überladen werden und Standardparameter besitzen. Zudem öffnet jede Funktion analog zu C++ einen neuen Sichtbarkeitsbereich. Die Basisdatentypen sind *int* (signed oder unsigned und jeweils 8, 16, 32 oder 64 Bit breit) und *bool*. Ein konkreter oder symbolischer Wert eines Basisdatentyps repräsentiert einen atomaren Ausdruck. Die Zwischensprache unterstützt zudem die arithmetischen und Booleschen Operatoren der Sprache C++.

Die Kernelemente von SystemC, Threads und Events, sind fest in die Zwischensprache eingebaut und werden damit direkt unterstützt. SystemC Kanäle können auf bestehende Sprachkonstrukte abgebildet werden (als Kombination von Threads, Funktionen und Events).

2.1 Beispiel

Bild 1 zeigt ein Beispielprogramm in SystemC. Das Programm besteht aus einem Modul und drei Threads: A, B und C. Der Thread A setzt die Variable a in Abhängigkeit von x. Ist x durch 2 teilbar, dann wird a auf den Wert 0 gesetzt ansonsten auf 1. Die Variable x wird mit einem zufälligen Wert initialisiert (entspricht z.B. einer Eingabe). Der Thread B wartet zunächst auf die Benachrichtigung des Events e und setzt anschließend $b = x / 2$. Der Thread C führt eine unmittelbare Benachrichtigung des Events e aus. Falls B noch nicht darauf wartet, dann geht die Benachrichtigung verloren. Nach

```
1  SC_MODULE(Module) {
2    sc_core::sc_event e;
3    uint x, a, b;
4
5    SC_CTOR(Module)
6      : x(rand()), a(0)
7      , b(0) {
8      SC_THREAD(A);
9      SC_THREAD(B);
10     SC_THREAD(C);
11   }
12
13   void A() {
14     if (x % 2)
15       a = 1;
16     else
17       a = 0;
18   }
19
20   void B() {
21     e.wait();
22     b = x / 2;
23   }
24
25   void C() {
26     e.notify();
27   }
28 };
29
30 int sc_main() {
31   Module m("top");
32   sc_start();
33   assert(2 * m.b + m.a ==
34          m.x);
35   return 0;
36 }
```

Bild 1: Beispielprogramm in SystemC

```
1  event e
2  uint x = ?<uint>
3  uint a = 0
4  uint b = 0
5
6  sc_thread A begin
7    if x % 2 goto elseif
8      a = 0
9      goto endif
10   elseif:
11     a = 1
12   endif:
13 end
14
15 sc_thread B begin
16   wait e
17   b = x / 2
18 end
19
20 sc_thread C begin
21   notify e
22 end
23
24 sc_main begin
25   sc_start
26   assert 2 * b + a == x
27 end
```

Bild 2: Die Übersetzung des Beispielprogramms in der Zwischensprache

der Simulation sollte a den Wert $(x \ \% \ 2)$ und b den Wert $(x \ / \ 2)$ haben. Deswegen wird auch erwartet, dass die Zusicherung $(2*b+a == x)$ gilt (Zeile 33). Es existieren dennoch Gegenbeispiele, z.B. die Ausführungsfolge CAB führt dazu, dass die Zusicherung verletzt wird. Der Grund ist dass b nicht richtig gesetzt wird aufgrund der verlorengegangenen Benachrichtigung.
Bild 2 zeigt ein zugehöriges semantisch äquivalentes Programm in der Zwischensprache. Im Vergleich zum SystemC Programm ist das Modul „entpackt“, d.h. die Variablen und Funktionen der Klasse sind jetzt globale Deklarationen mit eindeutigem Namen. Die vorhandenen Threads sind in der Zwischensprache statisch vorgegeben. Die Aufrufe von *wait* und *notify* werden direkt auf die gleichnamigen Anweisung abgebildet. Der If-Then-Else Block aus Thread A wird als Kombination von bedingten und unbedingten Sprüngen umgesetzt (Zeile 7-12). Die Variable x wird mit einem symbolischen Wert initialisiert (Zeile 2). Sie kann jeden Wert aus dem Wertebereich des zugrundeliegenden Datentyps (hier *unsigned int*) annehmen. Der Einstiegspunkt des Programms ist *sc_main* (Zeile 24-27). Die Anweisung `sc_start` startet die Simulationsphase.

