Optimierte Generierung von Konformitätstests für eingebettete Echtzeitsysteme

Marcel Pockrandt
Technische Universität Berlin
www.pes.tu-berlin.de

Zusammenfassung

Die Qualitätssicherung eingebetteter Echtzeitsysteme ist zumeist sehr aufwändig. Zwar existieren Möglichkeiten zur Verifikation, diese sind allerdings in den meisten Fällen nicht auf die konkrete Implementierung anwendbar. Um sicherzustellen, dass eine Implementierung ihre Spezifikation umsetzt, bieten sich Konformitätstests an. In diesem Papier stellen wir verschiedene Optimierungen eines Ansatzes zur Generierung von Konformtätstests für SystemC vor und erweitern ihn um die Möglichkeit zur automatischen Generierung von Testbenches. Diese erlauben die vollautomatische Bewertung der Konformität verfeinerter SystemC Modelle zu einem abstrakten Entwurf. Mit Hilfe von Experimenten zeigen wir die Fähigkeit, nichtkonformes Verhalten aufzufinden, und die Performanz unseres Ansatzes.

1 Motivation

Eingebettete Systeme werden häufig für sicherheitskritische Anwendungen verwendet, in denen Fehlfunktionen hohe finanzielle Kosten oder sogar die Gefährdung von Personen zur Folge haben können. Die Qualitätssicherung solcher Systeme spielt daher eine sehr wichtige Rolle. Besonders wünschenswert sind vollständige Verifikationstechniken, mit denen wichtige System-Eigenschaften für jede mögliche Eingabe überprüft werden können. Diese sind allerdings im Allgemeinen nur auf abstrakte Modelle und nicht auf die konkrete Implementierung anwendbar. Um zu überprüfen, ob eine Implementierung Eigenschaften einhält, die auf dem abstrakten Modell verifiziert wurden, können Konformitätstests verwendet werden. In [1] wird ein Ansatz zur Generierung von Konformitätstests für SystemC [2] vorgestellt. Dazu wird ein abstraktes SystemC Modell zunächst in ein semantisch äquivalentes UPPAAL Modell transformiert. Aus diesem werden dann durch eine Traversierung des Zustandsraums für einen gegebenen Eingabe-Trace alle möglichen Ausgabe-Traces berechnet. Die berechneten Traces können anschließend verwendet werden, um zu prüfen, ob ein verfeinertes SystemC-Modell Traces liefert, die auch im abstrakten Modell möglich sind. Allerdings ist dieser Ansatz sehr aufwändig und lässt sich nicht auf komplexe Modelle anwenden. Zudem erlaubt er keine vollautomatische Konformitätsbewertung.
Wir stellen daher Optimierungen des Ansatzes vor, die die Semantik von SystemC ausnutzen, um den Zustandsraum signifikant zu verkleinern. Weiterhin präsentieren wir einen Ansatz zur vollautomatischen Generierung von SystemC Testbenches. Eine ausführlichere Darstellung der Ergebnisse ist in [3] zu finden.

Im Folgenden geben wir zunächst in Abschnitt 2 einen Überblick über verwandte Arbeiten, stellen in Abschnitt 3 die bei der Erstellung der Timed Traces durchgeführten Optmierungen vor und erläutern in Abschnitt 4 die Generierung der SystemC Testbenches. Abschließend werden in Abschnitt 5 Ergebnisse verschiedener Experimente präsentiert und in Abschnitt 6 die Ergebnisse dieser Arbeit zusammengefasst.

2 Verwandte Arbeiten

Es existieren verschiedene Ansätze zur Generierung von Konformitätstests für Echtzeitsysteme. In [4] wird UPPAAL verwendet, um eine minimale Testsuite mit kompletter Pfadüberdeckung

zu erstellen. Einen ähnlichen Ansatz verfolgt [5], wobei hier zusätzliche Abstraktionsmechanismen verwendet werden, um die Testsuite weiter zu reduzieren. Beide Ansätze sind jedoch auf deterministische Systeme beschränkt, genau wie das UPPAAL CoVer[6] Tool. Im Gegensatz dazu ermöglicht UPPAAL TRON[6, 7] auch die Verwendung nichtdeterministischer Modelle, erzeugt die Testfälle allerdings online, wodurch die Wiederverwendung in späteren Entwicklungsphasen nicht möglich ist. Der in [1] vorgestellte und von uns in diesem Papier weiterentwickelte Ansatz ist in der Lage, nichtdeterministische Modelle zu bearbeiten und daraus offline Konformitätstests zu erzeugen, die über den gesamte Entwicklungsprozess verwendet werden können.

