Variablenelimination für symbolische Modelle

Dirk Richter, Wolf Zimmermann (richterd|zimmer@informatik.uni-halle.de)
Martin-Luther-Universität Halle-Wittenberg

Abstract: Zur Software-Modell-Prüfung, zum modellbasierten Testen und bei der Testdaten- und Codegenerierung sind die Größe und Komplexität von Modellen entscheidende Einflussfaktoren. Aus Quellcode (z.B. C oder Java) gewonnene Modelle in Form von symbolischen Kellersystemen (SPDS) erlauben nicht nur präzisere Ergebnisse, sondern führen auch ohne Modellexplosion bei **exakter** Nachbildung von Rekursion zu weniger Fehlalarmen. Für diese SPDS wurde ein Ansatz verfolgt, der die innere Struktur der Zustände ausnutzt, um den Zustandsraum der Modelle weiter zu verkleinern. Experimente zeigen, dass damit die Modellprüfung beschleunigt bzw. die Modellprüfung erst ermöglicht wird oder sich erübrigt.

Schlüsselworte: Kellersystem, Modellanalyse, Remopla, Moped, Softwaremodellprüfung

1 Einleitung

Im Gegensatz zu vergleichbaren Arbeiten bei 'Finite-State' Modellprüfern wie BLAST, SPIN, NuSMV/SMV, JavaPathFinder, F-Soft oder Bogor (Bandera Projekt) beschäftigen wir uns mit der Modellverbesserung **unendlicher** symbolischer Modelle. Viele Modellprüfer beschränken die Rekursionstiefe oder verbieten Methodenaufrufe. Durch diese Unter- bzw. Überapproximation von Methodenaufrufen enstehen Fehlalarme (False Negatives sowie Fehlabstraktionen), die durch korrekte Abbildung von Methodenaufrufen und Rekursion auf SPDS vermieden werden können. Die Beschränkung auf eine maximale Anzahl an Methodenaufrufen ist dann nicht nötig und vermeidet eine exponentielle Modellvergrößerung z.B. durch Inlining. Solche SPDS können mittels JMoped [SSE05] aus Java gewonnen und mittels des Modellprüfers Moped [ES01, Sch02, SSE05] überprüft werden (siehe Abbildung 1) und ermöglichen mächtigere interprozedurale und kontextsensitive Modell-Analysen, -Tests und -Prüfungen. Unter Verwendung des Cross-Compilers Grasshopper kann nicht nur Java 1.6 Code verwendet werden, sondern auch Microsoft Intermediate Language. Es ist auch möglich, die Gültigkeit von Java Modeling Language (JML) Annotationen zu überprüfen, wenngleich dies in der Praxis derzeit noch unhandlich ist.

Ziel dieser Arbeit ist die Optimierung solcher SPDS Modelle mittels Verkleinerung des Zustandsraums. Um SPDS Modelle zu optimieren, sind Informationen über das Modellverhalten wichtig, die durch im Programmiersprachenumfeld gängige Programmanalysen gewonnen werden können. Bei einer sog. Äquivalenzanalyse werden z.B. durch Verwendung von Konstantenpropagation und Konstantenfaltung sowie Copy-Propagation Gleichheit und Konstanz von Variablen erkannt. Liang und Harrold zeigen in [LH99, LH03],

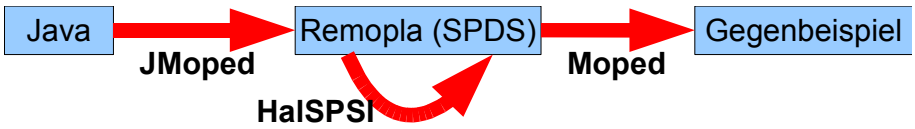

Abbildung 1: Einsatz unseres HalSPSI zur Optimierung von SPDS Modellen

dass bessere Äquivalenzanalysen zu besserem Slicing sowie besseren Zeiger- und Datenflussanalysen führen. In [RZ07] und [Ric08] wurde gezeigt, dass verschiedene Modellanalysen im Gegensatz zur Anwendung bei herkömmlichen Programmiersprachen **entscheidbar** werden, was prinzipiell die Existenz exakter Modellanalyseverfahren für SPDS nachweist. Dabei heißt eine Modellanalyse **exakt**, wenn das Analyseergebnis weder eine Über- noch eine Unterapproximation darstellt, also präzise das Verhalten des Modells berücksichtigt. Z.B. führt die Verknüpfung von Konstantenpropagation und Konstantenfaltung sowie Copy-Propagation zu einer konservativen Äquivalenzanalyse. Bei dieser kann es Variablen geben, die zwar gleich oder gar konstant sind, dies aber nicht entdeckt wird. Hier wird nun betrachtet, wie gegebene Äquivalenzanalysen für ein SPDS genutzt werden können, um SPDS durch sog. Variablenelimination zu vereinfachen.

