

Äquivalenzanalysen - exakt oder nicht - im Vergleich

Dirk Richter

Martin-Luther-Universität Halle-Wittenberg, Institut für Informatik,
06099 Halle/Saale, Germany, richterd@informatik.uni-halle.de

Zusammenfassung Symbolische Modelle definieren Zustandssysteme, welche zur Software-Modell-Prüfung, zum Modell basierten Testen und zur Codegenerierung genutzt werden können. Die Größe und Komplexität dieser Modelle sind dabei entscheidende Einflussfaktoren und sollten möglichst vereinfacht werden. Aus Quellcode 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 wurden zwei verschiedene Ansätze verfolgt, die die innere Struktur der Zustände ausnutzen, um den Zustandsraum der Modelle weiter zu verkleinern. Eigene Experimente zeigen, dass diese die Modellprüfung erheblich beschleunigen bzw. die Modellprüfung erst ermöglichen.

Key words: Kellersystem, Remopla, Moped, Software-Modell-Prüfung, Variablenelimination, Äquivalenzanalyse

1 Einleitung

Im Gegensatz zu vergleichbaren Arbeiten bei 'Finite-State' Modellprüfern wie BLAST, SPIN, NuSMV/SMV, F-Soft oder Bogor (Bandera Projekt) untersuche ich unbeschränkte unendliche Modelle in Form von sog. symbolischen Kellersystemen (SPDS). Viele Modellprüfer beschränken die Rekursionstiefe oder verbieten Methodenaufrufe. Durch diese Unter- bzw. Überapproximation von Methodenaufrufen entstehen Fehlalarme (False Negatives sowie Fehlabstraktionen), die durch korrekte Abbildung von Methodenaufrufen und Rekursion auf SPDS vermieden werden können. Die Beschränkung auf eine konstante maximale Anzahl an Methodenaufrufen ist dann nicht nötig und vermeidet eine exponentielle Modellvergrößerung z.B. durch Inlining. Solche SPDS können mittels JMoped aus Java Bytecode gewonnen und mittels des Modellprüfers Moped überprüft werden und ermöglichen mächtigere interprozedurale und kontextsensitive Modell-Analysen, -Tests und -Prüfungen. Bei Verwendung des Cross-Compilers Grasshopper kann nicht nur Java 1.6 Bytecode 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 noch unhandlich ist.

Listing 1.1. Java Beispiel zur Berechnung der Fakultät

```
int fac(int n) {
    if (n<=1) return 1;
    return n*fac(n-1);
}
```

Um SPDS Modelle zu optimieren, sind Informationen über das Modellverhalten wichtig, die durch im Programmiersprachenumfeld gängige Programm-Analysen gewonnen werden können. Bei einer sog. Äquivalenzanalyse werden z.B. durch Verwendung von Konstanten-Propagation und -Faltung sowie Copy-Propagation Gleichheit und Konstanz von Variablen erkannt. Liang und Harrold zeigen in [1, 2], dass bessere Äquivalenzanalysen zu besserem Slicing sowie besseren Zeiger- und Datenflussanalysen führen. In [3] und [4] 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 Konstanten-Propagation und -Faltung sowie Copy-Propagation zu einer konservativen Äquivalenzanalyse. Bei einer solchen konservativen Analyse kann es Variablen geben, die zwar gleich oder gar konstant sind, dies aber nicht durch die Analyse entdeckt wird.

Hier soll nun eine modifizierte Version der konservativen Äquivalenzanalyse für SPDS aus [4] kurz erläutert und mit einem neuen **exakten** Verfahren verglichen werden.