3 Symbolische Simulation

Der entwickelte Simulator führt eine symbolische Simulation von Programmen in der Zwischensprache durch und entdeckt dabei vorhandene Fehler. Zu einem Fehler gehören u.a. *Zusicherungsverletzungen*, *Division durch null oder Speicherzugriffsverletzungen*. Der Simulator ist in der Lage alle vorhandenen Fehler zu finden, und kann damit auch die Korrektheit eines Programms zeigen, solange die gesetzte Zeit- oder Speicherbeschränkung nicht überschritten wird. Die dafür benötigte vollständige Zustandsraumexploration wird durch die folgenden beiden Grundprinzipien der symbolischen Simulation erreicht:

- Anstelle ausschließlich mit konkreten Werten zu arbeiten, werden symbolische Werte verwendet. Wird ein Programm nur mit konkreten Eingabewerten getestet, dann erreicht man i.d.R. bereits bei kleineren Programmen nur eine unvollständige Testüberdeckung. Im Gegensatz dazu erhält man auf einem einzigen Ausführungspfad eine vollständige Testüberdeckung mittels symbolischer Eingabewerte. Eine einzelne Ausführung mit symbolischen Werten deckt also viele Ausführungen mit konkreten Werten ab.
- Der SystemC Simulationskernel betrachtet nur eine einzige Ausführungsreihenfolge der lauffähigen Threads. Damit durchsucht er nur einen (kleinen) Teil des vorhandenen Zustandsraumes, entsprechend werden oftmals Fehler übersehen. Die hier vorgestellte symbolische Simulation betrachtet alle Ausführungsreihenfolgen der Threads, soweit sie nicht als redundant identifiziert werden können.

Bild 3 zeigt den kompletten Suchbaum für das Beispielprogramm. Wie auf dem Bild zu erkennen gibt es bereits für dieses sehr einfachen Programm schon insgesamt 14 möglichen Ausführungspfade. Nur sechs davon enden mit einer Zusicherungsverletzung (durch die gefüllten Kasten gekennzeichnet). Die Knoten stellen die *Ausführungszustände* des Programms dar (Definition siehe unten). Die Beschriftung an einer Kante deutet an, welcher Thread bei diesem Zustandsübergang ausgeführt wurde. Im Folgenden werden die Grundimplementierung und die dazugehörigen Optimierungen im Zusammenhang mit Bild 3 näher erläutert.

3.1 Implementierung

Die Implementierung des Simulators besteht grundsätzlich aus drei Komponenten: einem Scheduler, einem Interpreter und einer SMT-Anbindung. Der Scheduler verwaltet die Zustandsmenge. Er ist dafür zuständig, den nächsten Ausführungszustand zu wählen und dann einen Thread, der in dem gewählten Zustand als lauffähig markiert ist. Der Interpreter kopiert diesen Zustand und führt dann den gewählten Thread auf dieser Kopie aus. Der Interpreter kann sowohl mit konkreten als auch mit symbolischen Werten arbeiten. Zur Lösung von symbolischen Ausdrücken, greift er über die SMT-Anbindung auf das SMT-Backend zu. Als SMT-Backend wird die Bibliothek MetaSMT [9] verwendet, um „automatisch“ von einer Vielzahl von SMT-Solvern zu profitieren.