2 Symbolische Kellersysteme

$M = (S, \rightarrow, L_A)$ heißt **Kripkestruktur**, falls S und A (nicht notwendigerweise endliche) Mengen sind, $\rightarrow \subseteq S \times S$ und $L_A : S \rightarrow 2^A$. Bei gegebener Kripkestruktur M ist das **Erreichbarkeitsproblem** die Frage, ob es in M einen Pfad von einem Zustand $s \in S$ zu einem anderen Zustand $z \in S$ gibt ($s \xrightarrow{*} z$?). Zur Beschreibung von (unendlich) großen Kripkestrukturen können Kellersysteme (Pushdown Systems) verwendet werden. $\mathcal{P} = (P, \Gamma, \hookrightarrow)$ heißt **Kellersystem**, falls P eine Menge von Zuständen, Γ eine endliche Menge (Kelleralphabet) und $\hookrightarrow \subseteq (P \times \Gamma) \times (P \times \Gamma^*)$ eine Menge von Transitionen ist. Informell ist ein Kellersystem ein Kellerautomat ohne Eingabe. Mit Software-Modellprüfung solcher Kellersysteme kann die Abwesenheit von Fehlern in Kellersystemen formal nachgewiesen werden [Sch02, ES01, Ber06, EKS02, EHRS00, Wal00, BEM97], was im Gegensatz zum Testen nicht möglich ist, da dort nur die Anwesenheit von Fehlern festgestellt werden kann, sofern kein vollständiges Testen wie beim JavaPathFinder 4 [VHB+03] durchgeführt wird. Allerdings ist die Software-Modellprüfung im Gegensatz zum nichtvollständigen Testen sehr viel aufwändiger, was den Einsatz in der Praxis erschwert. Unter Anderem liegt dies an dem bei der Modellprüfung wohl bekannten Problem der Zustandsraumexplosion. Da Kellersysteme ihrerseit per Definition über Zustände verfügen, bezeichnen wir diesen Zustandsraum als Konfigurationenraum, wobei (p, v) **Konfiguration** heißt, falls $p \in P$ und $v \in \Gamma^*$. (p, a) heißt **Kopf** der Konfiguration (p, aw), falls $a \in \Gamma$ und $w \in \Gamma^*$. Auf Konfigurationen wird die Transitionsrelation $\hookrightarrow$ erweitert zu $\rightarrow \subseteq (P \times \Gamma^*) \times (P \times \Gamma^*)$ mit $(p, aw) \rightarrow (q, bw) :\Leftrightarrow (p, a) \hookrightarrow (q, b)$. Mit $\xrightarrow{*}$ wird die reflexive und transitive Hülle von $\rightarrow$ bezeichnet. Bei einem **Symbolischen Kellersystem** (SPDS) werden die Transitionen nur indirekt (symbolisch) mittels Relationen beschrieben, was die Angabe des vollständi-

gen Kellersystems vereinfacht [Sch02].

Wie in den Abbildungen 2 und 3 zu sehen, können SPDS mit Hilfe der Modellsprache Remopla [KSS06] sehr kompakt beschrieben werden. Remopla ist zwar syntaktisch ähnlich zu Promela (Eingabesprache für den SPIN Modellprüfer), unterstützt aber keine parallelen Prozesse, dafür synchron parallele Konfigurationenübergänge und exakte Rekursion. Exakte Rekursion bedeutet hier, dass die in einem Modell enthaltene Methodenaufrufe während der Modellprüfung weder unter- noch überapproximiert werden, sondern analog dem Laufzeitsystem moderner Programmiersprachen in einem Keller verwaltet werden. Neben lokalen Variablen loc_q und Parametern $pars_q \subseteq loc_q$ eines Moduls q (auch Prozedur genannt), können in Remopla auch globale Variablen $globs$ sowie Arrays deklariert werden. Die lokalen Variablen und Methodenparameter werden in SPDS als Bitvektoren über dem Kelleralphabet zusammen mit der Aufrufhierarchie im Keller repräsentiert. Globale Variablen (in Java Klassenvariablen), die Halde (Heap) sowie Ausnahmen (Exceptions) werden mit Hilfe der Zustände eines Kellersystems beschrieben. Ziel dieser Arbeit ist es, das dazu nötige Kelleralphabet und die benötigten Kellerzustände bereits symbolisch a priori zu verringern. So konstruierte Modelle können dann durch den von uns entwickelten Optimierer **HalSPSI** verbessert (Verkleinerung des Zustandsraums) und anschließend durch den Modellprüfer Moped geprüft werden.

