Rekursionspräzise Intervallanalysen

Dirk Richter (richterd@informatik.uni-halle.de)

Martin-Luther-Universität Halle-Wittenberg

Zusammenfassung Intervallanalysen bestimmen zu einem gegebenen Programm konservative oder nicht-konservative Wertebereiche von Variablen. Desto genauer diese Wertebereiche bestimmt werden können, desto präziser sind die darauf basierenden Analysen wie z.B. die Laufzeitschätzung von Programmen bzw. führen zu weniger Fehlalarmen wie z.B. bei der Prüfung auf Feldzugriffe außerhalb zulässiger Indizees (ArrayOutOfBounds). Bei **un**beschränkter Rekursion und **un**beschränktem Speicher (Heap/Halde) ist das Problem der Bestimmung von Wertebereichen maximaler Genauigkeit (auch als exakte Wertebereiche bezeichnet) **unentscheidbar**. Im Folgenden wird gezeigt, wie bei **un**beschränkter Rekursion und beschränktem Speicher derartige exakte Wertebereiche in polynomieller Zeit (bzgl. Modellgröße) bestimmt werden können.

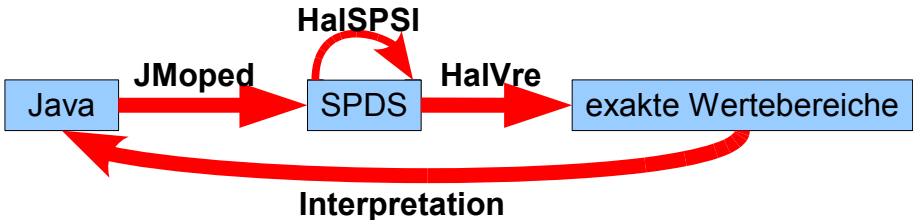
Schlüsselworte: exakte Wertebereichsanalyse, Intervallmenge, SPDS

1 Einleitung

Da viele Programmanalysen i.A. unentscheidbar für die meisten Hochsprachen sind, muss für die Analyse von der Semantik eines Programms abstrahiert werden. So ist es möglich, bestimmte Eigenschaften über das Verhalten eines Programms statisch zu bestimmen. Durch diese Abstraktion entstehen naturgemäß Ungenauigkeiten in den Analyseergebnissen. Ungenaue Analyseergebnisse führen aber in der Praxis zu ungenaueren Aussagen z.B. über die erwartete Laufzeit von Programmen oder zu unnötig großen Chip-Designs. Man ist daher bestebt, diese Ungenauigkeiten in den Analyseergebnissen zu minimieren. Eine Möglichkeit dazu bietet die korrekte Abbildung von Methodenaufrufen und Rekursion auf sog. symbolische Kellersysteme (SPDS). Die Beschränkung auf eine maximale Anzahl an Methodenaufrufen ist dann nicht nötig und vermeidet eine exponentielle Modellvergrößerung z.B. durch Inlining. Die hier betrachteten SPDS unterstützen neben einem Laufzeitkeller, Berücksichtigung von Rekursion, Prozedurparametern, lokalen und globalen Variablen allerdings **keine** Referenz-Paramter beim Methodenaufruf, da sie andernfalls turingmächtig wären und somit viele Analysen wegen dem Halteproblem unentscheidbar werden würden [1].

Programme (in unseren Experimenten Java) werden in einem ersten Schritt in solche SPDS (Rekursionsmodell) überführt (siehe Abbildung 1), welche präzise das rekursive Verhalten des Programms beschreiben. Für diese Rekursionsmodelle können, wie in Abschnitt 4 gezeigt, exakte Wertebereiche berechnet

Abbildung 1. Vorgehen beim Bestimmen der Wertebereiche von Variablen



werden. Je nach Art der Modellgenerierung lassen diese exakten Wertebereiche im Modell Rückschlüsse auf die Wertebereiche des ursprünglichen Programms zu (Abschnitt 5). Da die Bestimmung exakter Wertebereiche (nicht nur komplexitätstheoretisch) aufwändig ist, sind möglichst kleine Modelle für die Durchführbarkeit entscheidend. Derartige Verkleinerung von SPDS ist Gegenstand früherer Veröffentlichungen [2–5]. In diesen Veröffentlichungen wurde unser Werkzeug **HalSPSI** (Halle's Symbolic Pushdown System Improver) beschrieben, welches als Source-To-Source-Compiler gegebene SPDS für die Zwecke einer effizienteren Modellprüfung verbessert bzw. die Modellprüfung überhaupt erst ermöglicht. Gleichsam verbessert dieses Werkzeug die Anwendung von rekursionspräzisen Intervallanalysen (siehe Abschnitt 8).