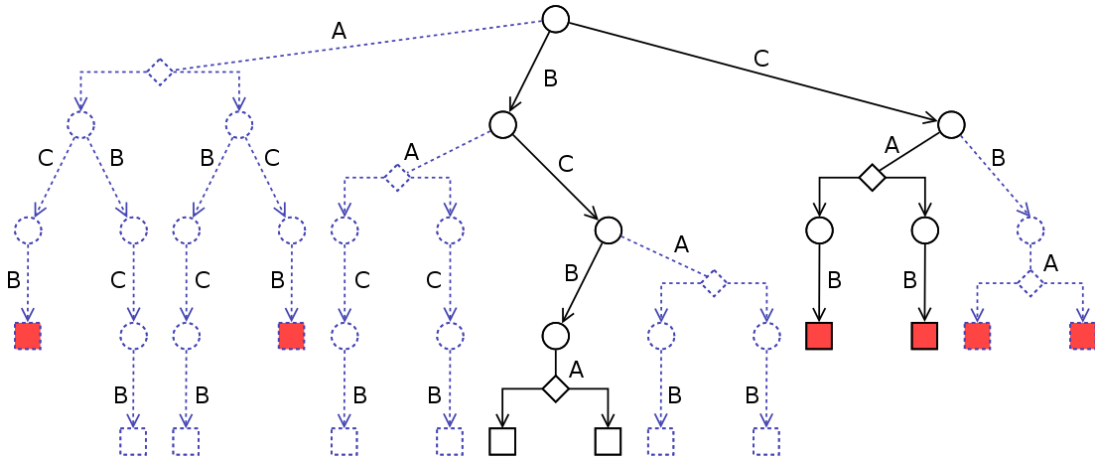

Bild 3: Suchbaum für das Beispiel

3.1.1 Ausführungszustand

Jeder Ausführungszustand enthält lokale und globale Variablen, die Instruktionszeiger, die Zustände einzelnen Threads (*lauffähig*, *blockiert* oder *terminiert*), sowie die zurzeit noch ausstehenden Events.
In unserem Beispiel enthält z.B. der initiale Zustand in Bild 3 (der oberste Knoten) die Initialisierungswerte der Variablen x, a und b; die *lauffähig*-Zustände der Threads A, B und C; Instruktionszeiger für A, B und C, die jeweils auf die erste Anweisung (Zeile 7, 16 und 21 in Bild 2) zeigen. Es gibt in diesem Zustand noch keine ausstehenden Events. Die Ausführung des Threads B ändert seinen Zustand zu *blockiert* und fügt e zu der Menge der ausstehenden Events hinzu. Außerdem wird der Instruktionszeiger für B aktualisiert, der nun auf Zeile 22 zeigt. Alle anderen Werte bleiben unverändert.

3.1.2 Scheduler

Die Aufgabe des Schedulers ist es, zunächst einen Ausführungszustand und dann einen der lauffähigen Threads auszuwählen. Der Interpreter führt dann den gewählten Thread aus und kehrt in einem von zwei Fällen zum Scheduler zurück:

1. Der ausgeführte Thread blockiert oder terminiert.
2. Der Thread führt einen bedingten Sprung mit einer symbolischen Bedingung aus, sodass die Bedingung und ihre Negation beide erfüllbar sind. In diesem Fall erzeugt der Interpreter einen neuen Ausführungszustand, damit beide Zweige des bedingten Sprunges unabhängig voneinander ausgeführt werden können. Außerdem werden beide Zustände als gestoppt markiert, um sie von den normalen Zuständen zu unterscheiden.

Im ersten Fall wird also ein neuer Zustand produziert und im zweiten Fall werden zwei neue Zustände produziert. In jedem Fall fügt der Scheduler die produzierten Zustände zu seiner Zustandsmenge hinzu.

Betrachten wir nun den siebten Ausführungspfad von links bzw. den ersten durchgezogenen Pfad in Bild 3. In dem Initialzustand wird der Thread B ausgewählt und ausgeführt. Dieser blockiert und deswegen wird ein neuer Zustand produziert. In diesem Zustand wird wiederum C ausgeführt. Dieser benachrichtigt das Event e und terminiert. Damit ist B wieder lauffähig in dem nächsten Zustand und wird auch ausgeführt. Nach der Terminierung von B wird A ausgeführt. Da A einen bedingten Sprung mit einer symbolischen Bedingung enthält, wird ein neuer Zustand $\diamond$ mit zwei Zweigen produziert. Der linke (rechte) Zweig entspricht dem Fall dass die Bedingung wahr (falsch) ist. In den beiden Zweigen gilt die Zusicherung.
Die Wahl des nächsten Ausführungszustands und Threads kann (großen) Einfluss auf den Simulationsverlauf nehmen (insbesondere, wenn das simulierte Programm Fehler enthält). Als Schedulingverfahren sind Tiefensuche, Breitensuche, und iterative Tiefensuche integriert.