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 ($s \xrightarrow{*} z$?). Statt des Erreichbarkeitsproblems können im Rahmen von Modellprüfung auch LTL- oder CTL*-Formeln überprüft werden [5]. Dabei ist eine LTL- oder CTL*-Formel [6] für diese Kripkestruktur oder deren symbolische Beschreibung gegeben. Gefragt wird dann nach der Gültigkeit dieser Formel für einen Anfangszustand bzw. einer Menge von Anfangszuständen dieser Kripkestruktur. 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 sind. Informell ist ein Kellersystem ein Kellerautomat ohne Eingabe. (p, v) heißt **Konfiguration**, 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 Konfigu-

Listing 1.2. Generiertes Remopla-Modell (Auszug) zu Listing 1.1

```
1 module int fac(int v0(16)) {
2   int s0(16); int s1(16); int s2(16);
3   fac_I: s0=v0, s1=s0, s2=s1;
4   fac_I1: s0=1, s1=s0, s2=s1;
5   fac_I2: if
6     :: s1>s0 -> goto fac_I7, s0=s2;
7     :: else -> s0=s2;
8     fi;
9   fac_I5: s0=1, s1=s0, s2=s1;
10  fac_I6: return s0;
11  fac_I7: s0=v0, s1=s0, s2=s1;
12  fac_I8: s0=v0, s1=s0, s2=s1;
13  fac_I9: s0=1, s1=s0, s2=s1;
14  fac_I10: if
15    :: s1>=s0 -> s0=s1-s0, s1=s2;
16    :: else -> s0=(65536-(s0-s1)%65536)%65536, s1=s2;
17    fi;
18  fac_I11: s0=fac_I(s0);
19  fac_I14: s0=(s1*s0)%65536, s1=s2;
20  fac_I15: return s0; }
```

rationen wird die Transitionsrelation \leftrightarrow erweitert zu $\rightarrow \subseteq (P \times \Gamma^*) \times (P \times \Gamma^*)$ mit $(p, aw) \rightarrow (q, bw) :\Leftrightarrow (p, a) \leftrightarrow (q, b)$. Mit $\overset{*}{\rightarrow}$ 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 vollständigen Kellersystems vereinfacht [7].

Solche SPDS können mit Hilfe der Modellsprache Remopla [8] modelliert werden. Remopla ist zwar syntaktisch ähnlich zu Promela (Eingabesprache für den SPIN Modellprüfer [9]), unterstützt aber keine parallelen Prozesse, sondern synchron parallele Konfigurationenübergänge und explizite Rekursion. In Listing 1.1 ist ein Java-Beispiel zu Berechnung der Fakultätsfunktion abgebildet. Dieses kann mittels des Werkzeuges JMoped automatisch in das Remopla-Modell aus Listing 1.2 überführt werden. Dabei wurden 16 Bit (angehängt an eine Variablendeklaration) zur Modellierung von Integern verwendet, um das Verhalten des Java Programms nachzubilden. Wie dabei nicht zu sehen ist, können in Remopla neben lokalen Variablen loc_q und Parametern $pars_q \subseteq loc_q$ eines Moduls q (auch Prozedur genannt) auch globale Variablen $globs$ sowie Arrays oder Verbunde (Struct) deklariert werden. Die lokalen Variablen und Methodenparameter werden in SPDS als Bitvektoren über dem Kellularphabet 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. Für weitere Details zur Konstruktion von Remopla-Modellen für C und Java sei auf [10–12, 3] verwiesen.

So konstruierte Modelle in Form eines Remopla-Modells können dann durch mein Werkzeug **HalSPSI** verbessert und anschließend durch den Modellprüfer Moped [10] geprüft oder anders weitergenutzt werden. Ziel der im Folgenden erklärten Äquivalenzanalysen ist es, das dazu nötige Kellularalphabet und die benötigten Kellerzustände bereits symbolisch a priori so stark wie möglich zu verringern. Desto genauer hierzu das Analyseergebnis ist, desto bessere Transformationen können später erfolgen. Exakte Äquivalenzanalysen können daher zu einer angestrebten optimalen Transformation führen.

3 Variablenelimination mittels Äquivalenzanalysen

Mittels so genannter Äquivalenzanalysen können Transformationen eingesetzt werden, welche den Zustands- und Konfigurationenraum verkleinern, indem gewisse Variablen als „überflüssig“ erkannt werden. Eine solche Transformation ist die Variablenelimination, welche Äquivalenzanalysen nutzt und hier kurz als Motivation erläutert wird.