Seien $Vars_q := globs \cup loc_q$, $EXPR_{Vars_q}$ arithmetische Ausdrücke über diesen Variablen und $f^q : Vars_q \to \mathbb{N}$ eine Variablenbelegung, welche aus dem Kopf einer Konfiguration bestimmt wird, sowie $[\![e]\!]_{f^q}$ die Auswertung eines Ausdrucks $e \in EXPR_{Vars_q}$ mittels der Variablenbelegung f^q. Dann sind die wichtigsten Remopla-Anweisungen:

- $\mathbf{x}_1 = \mathbf{e}_1, \mathbf{x}_2 = \mathbf{e}_2, \ldots, \mathbf{x}_n = \mathbf{e}_n$**;** mit $\mathrm{x}_i \in Vars_q$ und $\mathrm{e}_i \in EXPR_{Vars_q}$ ein synchron paralleler Konfigurationenübergang, welcher jeder Variablen x_i den ausgewerteten Ausdruck $[\![e_i]\!]_{f^q}$ zuweist.
- **p(**$\mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_n$**);** mit $\mathrm{e}_i \in EXPR_{Vars_q}$ ein Modulaufruf an das Modul p mit Call-By-Value Semantik.
- **return e;** mit $\mathrm{e} \in EXPR_{Vars_q}$ ein Modulende mit Rückgabewert $[\![e]\!]_{f^q}$.
- **goto L;** ein unbedingter Sprung an die Marke L.
- **if ::** $\mathbf{b}_1 -> \mathbf{s}_1$**; ::** $\mathbf{b}_2 -> \mathbf{s}_2$**;** $\ldots$ **::** $\mathbf{b}_n -> \mathbf{s}_n$**; fi;** eine bedingte Anweisung mit $\mathrm{b}_i \in EXPR_{Vars_q}$ und $[\![\mathrm{b}_i]\!]_{f^q} \in \{0, 1\}$, die eine zufällig ausgewählte Anweisung s_k mit $[\![\mathrm{b}_k]\!]_{f^q} = 1$ ausführt.

Kommentare gelten bis zum Zeilenende und werden mit dem Symbol # eingeleitet. Für Details zur Konstruktion von Remopla-Modellen aus C- und Java-Programmen sei auf [ES01, Obd01, Obd02, RZ07] verwiesen.

```
L1: d=a, b=0, c=a;
L2: print(d), d=0, b=d;
L3: print(c), b=c, a=0;
L4: print(b), d=b, c=0;
L5: print(d);
```

```
L1: d=a, c=a;
L2: print(d), d=0, b=d;
L3: print(c), a=0;
L4: print(c), d=c, c=0;
L5: print(d);
```

Abbildung 2: Beispiel für lokale Optima. Links: original, Rechts: nach Elimination von b (b ist in L4 äquivalent zu c und kann duch c ersetzt werden) sind keine weiteren Eliminationen mehr möglich.

3 Variablenelimination für SPDS

Um Zustands- und Konfigurationenräume zu verkleinern, können Transformationen eingesetzt werden, welche in einem weiteren Schritt gewisse Variablen als „überflüssig“ identifizieren. Eine solche Variablenelimination wird im Folgenden genauer beschrieben und berücksichtigt auch Parameter und globale Variablen bei der Transformation, da somit noch mehr Variablen eliminiert werden können. Ein Kopf (p, a) bzw. dessen Variablenbelegung $f^q : Vars_q \rightarrow \mathbb{N}$ an der Marke q heißt **realisierbar**, falls ein Pfad[1] aus einer gegebenen Anfangskonfiguration s zu einer Konfiguration (p, aw) existiert ($s \xrightarrow{*} (p, aw)$). Zwei Variablen x und y eines SPDS vor einer Marke q heißen **äquivalent** (in Zeichen $x \equiv_q y$), wenn für alle realisierbaren Variablenbelegungen f^q gilt $f^q(x) = f^q(y)$. Analog heißt eine Variable x konstant ($x \equiv_q c$), falls $f^q(x) = c$ mit $c \in \mathbb{N}$. Äquivalente Variablen und Konstanten vor jeder Marke werden zu **Äquivalenzklassen** zusammengefasst. Diese ergeben für die Marke q die **Äquivalenzinformation** $E_q = \{\{x_{11}, x_{12}, ...x_{1n_1}\}, \{x_{21}, x_{22}, ... x_{2n_2}\}, ...\{x_{m1}, x_{m2}, ...x_{mn_m}\}\}$ mit $\forall i, j, l : x_{li} \equiv_q x_{lj}$. In Remopla-Modellen wird dafür abkürzend geschrieben x_{11}=x_{12}...=x_{1n_1} ... x_{m1}=x_{m2}...=x_{mn_m}.