3.1.3 Interpreter

Der Interpreter ist für die Ausführung der lauffähigen Threads, also ihrer einzelnen Anweisungen, in einem Ausführungszustand verantwortlich. Deshalb ist er auch dafür zuständig, Fehler zur Laufzeit zu entdecken. Jeder Thread kann beliebig viele Funktionen aufrufen und dann blockieren (indem eine blockierende Anweisung ausgeführt wird). Zu einem späteren Zeitpunkt kann der Scheduler diesen Zustand und denn blockierten Thread zur Ausführung selektieren. Der Interpreter ist in der Lage die Ausführung des Threads an der richtigen Stelle weiterzuführen. Instruktionszeiger und ein Callstack werden verwendet um die Ausführung von einem gestoppten Thread fortzusetzen.

3.2 Optimierungen

Der bisher beschriebene Scheduler exploriert alle möglichen Ausführungsreihenfolgen. Der damit durchsuch-

te Zustandsraum wächst i.d.R. exponentiell mit der Größe des analysierten Eingabeprogramms. Die Optimierungen setzen an diesem Problem an. Es wird versucht den explorierten Zustandsraum zu reduzieren dabei aber die Vollständigkeit der Suche beizubehalten, d.h. die optimierte Suche kommt immer zum selben Ergebnis wie die unoptimierte Suche, dabei durchsucht sie aber nach Möglichkeit weniger Zustände.
Die Grundidee ist es, äquivalente Ausführungsreihenfolgen nur einmal zu betrachten. Eine Ausführungsreihenfolge besteht aus einer Liste von Transitionen. Die Ausführung einer Transition überführt also einen Ausführungszustand in den nächsten. Eine *Transition* besteht aus einer Liste von Anweisungen, die ununterbrochen ausgeführt werden. Jeder Thread kann in eine Menge von Transitionen eingeteilt werden. Eine Transition ist eindeutig identifiziert durch ein Tupel bestehend aus einem Thread und einer Startanweisung. Ein Thread ist genau dann lauffähig, wenn eine seiner Transitionen lauffähig ist. Schließlich gibt es den Thread nur einmal, die Transitionen sind nur eine logische Gruppierung seiner Anweisungen. Zu jedem Zeitpunkt kann der Instruktionszeiger des Threads nur auf eine Anweisung zeigen.
Zwei Transitionen können jeweils paarweise auf Abhängigkeit verglichen werden. Daraus ergibt sich eine *Abhängigkeitsrelation*. Intuitiv sind zwei Transitionen abhängig, wenn die Reihenfolge ihrer Ausführung eine Auswirkung auf das Ergebnis der Simulation hat. Zwei Transitionen sind z.B. abhängig, wenn sie beide die gleiche Speicherstelle beschreiben, denn der Wert der Speicherstelle hängt von der Ausführungsfolge der beiden Transitionen ab. Ein anderes Beispiel ist ein wait-notify-Paar wie die beiden Threads B und C aus dem Programm in Bild 2 (Zeile 16 und 21).

3.2.1 Statische Optimierung

Die Abhängigkeitsrelation bildet die Grundlage, um äquivalente Ausführungsreihenfolgen zu identifizieren. Sie wird verwendet, um zur Laufzeit *Persistent-Sets* und *Sleep-Sets* [6] für die erreichten Zustände zu berechnen. Diese werden für jeden erreichten Zustand verwaltet und enthalten jeweils eine Teilmenge von lauffähigen Transitionen. Zukünftige Suchraumexploration wird auf diese Teilmengen eingeschränkt. Damit wird die Anzahl der betrachteten Ausführungsfolgen und damit der explorierte Zustandsraum reduziert. Die benötigte Abhängigkeitsrelation wird *statisch*, also vor der eigentlichen Simulationsausführung, aufgebaut. Durch diese statischen Optimierungen kann der Suchbaum in Bild 3 um die gestrichelte Pfade reduziert werden. Intuitiv lässt sich die Reduzierung wie folgt erklären. Weil Thread A unabhängig von B und C sind, ist es unwichtig wann A ausgeführt wird, also z.B. sind CAB, CBA und ACB äquivalent. Die Reihenfolge zwischen B und C ist dagegen wichtig aufgrund ihrer Abhängigkeit.