Sei dazu angenommen, dass $Vars_q$ die Menge der an der SPDS Marke q verfügbaren Variablen (globale und lokale sowie Parameter-Variablen) ist. Ein Kopf (p, a) bzw. dessen Variablenbelegung $f^q : Vars_q \rightarrow \mathbb{N}$ an der Marke q heißt **realisierbar**, falls ein Pfad¹ aus einer gegebenen Anfangskonfiguration s zu einer Konfiguration (p, aw) existiert (in Zeichen $s \xrightarrow{*} (p, aw)$). 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 $c \in \mathbb{N}$. D.h. für alle k , welche $s \xrightarrow{*} k$ mit $q = Label(k)$ erfüllen, gilt: $f^q(x) = f^q(y)$ bzw. $f^q(x) = c$. Äquivalente Variablen und Konstanten vor jeder Marke werden zu **Äquivalenzklassen** zusammengefasst. Diese ergeben für die Marke q die **Äquivalenzinformation** $E_q = \{\{x_{11}, x_{12}, \dots, x_{1n_1}\}, \{x_{21}, x_{22}, \dots, x_{2n_2}\}, \dots, \{x_{m1}, x_{m2}, \dots, x_{mn_m}\}\}$ mit $\forall i, j, l : x_{li} \equiv_q x_{lj}$. In Remopla-Modellen wird dafür abkürzend geschrieben $x_{11} = x_{12} \dots = x_{1n_1} \dots x_{m1} = x_{m2} \dots = x_{mn_m}$.

Wird z.B. in Listing 1.2 zum Moduleintritt von fac keine Äquivalenz angenommen, gilt nach den parallelen Zuweisungen in Zeile 3 $v0 \equiv_{fac_I1} s0$ und nach den Zuweisungen aus Zeile 4 $s0 \equiv_{fac_I2} 1$ sowie $s1 \equiv_{fac_I2} v0$. Mit Hilfe dieser Äquivalenzinformationen kann z.B. der boolesche Ausdruck $s1 > s0$ in Zeile 6 durch den äquivalenten Ausdruck $v0 > 1$ ersetzt werden und erlaubt auf diese Weise eine mögliche Einsparung der beiden lokalen Variablen $s0$ und $s1$, was den Zustands- und Konfigurationenraum verkleinert.

In [4] wurde gezeigt, dass das Finden einer Ersetzung von Variablen durch äquivalente Variablen/Konstanten mit minimalem SPDS-Konfigurationenraum NP-schwer ist. In der Praxis ist dies allerdings dank effizienter ILP-Solver (Integer Linear Program) unkritisch. Daher wurde in den Experimenten das Verfahren aus [4] um eine optimale Repräsentantenwahl (Auswahl von Variablendeklarationen) erweitert, welche durch Konstruktion eines ILP ein Überdeckungsproblem

¹ nicht notwendigerweise endlich

löst. Dabei muss jede Variablenverwendung durch eine Variablendeklaration einer äquivalenten Variable oder Konstante „überdeckt“ werden. Dadurch ändert sich das durch das SPDS beschriebene Kellersystem nicht. Auch Aussagen von LTL/CTL*-Formeln bleiben unverändert, solange lediglich lesende Verwendungen von Variablen durch andere ersetzt werden. Wird allerdings eine Variable überflüssig und somit in einer späteren Phase aus dem Modell entfernt, so geht dies i.A. nur, sofern die gegebene LTL/CTL*-Formel nicht über diese ‚wegoptimierte‘ Variable spricht. Falls doch, so sind in jeder symbolischen Konfiguration des SPDS ggf. indirekte Abhängigkeiten einzuführen, um die durch die LTL/CTL*-Formel erzeugten indirekten Verwendungen (über Prädikate) ebenfalls abzudecken und damit die an der LTL/CTL*-Formel beteiligten Variablen in die Lösung der Repräsentantenwahl zu zwingen².