Informationen über das Modellverhalten können z.B. durch im Programmiersprachenumfeld gängige Programmanalysen gewonnen werden. In den hier betrachteten Fällen werden für jede Variable an jeder Marke bzw. symbolischen Konfiguration[1] verschiedene Eigenschaften berechnet. Bei einer sog. Intervallanalyse werden z.B. obere und untere Schranken für die zur Laufzeit realisierbaren Variablenwerte bestimmt (siehe Abschnitt 3). Dies gelingt etwa durch eine Fixpunktberechnung mittels Transferfunktionen [6, 7] oder durch Modellierung und Lösen eines linaren Programms [8]. Im Folgendem soll eine weitere Möglichkeit (implementiert in unserem Tool Halle's Variable Range Extractor **HalVre**, siehe Abbildung 1) zur Bestimmung solcher Variablenwerte vorgestellt und mit einem eher traditionell auf Fixpunktberechnung basierendem Verfahren verglichen werden. So ist es unter Berücksichtigung von beliebigen Methodenaufrufen und Rekursion statisch möglich, Wertebereiche von Variablen vorherzusagen.

In [2] und [3] wurde gezeigt, dass verschiedene Modellanalysen im Gegensatz zur Anwendung bei herkömmlichen Programmiersprachen **entscheidbar** werden für SPDS (siehe auch [1]), was prinzipiell die Existenz sog. **exakter** Modellanalyseverfahren für SPDS nachweist. Dabei heißt eine Modellanalyse dann **exakt**, wenn das Analyseergebnis weder eine Über- noch eine Unterapproximation darstellt, also präzise das Verhalten des Modells berücksichtigt. So führt die Verknüpfung von einer Konstantenpropagation und Konstantenfaltung sowie Copy-Propagation [9] zu einer nicht exakten (konservativen) Äquivalenzanalyse [5]. Bei dieser kann es Variablen geben, die zwar gleich oder gar konstant sind,

[1] im Sinn von Programmiersprachen auch als ein Programmpunkt auffassbar

Listing 1.1. Java Beispiel zur Berechnung der Fakultät

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

dies aber nicht durch die Analyse entdeckt wird. Eine solche Analyse heißt **konservativ**, falls zudem jede durch die Analyse vorhergesagte Äquivalenz auch in jeder Simulation bzw. Ausführung auftritt. Eine exakte Äquivalenzanalyse für SPDS ist in [5] beschrieben. Äquivalenzanalysen und Intervallanalysen haben gemeinsam, dass sie beide in der Lage sind, konstante Variablen zu erkennen (siehe Abschnitt 3 und Listing 1.3). Werden beide Analysen exakt durchgeführt, so erkennen sie natürlich auch genau die gleichen Konstanten.

2 Rekursionsmodelle

$M = (S, \rightarrow, L_A)$ heißt **Kripkestruktur**, falls S und A (nicht notwendigerweise endliche) Mengen sind, $\rightarrow \subseteq S \times S$ und $L_A : S \rightarrow 2^A$. 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. Solche Kellersysteme beschreiben einen sehr großen sog. Zustandsraum, was zu dem aus der Modellprüfung bekannten Problem der Zustandsraumexplosion führt. Da Kellersysteme ihrerseit per Definition über Zustände verfügen, bezeichnen wir diesen Zustandsraum als Konfigurationenraum, wobei (p, v) **Konfiguration** heißt, falls $p \in P$ und $v \in \Gamma^*$. Eine Konfiguration (p, v) heißt **erreichbar**, falls es beginnend aus einer der Initialkonfigurationen eine Folge von Transitionen nach (p, v) gibt (ein Berechnungspfad, oder kurz **Pfad**). (p, a) heißt **Kopf** der Konfiguration (p, aw), falls $a \in \Gamma$ und $w \in \Gamma^*$. Bei der Berechnung der rekursionspräzisen Intervallanalyse wird der Umstand ausgenutzt, dass die Menge der Köpfe endlich ist, auch wenn der Konfigurationenraum unendlich sein kann. 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 $\stackrel{*}{\rightarrow}$ 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 [10]. Im Folgenden wird ein solches symbolisches Kellersystem als **Rekursionsmodell** bezeichnet, wenn es aus einer höheren Programmiersprache wie Java oder C mit unbeschränkten Methodenaufrufen gewonnen wurde.