Tabelle 1: Vergleich mit Kratos (Laufzeit in Sek.)

	iter-ps	kratos
simple-fifo-c1-p2-bug	0,1998	423,7140
simple-fifo-c2-p1-bug	0,4607	293,3660
kundu-bug-1	0,0597	0,3880
kundu-bug-2	0,0723	1,0280
kundu-safe	15,6538	1,0600
toy-bug-1	0,1035	1,5801
toy-bug-2	0,0702	0,5560
toy-safe	0,1486	1,6281
mem-slave-tlm-safe.1	0,2016	2,9641
mem-slave-tlm-safe.2	0,2290	12,9488
mem-slave-tlm-safe.3	0,2721	147,477
mem-slave-tlm-safe.4	0,3112	timeout
mem-slave-tlm-safe.5	0,3539	timeout
transmitter-bug.13	0,1075	0,2280
transmitter-bug.20	0,1306	1,2480
transmitter-bug.50	0,3519	14,2609
transmitter-bug.100	0,9542	timeout
transmitter-bug.200	2,2820	timeout
token-ring-bug.13	0,0868	2,0201
token-ring-bug.15	0,1169	24,3135
token-ring-bug.18	0,1116	128,444
token-ring-bug.20	0,1370	timeout
token-ring-bug.200	3,0552	timeout

3.2.2 Dynamische Optimierung

Ein weiterer Ansatz ist die dynamische Backtracksuche. Abhängigkeiten werden dynamisch, also während der Simulation, berechnet und mit diesen Informationen wird die Anzahl der explorierten Ausführungsfolgen reduziert. Prinzipiell erkennt die dynamische Suche Abhängigkeiten genauer und kann damit den durchsuchten Zustandsraum stärker reduzieren. Dafür führt sie, im Vergleich zur statischen Suche, die einzelnen Transitionen langsamer aus.

4 Experimentelle Ergebnisse

Der entwickelte Simulator besteht aus mehreren Komponenten. Einige davon sind austauschbar oder optional. Zur Auswahl stehen verschiedene Scheduler, SMT-Solver, Optimierungen. Wir haben die einzelnen Komponenten evaluiert, und dabei hat sich eine überlegene Konfiguration herausgestellt: Iterative Tiefensuche in Kombination mit Persistent- und Sleep-Sets (*iter-ps*) mit Boolector als SMT-Solver. Entsprechend wird in diesem Abschnitt diese Kombination mit Kratos [4], einem der *State-of-the-Art* Ansätzen verglichen. Die Messwerte in Tabelle 1 wurden auf einem Linux (Debian) PC mit 1.4 GHz erstellt. Der entwickelte Simulator ist in Python (Version 2.7.3rc2) geschrieben. Die Suche wird auf maximal 600 Sekunden und 4GB Speicher beschränkt. Die Namen der Beispielprogramme enthalten das Wort *safe*, wenn das Programm fehlerfrei ist, ansonsten enthalten sie das Wort *bug*.
Die meisten der verwendeten Testprogramme sind Teil

der Kratos Distribution [4]. Einige Programme wurden leicht modifiziert. Kratos nutzt C als Eingabesprache, wobei verschiedene Konstrukte mittels Namenskonventionen identifiziert werden. Bei dem in dieser Arbeit entwickelten Ansatz wird eine Eigenentwicklung als Eingabesprache (siehe Abschnitt 2) verwendet. Es werden nur die Programme betrachtet, die sich in beide Richtungen haben übersetzen lassen.
Die Tabelle zeigt auf, dass der im Rahmen dieser Arbeit entwickelte symbolische Simulator gegenüber Kratos laufzeitmäßig meist überlegen ist. Besonders deutlich wird es bei den hochskalierten Beispielen: *transmitter-bug.200* und *token-ring-bug.200*. Es bleibt anzumerken, dass der Performanzunterschied nicht durch die Zwischensprache sondern nur durch die eingesetzten Beweistechniken zustande kommt.