4 Konservative Äquivalenzanalyse

Für gängige Hochsprachen ist es **unentscheidbar** (Reduktion Postsches Korrespondenzproblem) fest zu stellen, ob zwei Variablen x und y an einem Programmpunkt q äquivalent sind [13]. Wie in [4] gezeigt, ist dieses Problem für SPDS **entscheidbar**, wenngleich zu aufwändig um damit Modellprüfungen zu beschleunigen. Daher wurde zunächst eine konservative Äquivalenzanalyse untersucht. Eine Äquivalenzinformation $E_q = \{\{x_{11}, x_{12}, \dots, x_{1n_1}\}, \{x_{21}, x_{22}, \dots, x_{2n_2}\}, \dots, \{x_{m1}, x_{m2}, \dots, x_{mn_m}\}\}$ an einer Marke q heißt **konservativ**, falls $\forall i, j, l : x_{li} \equiv_q x_{lj}$. Anders als bei exakten Äquivalenzinformationen kann es hier Variablenäquivalenzen $a \equiv_q b$ geben, so dass $\exists g \in E_q : a, b \in g$.

Wird zu der in [4] beschriebenen Konstanten-Propagation und -Faltung noch eine „Copy-Propagation-Analysis“ verwendet, so können damit „Copy-Chains“³ verfolgt werden. Ein Ziel ist das Aufbrechen dieser „Copy-Chains“, damit letztendlich weniger Variablen im Modell benötigt werden und sich der Zustands- und Konfigurationenraum des Modells verringert. Konkret werden bei der implementierten interprozeduralen Äquivalenzanalyse Äquivalenzen an Konfigurationen für Variablen und zu Konstanten auswertbaren Ausdrücken verfolgt. Z.B. wird nicht durch jede Äquivalenzanalyse die Variablenäquivalenz $x \equiv_q y$ in Zeile 9 von Listing 1.3 erkannt. Es wird in einem Lauf des Modells aus Listing 1.3 eine Anweisungen aus den Zeilen 5 bis 7 zufällig ausgewählt, da alle If-Bedingungen erfüllt sind. Auch meine konservative Äquivalenzanalyse erkennt die frühzeitige Terminierung aller Modellpfade in Zeile 7 durch einen arithmetischen Überlauf nicht und kann damit die Äquivalenz $x \equiv_q y$ in Zeile 9 nicht erkennen.

5 Exakte Äquivalenzanalyse

In [7] wird beschrieben, wie zu einem gegebenen SPDS P symbolisch ein endlicher Automat $Post^*(P)$ konstruiert werden kann, welcher die Menge aller er-

² gleichgültig, ob optimale Repräsentantenwahl oder nicht

³ Copy-Chains sind Zuweisungen mit Wertweiterreichung, wie sie typischerweise bei einem simulierten Operadenstack bei Push- und Pop-Operationen auftreten.

Listing 1.3. Variablenäquivalenz von x und y an q trotz partieller Symmetrie

```
1   int x(1);
2   int y(1);
3   x=1, y=0;
4   if
5       :: true -> x=y, y=x;    # gleichbedeutend zu x=0, y=1;
6       :: true -> skip;
7       :: true -> y=x+1;      # Pfad terminiert wegen Überlauf
8   fi;
9   # {x=y} <= unerkenbare Äquivalenz für konservative Analyse
10  q: goto q;
```

reichbaren Konfigurationen der gegebenen Initialkonfigurationen charakterisiert. Mittels $Post^*(P)$ kann aus den Initialkonfigurationen zu jeder SPDS Marke q eine charakteristische Funktion f^q für die exakten Variablenbelegungen für lokale und globale Variablen bestimmt werden mittels Beschränkung der Transitionen aus $Post^*(P)$ auf symbolische Köpfe. Es wird nun gezeigt, wie aus dieser charakteristischen Funktion f^q die exakten Äquivalenzinformationen effizient bestimmt werden und dabei sog. Symmetrien ausgenutzt werden können. Nach [14] heißt dazu eine Funktion $f : \{0, 1\}^n \rightarrow \{0, 1\}$ partiell **symmetrisch in den Variablen** x_k **und** x_j , falls für jede Belegung der x_i gilt: $f(x_1, \dots, x_{k-1}, x_k, x_{k+1}, \dots, x_{j-1}, x_j, x_{j+1}, \dots, x_n) = f(x_1, \dots, x_{k-1}, x_j, x_{k+1}, \dots, x_{j-1}, x_k, x_{j+1}, \dots, x_n)$. Die Funktion f heißt **symmetrisch in einer Menge** von disjunkten Variablenmengen⁴ R , wenn f partiell symmetrisch in je zwei Variablen $a, b \in m$ jeder Menge $m \in R$ ist. Die Funktion $f_{x_i} := f(x_1, \dots, x_{i-1}, 1, x_{i+1}, \dots, x_n)$ heißt **positiver Kofaktor** und $f_{x'_i} := f(x_1, \dots, x_{i-1}, 0, x_{i+1}, \dots, x_n)$ **negativer Kofaktor** von f . Man bezeichnet $f_{ab} := (f_a)_b$ als iterierten Kofaktor. Weitere Details finden sich in [14].