3 Optimierte Traceerzeugung

Obwohl der in [1] vorgestellte Algorithmus in der Lage ist, das Verhalten eines Modells zu extrahieren und in Form von Timed Traces auszugeben, gibt es Einschränkungen, die die Anwendbarkeit auf komplexe Modelle verhindern. Zum Einen hat der ursprüngliche Algorithmus einen großen Speicherverbrauch, der bei komplexen Modellen zu Fehlern führt und einen erheblichen Einfluss auf die Laufzeit hat. Zum Anderen enthielten die generierten Traces eine große Anzahl nicht unterscheidbarer Zustände. Um die Generierung der Testbenches auch für komplexere Modelle zu ermöglichen, muss demnach zunächst die Generierung der Timed Traces sowohl hinsichtlich des Speicherverbrauchs als auch hinsichtlich der unterscheidbaren Zustände optimiert werden.

3.1 Zustandsoptimierungen

Die SystemC Testbenches können lediglich Änderungen an nach außen sichtbaren Variablen des zu testenden Systems überwachen. Aus diesem Grund haben wir zwei Verfahren zur Reduktion der Timed Traces entwickelt. Diese nutzen die Besonderheiten der SystemC Semantik aus und ermöglichen eine drastische Reduzierung des Zustandsraums.

Linear Time Chain Reduction (LTCR) Ein Großteil nicht unterscheidbaren Zustände besteht aus einer unverzweigten Kette von Zuständen, die alle die gleiche Variablenbelegung und überlappende Uhrenbedingungen besitzen. Solche Zustandsketten können wir zu einem einzigen Zustand zusammenfassen.

Branch Reduction (BR) Auch Zustände mit mehreren Nachfolgern können eventuell nicht von diesen unterschieden werden. In solch einem Fall können wir den Ausgangszustand mit seinen Nachfolgern zusammenfassen, wenn der Ausgangszustand und alle seine Nachfolgezustände identische Uhrenbedingungen und Variablenbelegungen besitzen.

3.2 Speicheroptimierungen

Der ursprüngliche Algorithmus musste sämtliche Zustände im Speicher halten, um überprüfen zu können, ob ein neu berechneter Zustand bereits vorher berechnet worden ist. Erst nach der Berechnung aller Zustände wurden diese als Trace gespeichert. Zur Optimierung haben wir Bitstate Hashing verwendet, um Zustände eine möglichst kurze Zeit im Speicher halten zu müssen und somit die Speicherlast zu reduzieren. Als Resultat müssen zu jedem Zeitpunkt maximal zwei Zustandsgenerationen im Speicher gehalten werden, wodurch der Speicheraufwand nicht mehr von der Gesamtkomplexität des Systems, sondern nur noch von der Anzahl der Nachfolgezustände einer Zustandsgeneration abhängig ist. Weiterhin haben wir eine Auslagerungsstrategie auf die Festplatte entwickelt, mit der der Algorithmus auch auf Zustände mit sehr vielen Nachfolgezuständen anwendbar ist. Desweiteren haben wir noch eine Reihe von kleineren Verbesserungen an der Zustandsrepräsentation vorgenommen, die den Speicherbedarf eines einzelnen Zustandes ebenfalls signifikant verringert haben.

4 Testbenchgenerierung

Der in [1] vorgestellte Ansatz zur Generierung von Konformitätstests arbeitet zwar vollautomatisch, allerdings benötigt die anschließende Konformitätsbewertung ein hohes Maß an manuellem Aufwand. Um eine möglichst große Automatisierung zu erreichen, wollen wir aus den Timed Traces SystemC Testbenches generieren, die dann eine vollautomatische Konformitätsbewertung des zu testenden Systems (SUT) ermöglichen. Die Timed Traces werden hierbei als Akzeptanzgraph interpretiert. Die Ausgabe des SUT wird dann daraufhin überprüft, ob innerhalb der durch den Akzeptanzgraph definierten Zeitschranken Zustandsübergänge stattfinden. Um das SUT testen zu können, muss die generierte Testbench sämtliche Ausgangsports des SUT überwachen.