Z.B. lassen sich die lesenden Verwendungen von b an der Marke L4 in Abbildung 2 (links) durch die Variable c ersetzen (rechts), da diese beiden Variablen wegen der Zuweisung b=c in L3 äquivalent sind. Die Variablenbelegungen des Modells ändern sich dabei nicht. Dies führt zu keiner weiteren Verwendung der Variablen b, die deshalb aus dem Modell eliminiert werden kann. In diesem Beispiel werden dabei die Anzahl $2 \cdot 10^{39}$ der notwenigen Köpfe, welche zu verarbeiten sind (6 symbolische Konfigurationen bei 5 symbolischen Transitionen und 4 Variablen mit 32 Bit), um den Faktor $4 \cdot 10^9$ reduziert. In rekursiven Aufrufen setzt sich diese Reduktion fort, wobei im Gegensatz zur Modellprüfung von endlichen Modellen die Rekursionstiefe und damit auch die Anzahl der Konfigurationen unbeschränkt sind. Weitere Ersetzungen sind dann allerdings nicht mehr möglich und die Variablen a, c und d verbleiben im Modell. Werden stattdessen d in L2 durch a, c in L3 durch b und d in L5 durch b ersetzt, so werden nur noch die Variablen a und b lesend verwendet, was zu zwei anstatt nur einer Variablenelimination führt. Dann werden die Anzahl der Köpfe von $2 \cdot 10^{39}$ um den Faktor $2 \cdot 10^{19}$ auf $1 \cdot 10^{20}$ reduziert. Dies zeigt, dass es nicht trivial ist zu entscheiden, welche Transformation zu einem besseren Ergebnis und insbesondere zu den meisten Variableneliminationen führen wird. Die Optimierung erfolgt

[1] nicht notwendigerweise endlich

U	x_1	x_2	x_3	x_4
m_1	1	1	0	1
m_2	0	1	0	1
m_3	1	0	1	1
m_4	0	0	1	1

```
  # {x1=m1=m3}       module use(int v, int p) {
L1: use(x1,1);         print(v);
  # {x2=m1=m2}         xi=undef, mi=undef; # i in [1..4]
L2: use(x2,2);         if
  # {x3=m3=m4}         :: p==1 -> m1=x2,m2=x2;
L3: use(x3,3);         :: p==2 -> m3=x3,m4=x3;
  # {x4=m1=m2=m3=m4}   :: p==3 -> m1=x4,m2=x4,m3=x4,m4=x4;
L4: use(x4,4);         fi; return; }
```

Abbildung 3: Reduktion vom Überdeckungsproblem auf optimale Repräsentantenwahl $\{m_1, m_3\}$

in folgenden Phasen[2]:

1. Berechne Äquivalenzinformationen mittels gegebener Äquivalenzanalyse.

2a. Ersetze Variablenverwendungen durch äquivalente Konstanten, falls möglich.

2b. Ersetze Variablenverwendungen durch äquivalente Variablen (Repräsentanten).

3. Eliminiere Variablendeklarationen von nichtlesend verwendeten Variablen.

Im Folgendem sei angenommen, dass die Äquivalenzinformationen aus der ersten Phase gegeben sind. Sie können z.B. mittels angepasster Datenflussanalysen approximiert oder exakt berechnet werden [Ric08]. Das Ziel der NP-harten Repräsentantenwahl (siehe Satz 1) in Phase 2b ist es, die Repäsentanten so zu bestimmen, dass in Phase 3 möglichst viele Variablen eliminiert werden. Gar nicht verwendete Variablen, die jedoch deklariert sind, können in Phase 3 direkt aus der symbolischen Beschreibung entfernt werden. Werden Variablen aber nur schreiben und nichtlesend verwendet, so sind diese Variablen automatisch tot und damit ebenso überflüssig, da ihr Wert nirgens benötigt wird.

Satz 1 *(Komplexität der Repräsentantenwahl)*
Die optimale Repräsentantenwahl[3] *bei gegebenen Äquivalenzinformationen ist NP-hart.*

Beweis (Skizze): Reduktion des NP-vollständigen Überdeckungsproblems auf die optimale Repräsentantenwahl. Sei eine beliebige Überdeckungsmatrix $U = (u_{ij}) \in \{0,1\}^{m,n}$ gegeben. Gesucht ist eine minimale Auswahl an Zeilen von U, so dass in jeder Spalte mindestens eine 1 steht. Man konstruiere ein SPDS wie in Abbildung 3. Dann wird durch die

[2]Falls nötig erfolgt eine Umbenennung der z.B. gleichenamigen lokalen bzw. globalen Variablen.
[3]d.h. eine minimale Auswahl an Repräsentanten bzw. Variablendeklarationen

optimale Repräsentantenwahl für $L1..Ln$ das Überdeckungsproblem gelöst. Die geltenden Äquivalenzen finden sich als Kommentar mit „#“ eingeleitet.