Symmetrien und die zuvor beschriebenen Äquivalenzklassen weisen einen interessanten Zusammenhang auf: jede Äquivalenz zweier Variablen in einem SPDS führt zu partiellen Symmetrien. Wie folgendes Theorem zeigt, gilt dies allerdings nicht für die Umkehrung, weswegen Symmetrieberechnungen nicht ausschließlich zur Bestimmung der Äquivalenzklassen genutzt werden können.

Theorem 1. Partielle Symmetrie der Variablenäquivalenz

Sind zwei Variablen $x = (x_1, x_2, \dots, x_n)$ und $y = (y_1, y_2, \dots, y_n)$ eines SPDS an einer Marke q äquivalent (also $x \equiv_q y$), so ist die charakteristische Funktion $f^q : \{0, 1\}^{2n} \rightarrow \{0, 1\}$ der Variablenbelegungen⁵ für q partiell symmetrisch in der Menge von Variablenmengen $\{\{x_1, y_1\}, \{x_2, y_2\}, \dots, \{x_n, y_n\}\}$. D.h. für jedes i ist f^q in den Variablen x_i und y_i partiell symmetrisch. Die Umkehrung des Theorems gilt nicht.

⁴ $R \subseteq 2^{\{x_1, x_2, \dots, x_n\}}$ bzw. $\cup R \subseteq \{x_1, x_2, \dots, x_n\}$ wobei $\forall a, b \in R : a \cap b = \emptyset$

⁵ z.B. als BDD repräsentiert

Proof. „ \Rightarrow “ Sei $x \equiv_q y$, so gilt für jede Belegung der Variablen x und y : $x_1 = y_1, x_2 = y_2, \dots, x_n = y_n$ und damit $\forall i : f^q$ ist partiell symmetrisch in den Variablen x_i und $y_i \Rightarrow f^q$ ist partiell symmetrisch in der Partition $\{\{x_1, y_1\}, \{x_2, y_2\}, \dots, \{x_n, y_n\}\}$.

„ \Leftarrow “ Gegeben sei das SPDS aus Listing 1.3. Darin sind alle Variablenbelegungen an der Marke q als $\{\{x = 0, y = 1\}, \{x = 1, y = 0\}\}$ erkennbar. x und y sind hiernach nicht äquivalent. Wegen $f^q(x, y) = f(0, 1) = f(1, 0) = 1$ ist f^q aber trotzdem partiell symmetrisch in den Variablen x und y . ■

Für einen Äquivalenztest zweier Variablen genügt es, Kofaktoren zu bilden:

Theorem 2. Äquivalenzbestimmung zweier SPDS-Variablen x und y

Sei f^q die charakteristische Funktion aller Köpfe von Konfigurationen an der Marke q . Dann gilt für $x, y \in \text{Vars}_q$: $x \equiv_q y \Leftrightarrow \forall i : f_{x_i y'_i}^q = \emptyset \wedge f_{x'_i y_i}^q = \emptyset$

Proof. O.B.d.A. sei $f := f^q$ mit $f \neq \emptyset$.