Wie in den Listings 1.1 und 1.2 zu sehen, können SPDS mit Hilfe der Modellsprache Remopla [11] sehr kompakt beschrieben werden. Dabei wurden 16 Bit (angehängt an eine Variablendeklaration) zur Modellierung von Ganzzahlen

Listing 1.2. Mit JMoped transformiertes Remopla Beispiel von Listing 1.1

```
1  module int fac(int v0(16))
2  {
3  int s0(16);
4  int s1(16);
5  int s2(16);
6  fac_I:   s0=v0, s1=s0, s2=s1;
7  fac_I1:  s0=1, s1=s0, s2=s1;
8  fac_I2:  if
9           :: s1>s0 -> goto fac_I7, s0=s2;
10          :: else -> s0=s2;
11          fi;
12 fac_I5:  s0=1, s1=s0, s2=s1;
13 fac_I6:  return s0;
14 fac_I7:  s0=v0, s1=s0, s2=s1;
15 fac_I8:  s0=v0, s1=s0, s2=s1;
16 fac_I9:  s0=1, s1=s0, s2=s1;
17 fac_I10: if
18          :: s1>=s0 -> s0=s1-s0, s1=s2;
19          :: else -> s0=(65536-(s0-s1)%65536)%65536, s1=s2;
20          fi;
21 fac_I11: s0=fac_I(s0);
22 fac_I14: s0=(s1*s0)%65536, s1=s2;
23 fac_I15: return s0;
24 }
```

(Integern) verwendet, um das Verhalten des Java Programms nachzubilden. 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 Programm enthaltenen Methodenaufrufe durch das Modell 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[2].

Formal kann Remopla wie folgt zusammengefasst werden. Seien $Vars_q := globs \cup loc_q$, $EXPR_{Vars_q}$ arithmetische Ausdrücke über diesen Variablen und

[2] Ziel früherer Arbeiten [2–5] war es, das dazu nötige Kelleralphabet und die benötigten Kellerzustände bereits symbolisch a priori zu verringern.

$\Phi^q : Vars_q \to \mathbb{N}$ eine Variablenbelegung, welche aus dem Kopf einer Konfiguration bestimmt wird, sowie $[\![e]\!]_{\Phi^q}$ die Auswertung eines Ausdrucks $e \in EXPR_{Vars_q}$ mittels der Variablenbelegung Φ^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 $[\![e_i]\!]_{\Phi^q}$ zuweist.
- **p(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 $[\![e]\!]_{\Phi^q}$.
- **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 $[\![\mathrm{b}_i]\!]_{\Phi^q} \in \{0, 1\}$, die eine zufällig ausgewählte Anweisung s_k mit $[\![\mathrm{b}_k]\!]_{\Phi^q} = 1$ 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 [12–14, 2] verwiesen.

3 Konservative Wertebereichsanalyse für SPDS

In userem Werkzeug **HalSPSI** wurde eine sog. konservative Wertebereichsanalyse durch Kombination von *Range Propagation* und *Range Analysis* [15] basierend auf einer Fixpunktberechnung umgesetzt. Sie ist interprozedural[3] sowie flusssensitivt[4]. Konservativ bedeutet hier, dass nicht für jeden durch **HalSPSI** vorgergesagter Wert einer Variablen es einen Simulationslauf des SPDS geben muss. Wird aber ein gewisser Wert einer Variablen bei einem Simulationslauf angenommen[5], so ist dieser in dem konservativen Analyseergebnis enthalten. Techniken wie Aufweitung (*Widening*) und Einengung (*Narrowing*) [16, 7] können zur Konvergenzverbesserung der Fixpunktberechnung eingesetzt werden und tragen zu einer Verstärkung der Ungenauigkeit der Analyse bei. Da ihr Einsatz für SPDS nicht zwingend erforderlich ist (siehe Abschnitt 7), wurde aus Präzisionsgründen in **HalSPSI** darauf verzichtet. Ziel von Intervallanalysen ist die Bestimmung von oberen und unteren Schranken für die zur Laufzeit realisierbaren Wertebereiche. In der in **HalSPSI** implementierten Wertebereichsanalyse wird nicht nur eine solche obere und untere Schranke je Variable bestimmt, sondern vielmehr der ganze potentielle Werteraum (alle möglichen konkreten Variablenbelegungen) durch eine Vielzahl von Intervallen je Variable und Programmpunkt abgebildet. Eine obere und untere Schranke kann daraus abgeleitet werden.