In Phase 2b wird eine Äquivalenzmatrix $A = (a_{ij})$ mit $a_{ij} \in \{0, 1\}$ erzeugt, um die Repräsentantenwahl optimal mit einem ILP-Solver zu lösen. Die Zeilen von A entsprechen den lokal und global deklarierten sowie den Parameter-Variablen. Die Spalten entsprechen den Variablenverwendungen an den entsprechenden Marken, d.h. ist $a_{ij} = 1$ gdw. die j-te Variablenverwendung gemäß gegebener Äquivalenzanalyse äquivalent ist zur Variable i. Wenn Variablen durch äquivalente ersetzt werden, dann werden entsprechende Zeilen in A gestrichen. Deshalb sollen möglichst viele Zeilen gestrichen werden, so dass jede Spaltensumme ≥ 1 bleibt. In Phase 2 ergeben sich auf diese Weise deklarierte Variablen, welche lediglich schreibend oder gar nicht verwendet werden. Diese Variablen sind redundant und können in Phase 3 aus dem Modell entfernt werden. Dies komprimiert die innere Struktur des Modells durch Verlagerung auf äquivalente Bereiche. Wenn dann wie im Beispiel aus Abbildung 2 die Variablendeklaration einer Variablen auf der linken Seite einer Zuweisung entfernt wurde, dann kann (wie bei unserem **HalSPSI**) eine einfache Seiteneffekteanalyse durchgeführt werden, um zu bestimmen, ob die Zuweisung komplett entfallen darf, da die Ausdrucksauswertung u.U. Seiteneffekte wie Division durch 0, Modulaufruf oder einen Überlauf enthalten kann.

Satz 2 *(Korrektheit der Transformation)*
Die Transformation aus Phase 2 ändert das Verhalten des SPDS nicht. Das in Phase 3 reduzierte und das ursprüngliche SPDS sind bisimular.

Beweis (Skizze): Phase 2: Anwenden von Phase 2 lässt das durch das SPDS definierte Kellersystem unverändert, da lediglich äquivalente Konstanten/Variablen ersetzt werden.
Phase 3: Der Inhalt von nichtlesend verwendeten bzw. toten Variablen hat natürlich keinen Effekt auf das Modellverhalten. Konfigurationenübergänge werden daher nicht beeinflusst durch die Abwesenheit dieser Variablen.

4 Experimente

Als Grundlage für die Experimente dienten 191 Remopla-Modelle, zu denen exakte Äquivalenzinformationen berechnet werden konnten. Die Modelle wurden mittels JMoped aus Java Beispielen gewonnen, welche zum größten Teil zu den Benchmark-Instanzen der Werkzeuge JMoped bzw. Moped gehören. Zur Modellgenerierung wurden jeweils unterschiedliche Bitbreiten (bis zu 8 Bit) zur Modellierung von Ganzzahlen (Integer) verwendet. Aufgabe für den Modellprüfer Moped war es, zu diesen Modellen jeweils vor und nach der Variablenelimination das Fehlschlagen von Java-Zusicherungen (Assertions) zu prüfen. In gleicher Weise können auch nicht (korrekt) behandelte Ausnahmen, Speicherüberläufe oder andere Fehler automatisch geprüft werden. Durchgeführt wurden die Experimente mit einem AMD 64 X2 4200+ mit 2 GB Hauptspeicher unter Linux (Kernel 2.6.27). Für die Untersuchungen wurden neben einer exakten Äquivalenzanalyse [Ric09] auch eine approximative (konservative) interprozedurale kontextinsensitive Äquivalenz-

Tabelle 1: Modellprüfzeiten für Moped (inklusive Modellanalyse und Transformation).