„ \Rightarrow “ Sei $x \equiv_q y$. Dann gilt für jede mögliche Belegung der Variablen x bzw. y durch $a = (a_1 a_2 \dots a_n) \in \{0, 1\}^n$ bzw. $b = (b_1 b_2 \dots b_n) \in \{0, 1\}^n$: $(f[x := a, y := b] \neq \emptyset) \Rightarrow a = b$. Gäbe es nun ein i , so dass $f_{x_i y'_i} \neq \emptyset$, so gäbe es Belegungen α, β mit $\alpha_i = 1$ und $\beta_i = 0$, so dass $f[x := \alpha, y := \beta] \neq \emptyset$. Die Voraussetzung liefert dann aber wegen $\alpha = \beta$ einen Widerspruch zu $\alpha_i = 1 \neq 0 = \beta_i$. D.h. es kann kein solches i geben, also ist $\forall i : f_{x_i y'_i} = \emptyset$. Analoges gilt für die Annahme $f_{x'_i y_i} \neq \emptyset$.

„ \Leftarrow “ Sei $\forall i : f_{x_i y'_i} = \emptyset \wedge f_{x'_i y_i} = \emptyset$ und angenommen $x \not\equiv_q y$. Nach Annahme $\exists j, \exists a, b \in \{0, 1\}^n : a_j \neq b_j$ und $f[x := a, y := b] \neq \emptyset$.

1. Fall: $a_j = 1$. Wegen $a_j \neq b_j$ ist dann $b_j = 0$ und damit $\emptyset \neq f[x := a, y := b] = f[x := a, y := b]_{x_j} = f[x := a, y := b]_{x_j y'_j}$, und damit auch $f_{x_j y'_j} \neq \emptyset$. Ein Widerspruch zur Voraussetzung. Damit gibt es kein solches j und es ist $x \equiv_q y$.

2. Fall: $a_j = 0$. Analog. ■

Der Aufwand zur Bestimmung aller Äquivalenzklassen an einer Marke q beträgt damit $O(m^2 \cdot n \cdot |G_f|)$, wobei m die Anzahl der SPDS-Variablen⁶ ist und $|G_f|$ die Größe eines BDDs für f . Die Symmetrieeigenschaft bzw. vielmehr die Asymmetrieeigenschaft in Theorem 1 kann als hinreichende Bedingung zur Nicht-Äquivalenz von SPDS-Variablen genutzt werden. So kann die Berechnung fast aller Kofaktoren aus Theorem 2 entfallen, denn für BDDs gibt es effiziente Asymmetrietests in Form von Signaturen [14, 15], welche zu zwei gegebenen Variablen x_i und x_j einer Funktion f die Asymmetrie in diesen beiden Variablen entdecken. Formal: $\tau(x_i, y_i) \Rightarrow f$ nicht partiell symmetrisch in x_i und y_i .

Lemma 1. Einsparung von Kofaktorbildungen aus Theorem 2

Sei τ ein Asymmetrietest. Dann gilt: $(\exists i : \tau(x_i, y_i)) \Rightarrow x \not\equiv_q y$.

Proof. Ang. $\exists i : \tau(x_i, y_i)$. Dann ist per Definition f in den Variablen x_i und y_i nicht partiell symmetrisch. Wegen Theorem 1 kann somit nicht $x \equiv_q y$ gelten. Es brauchen somit keine Kofaktoren aus Theorem 2 berechnet werden. ■

⁶ Summe aus lokal, global und Parameter-Variablen mit Vielfachheit bei Arrays

Wie Experimente zeigen, sind die Äquivalenzklassen sehr klein⁷ mit maximaler Größe $k \ll m$. Werden nur Signatur basierte Asymmetrietests (alle aus [14, 15]) mit Berechnungszeit $O(|G_f|)$ verwendet, so reduziert sich der Aufwand zur Bestimmung aller Äquivalenzklassen auf $O(k^2 \cdot c^2 \cdot n \cdot |G_f| + m \cdot n \cdot |G_f| + m^2 \cdot n)$ mit sehr kleinem c , welches die Anzahl der binären Variableninäquivalenzen misst, die durch keinen Asymmetrietest entdeckt wurden⁸.