Durch diese Eigenschaften wird eine Präzisionssteigerung der Analyse gegenüber konventioneller Intervallanalysen erzielt, welche lediglich **eine** obere und untere Schranke beachten.

[3] Im Gegensatz zu intraprozeduralen Analysen (nur innerhalb von Prozeduren) werden hier Analyseergebnisse über Prozedurgrenzen transportiert.

[4] Im Gegensatz zu flussinsensitiven Analysen wird hier die zeitliche Abfolge von Anweisungen berücksichtigt.

[5] Dieser wird im Folgendem auch kurz als **realisierbar** bezeichnet.

Abbildung 2. Beispiele zur Operatorinterpretation bei Intervallmengen

```
{(3,1000)} << {(1)} = {(6), (8), (10), (12), ..., (2000)}
{(3,1000)} * {(3,1000)} = {(9), (12), (15, 16), (18), ..., (1000000)}
{(1000)} / {(3,70)} = {(142), (144), (147), (149), ... ,(3333)}
{(1000)} >> {(0,10)) = {(0,1), (3), (7), (15), ..., (1000)}
{(1001)} % {(10,100)} = {(0, 2), (5) ,(7) ,(9), ... (77), (81)}
```

Listing 1.3. Mit Konstantenfaltung/-Propagation nicht erkennbare Konstante y

```
L: y = x/2;
if
  :: (x = 0) -> x = 1; goto L;
  :: else -> break;
fi
```

Im Folgenden schreiben wir $x@l \in \{(a_1, b_1), (a_2, b_2), \ldots, (a_k, b_k)\}$, wenn die Analyse bestimmt hat, dass die Variable x an der Konfiguration l in einem der Intervalle $[a_i, b_i]$ liegt. Wir bezeichnen $\{(a_1, b_1), (a_2, b_2), \ldots, (a_k, b_k)\}$ als **Intervallmenge** und schreiben statt (a, b) auch (a), falls $a = b$. Um arithmetische Operationen möglichst präzise abstrakt zu interpretieren, müssen diese auf derartige Intervallmengen verallgemeinert werden. Bei den Operationen $*(Multiplikation), /(Division), \%(Division\ mit\ Rest), <<\ (Shift\ links)$ als auch $>>\ (Shift\ rechts)$ können die Intervalle ggf. in einzelne Werte „zerfallen“, wie die Beispiele aus Abbildung 2 zeigen. Insbesodere das zweite Beispiel erzeugt bereits über 173000 Intervalle. Dies ist vermutlich einer der Gründe, weswegen in Intervallanalysen in der Litertur gern von solchen Rechenoperationen abstrahiert wird [17]. **HalSPSI** hingegen (wie auch **HalVre**) interpretiert diese Operationen vollständig, da dies in der Praxis selten vorkommende Artefakte sind und sich zudem die Anzahl der Intervalle auf die in der Tabelle 1 angegebene Anzahl beschänken. Vielmehr steigt statt dessen die gewonnene Präzision.

Tabelle 1. Maximale Anzahl an Intervalle verschiedener Bit-Größen

Integer-Bitbreite	2	3	4	5	6	7	8	9	10	14	16
Max. Intervalle	2	4	8	16	32	64	128	256	512	8192	32768

Wie Listing 1.3 zeigt, können durch Intervallanalysen auch Konstanten identifiziert werden, welche mit anderen einfacheren Methoden nicht erkannt werden. Die Variable y hat im Listing 1.3 stets den Wert 0, was bereits durch eine ein-

fache Intervallanalyse erkannt wird, nicht jedoch durch Konstantenfaltung oder -Propagation.

Durch Kontextsensitivität wie z.B. durch Context-Cloning [18] (Vorsicht: Modellvergrößerung) oder Berücksichtigung von Call-Strings [19, 20] kann die Präzision der konservativen Wertebereichsanalye bezüglich der Rekursion weiter verbessert werden. Vollständige Kontextsensitivität jedoch ohne Modellvergrößerung kann durch exakte Wertebereichsanalysen erreicht werden, weshalb im Folgenden eine solche beschrieben werden soll.