Beispiel(Auswahl)	**4Bit-**	**4Bit+**	**5Bit-**	**5Bit+**	**6Bit-**	**6Bit+**	**7Bit-**	**7Bit+**	**8Bit-**	**8Bit+**
ArrayFib	6 s	3 s	67 s	13 s	329 s	120 s	1446 s	522 s	MOut	MOut
ArrayUtils	8 s	1 s	109 s	2 s	453 s	12 s	1729 s	125 s	MOut	**1826 s**
ConcreteF..Class	$<$ 1 s	$<$ 1 s	6 s	$<$ 1 s	39 s	$<$ 1 s	111 s	1 s	329 s	1 s
Dispatching	329 s	12 s	MOut	**758 s**	MOut	MOut	MOut	MOut	MOut	MOut
Ex..InOneLine	5 s	$<$ 1 s	58 s	1 s	264 s	6 s	962 s	15 s	MOut	**36 s**
Fibonacci	1 s	2 s	5 s	5 s	37 s	14 s	115 s	51 s	328 s	169 s
IntBufferTest	368 s	21 s	MOut	**919 s**	MOut	MOut	MOut	MOut	MOut	MOut
Isq	$<$ 1 s	1 s	1 s	1 s	4 s	4 s	30 s	11 s	355 s	44 s
LinkedList	2 s	$<$ 1 s	12 s	1 s	59 s	5 s	234 s	12 s	702 s	34 s
MemoFib	4 s	$<$ 1 s	MOut	**9 s**	MOut	**113 s**	MOut	**502 s**	MOut	MOut
Par..Restrictions	5 s	1 s	68 s	9 s	340 s	110 s	1538 s	500 s	MOut	MOut
RecFib	7 s	$<$ 1 s	7 s	$<$ 1 s	39 s	1 s	111 s	1 s	317 s	5 s
ShortEval	343 s	20 s	MOut	**839 s**	MOut	MOut	MOut	MOut	MOut	MOut
While	$<$ 1 s	$<$ 1 s	1 s	$<$ 1 s	1 s	1 s	2 s	1 s	3 s	2 s
false_neg_bits	339 s	15 s	MOut	**741 s**	MOut	MOut	MOut	MOut	MOut	MOut
Gesamt	1417 s	76 s	334 s	32 s	1565 s	273 s	6278 s	1239 s	2034 s	255 s

Spalten „-“ ohne und Spalten „+“ mit Variablenelimination. Ist in einem der Fälle ein Speicherüberlauf (Einträge MOut) aufgetreten, so wurde die Laufzeit der anderen Version (Fettschrift) nicht in die Gesamtzeit aufgenommen.

Tabelle 2: Gleichheit der approximativen und exakten Äquivalenzklassen

Code-Beispiel	
ParameterRestrictions.java (7Bits)	96%
ConcreteFieldClass.java (8Bit)	92%
While.java (8Bit)	85%
Durchschnitt (alle 191 Instanzen)	76%

analyse verwendet, um die notwendigen Äquivalenzinformationen zu bestimmen. Die Berechnung der exakten Äquivalenzanalyse benötigte oft mehr Zeit als die Modellprüfung selbst, während das in Tabelle 1 benutzte approximative Verfahren vernachlässigbar kleine Laufzeiten hat. Die Repräsentanten wurden in den Untersuchungen aus Tabelle 1 optimal gewählt, können in **HalSPSI** aber auch heuristisch bestimmt werden. Dann werden diejenigen Repräsentanten bevorzugt, welche besonders häufig in Äquivalenzklassen auftreten, da diese intuitiv eine besonders hohe Wahrscheinlichkeit besitzen, andere Variablen zu überdecken. Beide Äquivalenzanalysen und die Variablenelimination wurden in erweiterten Formen in unser **HalSPSI** für SPDS in Remopla-Syntax implementiert. Durch die konservative Äquivalenzanalyse werden in den Beispielen im Durchschnitt 76% und in Extremfällen über 96% der auftretenden Äquivalenzen unter den Variablen erkannt (siehe Tabelle 2). Wird demnach zu der Benchmark-Instanz ConcreteFieldClass.java ein Remoplamodell mit 8 Bit Integern erzeugt, so sind die Äquivalenzklassen, welche die konservative Äquivalenzanalyse liefert um ca. 8% kleiner als die tatsächlichen (exakten) Äquivalenzklassen. In Tabelle 1 wurde die Modellprüfzeit (einschließlich Modell-Generierung und -Optimierung) für den Modellprüfer Moped für die optimierten und unoptimierten Modelle miteinander verglichen. Es wurden verschiedene Integer-Bitbreiten untersucht. Wobei $^-$ für das unveränderte Orinigalverfahren und $^+$ für die Variablenelimination mit konservativer Äquivalenzanalyse steht. Dabei beträgt die Zeit zur Analyse und Transformation der Modelle mittels der konservativen Äquivalenzanalyse durchschnittlich weniger

als eine Sekunde. Es zeigt sich, dass die Transformationen die Modellgröße signifikant um viele Größenordnungen verringern und sich dies auf eine wesentlich geringere Modellprüfzeit auswirkt (einschließlich der Zeit für die Analyse und Transformation der Modelle). In seltenen Fällen (wie bei Fibonacci bei 4Bit), kann es vorkommen, dass die Zeit für die Analyse und Transformation der Modelle die eigentliche Zeit für die Modellprüfung übersteigt. Dies gilt aber nur für sehr kleine Laufzeiten des Modellprüfers. Die Effekte der Transformation und insbesondere deren reduzierende Auswirkung auf die Modellgröße führen in den Fällen von großen Konfigurationenräumen zu um so mehr Reduktionen und Verbesserungen der Modellprüfzeiten. Insbesondere ermöglichte die Variablenelimination überhaupt erst in 9 Fällen die Modellprüfung, da unoptimiert Speicherüberläufe auftreten.