5 Zusammenfassung

In dieser Arbeit ist ein Ansatz zur symbolischen Simulation von SystemC Programmen vorgestellt worden. Ein SystemC Programm wird zunächst in die Zwischensprache übersetzt und anschließend vom Simulator symbolisch ausgeführt. Der Simulator erlaubt eine *vollständige Zustandsraumexploration*. Damit kann er sowohl zur Fehlersuche als auch zur Verifikation eingesetzt werden. Um den Zustandsraum zu reduzieren, werden unterschiedliche Suchverfahren sowie statische und dynamische Optimierungen eingesetzt. Der entwickelte Simulator ist gegenüber Kratos [4], zurzeit einem der *State-of-the-Art* Model Checker für SystemC, in den meisten Beispielen laufzeitmäßig überlegen.

6 Literatur

[1] B. Bailey, G. Martin, and A. Piziali. *ESL Design and Verification: A Prescription for Electronic System Level Methodology*. Morgan Kaufmann/Elsevier, 2007.

[2] N. Blanc and D. Kroening. Race analysis for SystemC using model checking. *ACM Trans. on Design Automation of Electronic Systems*, 15:21:1–21:32, June 2010.

[3] N. Bombieri, F. Fummi, and G. Pravadelli. Incremental ABV for functional validation of TL-to-RTL design refinement. In *DATE*, pages 882–887, 2007.

[4] A. Cimatti, A. Griggio, A. Micheli, I. Narasamdya, and M. Roveri. Kratos - a software model checker for SystemC. In *CAV*, pages 310–316, 2011.

[5] L. Ferro and L. Pierre. ISIS: Runtime verification of TLM platforms. In *FDL*, pages 1–6, 2009.

[6] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Springer, 1996.

[7] D. Große and R. Drechsler. *CheckSyC*: An efficient property checker for RTL SystemC designs. In *ISCAS*, pages 4167–4170, 2005.

[8] D. Große, H. M. Le, and R. Drechsler. Proving transaction and system-level properties of untimed SystemC TLM designs. In *MEMOCODE*, pages 113–122, 2010.

[9] F. Haedicke, S. Frehse, G. Fey, D. Große, and R. Drechsler. metaSMT: Focus on your application not on solver integration. In *DIFTS'11: 1st International workshop on design and implementation of formal tools and systems*, pages 22–29, 2011.

[10] P. Herber, J. Fellmuth, and S. Glesner. Model checking SystemC designs using timed automata. In *CODES+ISSS*, pages 131–136, 2008.

[11] IEEE Std. 1666. *IEEE Standard SystemC Language Reference Manual*, 2011.

[12] D. Karlsson, P. Eles, and Z. Peng. Formal verification of SystemC designs using a petri-net based representation. In *DATE*, pages 1228–1233, 2006.

[13] S. Kundu, M. Ganai, and R. Gupta. Partial order reduction for scalable testing of SystemC TLM designs. In *DAC*, pages 936–941, 2008.

[14] M. Moy, F. Maraninchi, and L. Maillet-Contoz. LusSy: an open tool for the analysis of systems-on-a-chip at the transaction level. *Design Automation for Embedded Systems*, pages 73–104, 2006.

[15] J. Ruf, D. W. Hoffmann, T. Kropf, and W. Rosenstiel. Simulation-guided property checking based on multi-valued ar-automata. In *DATE*, pages 742–748, 2001.

[16] D. Tabakov and M. Y. Vardi. Monitoring temporal SystemC properties. In *MEMOCODE*, pages 123–132, 2010.

[17] C. Traulsen, J. Cornet, M. Moy, and F. Maraninchi. A SystemC/TLM semantics in promela and its possible applications. In *SPIN*, pages 204–222, 2007.

[18] M. Y. Vardi. Formal techniques for SystemC verification. In *DAC*, pages 188–192, 2007.