Erhält die Testbench einen vollständigen Ausgabetrace, der zu einem Endzustand des Akzeptanzgraphs führt, so liefert die Testbench ein *pass* zurück. Falls während des Tests ein nicht im Akzeptanzgraph vorhandener Zustand erreicht wird, sei es durch eine inkorrekte Variablenbelegung oder durch das Verletzen von Zeitschranken, so wird stattdessen ein *fail* zurückgegeben. Hierbei stellten insbesondere blockierende Ports ein Problem dar. Ein Prozess, der auf Änderungen an einem solchen Port reagieren soll, blockiert, bis eine Änderung stattgefunden hat. Ein Prozess, der mehrere blockierende Ports überwachen soll, könnte dementsprechend Änderungen an einem Port verpassen, da er gerade auf Änderungen an einem anderen Port wartet. Wir umgehen diese Problematik dadurch, dass wir für jeden blockierenden Port einen eigenen Monitor-Prozess erstellen. Ebenso konnten die zu testenden Systeme nichtdeterministisches Verhalten zeigen, wodurch die Testbench gegebenfalls nicht feststellen konnte, in welchem Zustand sich das SUT aktuell befindet. Aus diesem Grund arbeitet unsere Testbench mit einer Menge möglicher Zustände, in denen sich das SUT befinden kann. Durch Beobachtung des weiteren Systemverhaltens kann sich im Testverlauf die Menge der möglichen Zustände vergrößern und verkleinern.

Die Testbench selbst besteht aus mehreren Komponenten. Die Lookup-Table (LUT) stellt den Akzeptanzgraphen dar. Sie beinhaltet für jeden Zustand s die Belegungen aller Variablen $(v_0, ..., v_n)$, eine obere und eine untere Zeitschranke, die beschreibt, wann dieser Zustand aktiv sein kann (g_l, g_u) sowie alle von diesem Zustand aus erreichbaren Nachfolgezustände $(s_0, ..., s_m)$. Zur Überwachung der Ports existieren zwei verschiedene Arten von Monitorprozessen. Der Nonblocking Monitor überwacht sämtliche nichtblockierenden Ports des SUT, während wir für jeden blockierenden Port einen separaten Blocking Monitor benötigen. Registriert einer der Monitore eine Änderung, so hat ein Zustandswechsel stattgefunden. Anschließend prüft der Monitor mit Hilfe der LUT sämtliche direkten Nachfolgezustände des aktuellen Zustands, um herauszufinden, in welchem Zustand sich das System jetzt befindet. Falls kein möglicher Folgezustand gefunden wird, befindet sich das System in einem nicht konformen Zustand, und es wird ein *fail* ausgegeben. Der Timing Monitor überwacht den zeitlichen Ablauf der Zustandsänderungen. Er wird von den anderen Monitoren bei Zustandsänderungen benachrichtigt und überprüft dann, ob die als mögliche Folgezustände markierten Zustände auch die in der LUT definierten Zeitschranken einhalten. Zusätzlich stellt er sicher, dass ein *fail* ausgegeben wird, falls sich das SUT zu lange in einem Zustand befindet.

5 Experimente

Um den Erfolg des vorgestellten Verfahrens zu zeigen, präsentieren wir im Folgenden die Ergebnisse verschiedener Experimente. Hierbei verwenden wir drei verschiedene Modelle. Das Producer-Consumer-Beispiel (ProdCon) beinhaltet zwei Prozesse, die über einen FIFO miteinander kommunizieren. PSwitch implementiert einen Packet-Switch und wurde von uns mit unterschiedlichen Anzahlen an schreibenden und lesenden Prozessen verwendet. Das dritte Modell ist ein Anti-Blockiersystem (ABS), das in einem Studentenprojekt entwickelt wurde und deutlich umfangreicher ist.

5.1 Speicheroptimierungen

Die in Abschnitt 3.1 vorgestellten Speicheroptimierungen wurden von uns direkt in das in [1] vorgestellte ATENA-Tool (Automatic Transformation Engine for Nondeterministic Timed Automata) integriert. Ein Vergleich mit der ursprünglichen Version zeigt eine signifikante Verbesserung des Speicherbedarfs und daraus resultierend auch eine deutliche Verbesserung der Laufzeit bei Modellen, die mit beiden Varianten benutzbar waren. Zusätzlich ermöglichen uns die Optimierungen auch die Anwendung von ATENA auf Modelle, bei denen dies vorher aufgrund von Speicherknappheit nicht möglich war. Tabelle 1 vergleicht die Berechnungszeit sowie den Speicherverbrauch des ursprünglichen Algorithmus (Base) mit dem von uns optimierten Algorithmus (Opt).