Theorem 3. Identifikation von Konstanten

Eine Variable $x = (x_1 x_2 \dots x_n)$ an einer Marke q ist konstant gdw.

$$\forall i : f_{x_i} = \emptyset \vee f_{x_i} = \emptyset.$$

Ihr Wert entspricht dabei derjenigen Belegung \bar{x}_i der x_i , so dass $f_{\bar{x}_1 \bar{x}_2 \dots \bar{x}_n} \neq \emptyset$.

Die Funktion *Cudd_FindEssential* des BDD-Pakets Cudd findet genau derartige Variablenbelegungen. Für konstante Variablen müssen ebenso keine Kofaktoren aus Theorem 2 bestimmt werden.

6 Experimente und Vergleich der Äquivalenzanalysen

Die konservative Äquivalenzanalyse liefert eine Teilmenge der exakten Äquivalenzinformationen. Die Überapproximation meiner Implementation entsteht zum Einen durch Abstraktion von Arrays und Verbunden – eine Erweiterung ist möglich, wenngleich nicht nötig, wie die Tabellen 1 und 2 zeigen, da so bereits die meisten Äquivalenzen entdeckt werden und sich die Kellersystemgröße nicht mehr wesentlich verringert. Zum Anderen entsteht Ungenauigkeit durch die Abstraktion nicht realisierbarer Pfade im Modell, da alle potentiell möglichen interprozeduralen Kontrollflüsse anhand des Kontrollflussgraphen betrachtet werden (eine Überapproximation der Pfade, siehe Listing 1.3).

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, wobei zur Modellgenerierung jeweils unterschiedliche Bitbreiten (bis zu 8 Bit) zur Modellierung von Integern verwendet wurden. 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).

In den Experimenten (Tabelle 1) werden durch die konservative Äquivalenzanalyse im Durchschnitt 76% und in Extremfällen über 96% der auftretenden Äquivalenzen unter den Variablen erkannt. In Tabelle 2 ist die Modellgröße (gemessen in der Anzahl der Köpfe) miteinander verglichen. Wobei - für die unoptimierte Variante ohne **HalSPSI**, + für die konservative und * für die exakte

⁷ Durchschnitt $k=2,45$

⁸ erster Summand ergibt sich aus den äquivalenten sowie den nicht durch Asymmetrietests erkennbaren Inäquivalenzen (k und c sehr klein); der zweite aus Berechnung der Signaturen (jedes Bit jeder Variablen); der dritte aus Vergleich der Signaturen

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

Tabelle 1. Gleichheit der approximativen und exakten Äquivalenzklassen

Code-Beispiel	hd^-	hd^+	hd^*
ParameterRestrictions(7Bits)	$3,0 \cdot 10^{51}$	$2,3 \cdot 10^{49}$	$2,3 \cdot 10^{49}$
ConcreteFieldClass(8Bit)	$3,7 \cdot 10^{48}$	$1,4 \cdot 10^{40}$	$1,3 \cdot 10^{40}$
While(8Bit)	$4,3 \cdot 10^{45}$	$1,0 \cdot 10^{42}$	$1,0 \cdot 10^{42}$
Durchschnitt (alle 191)	$8,6 \cdot 10^{89}$	$5,2 \cdot 10^{52}$	$2,0 \cdot 10^{52}$

Tabelle 2. Vergleich der Modellgröße (Anzahl Köpfe) vor der Reduktion (hd^-) und nach der Reduktion für approximative (hd^+) und exakte Variante (hd^*)

Modellanalyse steht. Es zeigt sich, dass die Transformationen die Modellgröße signifikant um viele Größenordnungen verringern, was sich auf eine wesentlich geringere Modellprüfzeit auswirkt. Die konservative Analyse verbessert das Modell bereits erheblich, welches durch ein exaktes Verfahren nur noch relativ wenig optimiert werden kann. Die Laufzeit für ein exaktes Verfahren ist jedoch größer als eine Modellprüfung selbst, wo hingegen die Laufzeit für das konservative Verfahren vernachlässigbar klein gegenüber der Modellprüfzeit ist. In 5% der Modellprüfungen konnten dank vorheriger Äquivalenzanalyse und Transformation Speicherüberläufe verhindert und das Modell nun erfolgreich geprüft werden.