5 Verwandte Arbeiten

Einige Modellprüfer für endliche Systeme (darunter SPIN) nutzen Slicing-Techniken, um ihre Modelle zu verkleinern [MT98]. Dabei können auch Variablen überflüssig werden. Jedoch ist die maximal mögliche Elimination von Variablen dort nicht das Ziel. Zur besseren Modellgenerierung aus Quellcode können ebenfalls Programmanalysen eingesetzt werden [YWG09, ZYea08, GGea08]. Unser Ansatz hingegen ist unabhängig von der Quellsprache und erlaubt auch die Optimierung bestehender oder von Hand erstellter Modelle und kann zusätzlich zur verbesserten Modellgenerierung genutzt werden. Möchte man Programmanalysen [Muc97] in Remopla verwenden, so müssen diese um synchron parallele Konfigurationsübergänge ergänzt und können im Gegensatz zu Hochsprachen exakt durchgeführt werden [Ric08]. Dies führt zu besseren Transformationen (mehr Variableneliminationen) und zu präziseren Analysen für die Ausgangssprache. Zustandsreduktionstechniken mittels Lebendigkeitsanalysen für asynchrone endliche Systeme finden sich in [FBG03, YG04]. Unsere Techniken sind im Gegensatz dazu nicht auf endliche Modelle beschränkt und berücksichtigen exakte Rekursion. Eine weitere ähnliche Technik zu unserer Variablenelimination ist Macro Expansion [YSBO99], wo im Modellprüfprozesses direkt BDD Variablen durch äquivalente Ausdrücke ersetzt werden. Dazu ist es jedoch zunächst nötig, erst einmal das Modell (Kellersystem) zu erzeugen. Unser Source-To-Source Compiler **HalSPSI** führt die Transformationen direkt in der symbolischen Modellbeschreibung durch. Ein reduziertes Modell steht dann auch für Folgeschritte wie Testfallgenerierung zur Verfügung, ohne dabei das zugehörige Kellersystem konstruieren zu müssen. Macro Expansion kann vielmehr zusätzlich nach unserer Transformation während der Modellprüfung eingesetzt werden.

6 Zusammenfassung

Es wurde unsere Methode Variablenelimination für unendliche Modelle mit exakt behandelbarer Rekursion vorgestellt, welche die innere Zustandsstruktur komprimiert und Modelle bereits symbolisch vereinfacht. Dies ermöglicht u.A. effizienteres Testen, effi-

zientere Testdatengenerierung mit kompakteren Testdaten, einfachere Simulationen usw.. Unsere Experimente zeigen, dass durch unser System **HalSPSI** die Modellprüfung erheblich beschleunigt bzw. die Modellprüfung erst ermöglicht wird. Wie in den Experimenten zu erkennen war, finden bereits nichtexakte Äquivalenzanalysen viele Äquivalenzen und führen zu deutlichen Verbesserungen der Modelle. Das uns derzeit bekannte Verfahren zur Bestimmung exakter Äquivalenzinformationen ist allerdings ebenso aufwändig wie die Modellprüfung selbst[4]. Dennoch können nicht nur approximative, sondern auch exakte Methoden genutzt werden um präziser das Verhalten und Eigenschaften von SPDS und damit von C oder Java-Programmen vorherzusagen. Insbesondere ermöglicht die bessere Modellprüfung von SPDS, welche eine Form des vollständigen Testens darstellt, somit ein umfangreicheres Testen für die zu Grunde liegenden C oder Java-Programme. Hierzu bietet unser **HalSPSI** neben der hier präsentierten Variablenelimination noch weitere Reduktionsverfahren wie Wertebereichsreduktion, Stotterreduktion oder Slicing an. Im Zusammenspiel mit diesen Transformationen kann die Modellprüfung in Einzelfällen sogar ganz entfallen. Es ergeben sich dann nochmals beträchtliche Verbesserungen durch Synergieeffekte.

Literatur

[BEM97] A. Bouajjani, J. Esparza und O. Maler. *Reachability Analysis of Pushdown Automata: Application to Model-Checking*. Proc. of the 8th International Conference on Concurrency Theory, LNCS 1243, 1997.