5.2 Testbenchgenerierung

Die in Abschnitt 4 vorgestellten Testbenchmodule können mit dem von uns entworfenen TBGeneSys-Tool (TestBenchGenerator for SystemC) vollautomatisch aus den Timed Traces der ATENA Ausgabe generiert werden. Während des Einlesens der Traces werden gleichzeitig die in 3.1 beschriebenen Reduktionsverfahren angewendet. Um den Erfolg der Testbenchgenerierung zu

Tabelle 1: Ergebnisse der Optimierungen

	CPU Zeit (s)			**RAM** (MB)		
	Base	Opt	Impr (%)	Base	Opt	Impr (%)
ProdCon	4.90	5.07	- 3.5	5	5	0
PSwitch						
1m1s	25.11	9.49	62.2	58	5	91.4
1m2s	34.27	13.90	59.4	98	5	94.9
2m1s	42.38	20.72	51.1	160	5	96.9
2m2s	54.77	27.43	49.9	275	13	95.3
4m4s	↯	443	∞	↯	302	∞
ABS	↯	10210	∞	↯	302	∞

Tabelle 2: Ergebnisse der Konformitätstests

	ProdCon	PSwitch	ABS
Selbsttest	*pass*	*pass*	*pass*
korrekt verfeinert	-	-	*pass*
fehlende/geänderte Bedingung	*pass*/*fail*	*fail*	*fail*
fehlende/geänderte Zuweisung	*pass*/*fail*	*fail*	*fail*
permutierte Variablen	*fail*	*fail*	*fail*
verspätetes oder fehlendes Signal	*fail*	*fail*	*fail*

testen, haben wir aus verschiedenen SystemC-Modellen Testbenches generiert. Diese Testbenches wurden anschließend mit unterschiedlichen Varianten des ursprünglichen Modells getestet. Die Ergebnisse sind in Tabelle 2 zu sehen. Der Selbsttest (Modell mit aus sich generierter Testbench) ergab in allen Fällen eine Konformität. Auch ein korrekt verfeinertes Modell wurde als konform zum abstrakten Modell erkannt. Ebenso wurden die meisten Modelle mit injizierten Fehlern als nicht-konform erkannt. Lediglich bei dem Producer Consumer Beispiel konnten nicht alle Fehler erkannt werden. Dies liegt darin begründet, dass der Eingabe-Trace, für den die Konformitätstests generiert wurden, das Modell nicht vollständig überdeckte. Die Änderungen an nicht überdeckten Codeabschnitten hatten dann auch keinen Einfluss auf das Systemverhalten.

6 Fazit

In diesem Papier haben wir einen effizienten und vollautomatisch durchführbaren Ansatz zur Konformitätsbewertung von SystemC Modellen vorgestellt. Der von uns optimierte Algorithmus ist in der Lage, auch für komplexe und nichtdeterministische Modelle offline Konformitätstests zu generieren. Zusätzlich haben wir einen Ansatz zur automatischen Generierung von SystemC Testbenches vorgestellt, der eine vollautomatische Konformitätsbewertung erlaubt. Unsere Experimente zeigen sowohl ein signifikant besseres Laufzeitverhalten und einen deutlich geringeren Speicherverbrauch des optimierten Algorithmus als auch die Fehlererkennungsmöglichkeiten der generierten Testbenches.

Danksagung

Der Autor möchte sich bei Sabine Glesner und Paula Herber für die Unterstützung bei der hier vorgestellten Arbeit bedanken.

Literatur

[1] P. Herber, F. Friedemann, and S. Glesner, "Combining Model Checking and Testing in a Continuous HW/SW Co-Verification Process," in *Tests and Proofs*, ser. LNCS, vol. 5668. Springer, 2009.

[2] IEEE Standards Association, "IEEE Std. 1666–2005, Open SystemC Language Reference Manual," 2005.

[3] P. Herber, M. Pockrandt, and S. Glesner, "Automated Conformance Evaluation of SystemC Designs using Timed Automata," in *IEEE European Test Symposium*, 2010.

[4] J. Springintveld, F. Vaandrager, and P. R. D'Argenio, "Testing timed automata," *Theoretical Computer Science*, vol. 254, 2001.

[5] R. Cardell-Oliver, "Conformance tests for real-time systems with timed automata specifications," *Formal Aspects of Computing*, vol. 12(5), pp. 350–371, 2000.

[6] A. Hessel, K. G. Larsen, M. Mikucionis, B. Nielsen, P. Pettersson, and A. Skou, *Formal Methods and Testing*. Springer, 2008, ch. Testing Real-Time Systems Using UPPAAL.

[7] K. G. Larsen, M. Mikucionis, and B. Nielsen, *Formal Approaches to Software Testing*. Springer, 2005, ch. Online Testing of Real-time Systems Using UPPAAL, pp. 79–94.