7 Zusammenfassung und Ausblick

Es wurden zwei Äquivalenzanalysen vorgestellt, mit denen die innere Zustandsstruktur komprimiert und Modelle bereits symbolisch vereinfacht werden können. Die in den Experimenten verwendete Variablenelimination ermöglicht damit u.a. effizienteres Testen, effizientere Testdatengenerierung mit kompakteren Testdaten und einfachere Simulationen. Exakte Äquivalenzinformationen führen dabei aber auch zu präziseren Analysen (z.B. Points-To-Analysen) für die Ausgangssprache wie C oder Java. Die Experimente zeigen, dass durch **HalSPSI** die Modellprüfung erheblich beschleunigt bzw. die Modellprüfung erst ermöglicht wird. Im Zusammenspiel mit anderen Transformationen wie z.B. Slicing, Wertebereichsreduktion oder Stotterreduktion kann die Modellprüfung in Einzelfällen sogar ganz entfallen. Es ergeben sich dann nochmals beträchtliche Verbesserungen durch Synergieeffekte. Z.B. wird bereits die Kombination der konservativen und der exakten Äquivalenzanalyse⁹ die Ingesamtlaufzeit für die exakte Äquivalenzanalyse reduzieren und immer noch zu exakten Ergebnissen führen. Unabhängig von der Reduktion durch Variablenelimination aus Abschnitt 3 kann in **HalSPSI** der SMT-Solver Yices aktiviert werden, um boolsche (Teil-)Ausdrücke

⁹ erst die schnelle konservative, dann die exakte durchführen

zu vereinfachen. Analog zum Extrahieren der exakten Äquivalenzinformationen können auf ähnliche Weise sämtliche boolesche Ausdrücke exakt auf Konstanz geprüft und vereinfacht werden, um somit Kontrollflüsse genauer vorher zu sagen.

Literatur

1. D. Liang and 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.
2. D. Liang and 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.
3. D. Richter and W. Zimmermann. *Slicing zur Modellreduktion von symbolischen Kellersystemen*. Proc. of the 24. Workshop of GI-section 'Programmiersprachen und Rechenkonzepte', University Kiel, 2007.
4. D. Richter. *Modellreduktionstechniken für symbolische Kellersysteme*. Proc. of the 25. Workshop of GI-section 'Programmiersprachen und Rechenkonzepte', University Kiel, 2008.
5. R. Mayr. *Process Rewrite Systems*. In Electronic Notes in Theoretical Computer Science, Volume 7, pp. 185-205, 1997.
6. E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, Massachusetts Institute of Technology, Cambridge, 2000.
7. S. Schwoon. *Model-Checking Pushdown Systems*. Technische Universität München, 2002.
8. S. Kiefer, S. Schwoon, and D. Suwimonteerabuth. *Introduction to Remopla*. Institute of Formal Methods in Computer Science, University of Stuttgart, 2006.
9. G. J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2004.
10. J. Esparza and S. Schwoon. *A BDD-based model checker for recursive programs*. LNCS Volume 2102, pp. 324-336, Springer, 2001.
11. J. Obdrzalek. *Formal verification of sequential systems with infinitely many states*. Master's Thesis, FI MU Brno, Masaryk University, 2001.
12. J. Obdrzalek. *Model Checking Java Using Pushdown Systems*. LFCS, University of Edinburgh, 2002.
13. O. Rüdthig, J. Knoop, and B. Steffen. *Detecting Equalities of Variables: Combining Efficiency with Precision*. Proc. of the 6th International Symposium on Static Analysis, LNCS 1694, 1999.
14. P. Molitor and C. Scholl. *Datenstrukturen und effiziente Algorithmen für die Logiksynthese kombinatorischer Schaltungen*. Teubner Verlag, 1999.
15. P. Molitor and J. Mohnke. *Equivalence Checking of Digital Circuits: Fundamentals, Principles, Methods*. Springer Netherlands, 2004.