[Ber06] F. Berger. *A test and verification environment for Java programs*. Diplomarbeit Nr. 2470, Universität Stuttgart, 2006.

[EHRS00] J. Esparza, D. Hansel, P. Rossmanith und S. Schwoon. *Efficient algorithms for model checking pushdown systems*. Proc. of the 12th International Conference on Computer Aided Verification, LNCS 1855, 2000.

[EKS02] J. Esparza, A. Kucera und S. Schwoon. *Model-Checking LTL with Regular Valuations for Pushdown Systems*. Proc. of the 4th International Symposium on Theoretical Aspects of Computer Software, LNCS 2215, 2002.

[ES01] J. Esparza und S. Schwoon. *A BDD-based model checker for recursive programs*. LNCS Volume 2102, 324-336, Springer, 2001.

[FBG03] J.C. Fernandez, M. Bozga und L. Ghirvu. *State space reduction based on live variables analysis*. Science of Computer Programming, Vol 47, Issue 2, 203-220, 2003.

[GGea08] K. M. Ganai, A. Gupta und F. Ivancic et al. *Towards Precise and Scalable Verification of Embedded Software*. Proc. of Design and Verification Conference (DVCon), 2008.

[KSS06] S. Kiefer, S. Schwoon und D. Suwimonteerabuth. *Introduction to Remopla*. Institute of Formal Methods in Computer Science, University of Stuttgart, 2006.

[LH99] D. Liang und M. J. Harrold. *Equivalence analysis: a general technique to improve the efficiency of data-flow analyses in the presence of pointers*. ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, 1999.

[4] Es ist sogar komplexitättheoretisch optimal.

[LH03] D. Liang und M. J. Harrold. *Equivalence analysis and its application in improving the efficiency of program slicing*. Transactions on Software Engineering and Methodology (TOSEM), Volume 11 Issue 3, 2003.

[MT98] L. Millett und T. Teitelbaum. *Slicing Promela and its applications to model checking, simulation, and protocol understanding*. Proc. 4th International SPIN Workshop, 1998.

[Muc97] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[Obd01] J. Obdrzalek. *Formal verification of sequential systems with infinitely many states*. Master's Thesis, FI MU Brno, Masaryk University, 2001.

[Obd02] J. Obdrzalek. *Model Checking Java Using Pushdown Systems*. LFCS, University of Edinburgh, 2002.

[Ric08] D. Richter. *Modellreduktionstechniken für symbolische Kellersysteme*. Proc. of the 25. Workshop 'Programmiersprachen und Rechenkonzepte', University Kiel, 2008.

[Ric09] D. Richter. *Äquivalenzanalysen - exakt oder nicht - im Vergleich*. Erscheint im Rahmen des 26. Workshops 'Programmiersprachen und Rechenkonzepte', University Kiel, 2009.

[RZ07] D. Richter und W. Zimmermann. *Slicing zur Modellreduktion von symbolischen Kellersystemen*. Proc. of the 24. Workshop of GI-section 'Programmiersprachen und Rechenkonzepte', University Kiel, 2007.

[Sch02] S. Schwoon. *Model-Checking Pushdown Systems*. Technische Universität München, 2002.

[SSE05] D. Suwimonteerabuth, S. Schwoon und J. Esparza. *jMoped: A Java Bytecode Checker Based on Moped*. Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, Springer, 2005.

[VHB+03] W. Visser, K. Havelund, G. Brat, S. Park und F. Lerda. *Model Checking Programs*. Automated Software Engineering Volume 10(2), 203-232, Kluwer Academic, 2003.

[Wal00] Igor Walukiewicz. *Model checking CTL Properties of Pushdown Systems*. In FSTTCS'00, LNCS 1974, 2000.

[YG04] K. Yorav und O. Grumberg. *Static analysis for state-space reductions preserving temporal logics*. Formal Methods in System Design, Vol 25(1), 67-96, Springer, 2004.

[YSBO99] B. Yang, R. Simmons, R.E. Bryant und D.R. O'Hallaron. *Optimizing symbolic model checking for constraint-rich models*. Proc. of CAV 11th International Conference, 328-340 , Springer, 1999.

[YWG09] Z. Yang, C. Wang und A. Gupta. *Model checking sequential software programs via mixed symbolic analysis*. ACM Transactions on Design Automation of Electronic Systems (TODAES), Vol 14, Issue 1, 2009.

[ZYea08] A. Zaks, Z. Yang und I. Shlyakhter et al. *Bitwidth Reduction via Symbolic Interval Analysis for Software Model Checking*. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol 28, Issue 8, 1513-1517, 2008.