4 Exakte Wertebereichsanalyse für SPDS

Im Folgenden wird kurz beschrieben, wie unser Werkzeug **HalVre** exakte Intervallmengen für SPDS bestimmt.

Lemma 1. *(**Entscheidbarkeit der Intervallanalyse**)*
*Seien $a, b \in \mathbb{R}$. Dann ist für SPDS **entscheidbar**, ob für eine Variable x an einer erreichbaren Konfiguration p stets gilt: $a \leq x \leq b$.*

Proof. Reduktion auf Erreichbarkeitsproblem[6] (analog Äquivalenzanalyse in [3]).

Theorem 1. *(**Entscheidbarkeit der Wertebereichsanalyse**)*
*Seien $a_i, b_i \in \mathbb{R}$ mit $i \in \{1, 2, \ldots, n\}$. Dann ist für SPDS **entscheidbar**, ob für eine Variable x an einer erreichbaren Konfiguration p stets gilt: $\exists i : a_i \leq x \leq b_i$.*

Proof. trivial nach Lemma 1.

Auf Grund dieser Entscheidbarkeitsaussage existiert eine kleinste Intervallmenge, welche in diesem Sinne die wenigsten Variablenwerte überdeckt, dennoch konservativ ist und von jeder exakten Wertebereichsanalyse berechnet wird. So sind exakte Intervallanalysen wie auch exakte Wertebereichsanalysen stets automatisch interprozedural, kontextsensitiv, flusssensitiv und pfadsensitiv etc..

Theorem 2. *(**Komplexität exakter Wertebereichsanalysen**)*
Eine exakte Wertebereichsanalyse für ein SPDS ist komplexitättheoretisch mindestens so schwer wie die Erreichbarkeitsprüfung von Konfigurationen[7].

Proof. Informal: sie löst die Erreichbarkeitsfrage für alle Marken des SPDS. □

In [10] wird beschrieben, wie zu einem gegebenen SPDS P symbolisch ein endlicher Automat $Post^*(P)$ konstruiert werden kann, welcher die Menge aller erreichbaren Konfigurationen aus gegebenen Initialkonfigurationen akzeptiert. Mittels $Post^*(P)$ kann zu jeder SPDS Marke q eine charakteristische Funktion $f^q : \{0,1\}^* \to \{0,1\}$ für die exakten Variablenbelegungen für lokale und globale

[6] Die Frage, ob es im SPDS einen Pfad (Transitionenfolge) von einer der Anfangskonfigurationen zu einer gegebenen Konfiguration, Kopf oder Marke gibt.
[7] Modellprüfung

Listing 1.4. Definierte Variablen a_i mit Dimensionen d_i und Bitbreiten k_i

```
int a1[d1](k1);
int a2[d2](k2);
...
int am[dm](km);
```

Variablen[8] bestimmt werden durch Beschränkung der Transitionen aus $Post^*(P)$ auf Köpfe (Schritt 1). Aus diesen Köpfen können dann in einem zweiten Schritt die entsprechenden Wertebereiche der Variablen ermittelt werden.

Zur Veranschaulichung seien nun in einer Methode P an der Marke q die lokalen und globalen Variablen a_i (einschließlich Parameter) wie in Listing 1.4 gegeben mit Dimensionen d_i (a_i ist ein Array gdw. $d_i > 1$) und Bitbreiten k_i:

Dann ist mit $x_{i,d,k} \in \{0,1\}$ für $i \in \{1,2,\dots m\}, d \in \{1,2,\dots d_i\}, k \in \{1,2,\dots k_i\}$

$$
\begin{aligned}
f^q(\quad & x_{1,1,1}, && x_{1,1,2}, && x_{1,1,3}, && \dots, && x_{1,1,k_1}, \\
& x_{1,2,1}, && x_{1,2,2}, && x_{1,2,3}, && \dots, && x_{1,2,k_1}, \\
& \dots \\
& x_{1,d_1,1}, && x_{1,d_1,2}, && x_{1,d_1,3}, && \dots, && x_{1,d_1,k_1}, \\
& x_{2,1,1}, && x_{2,1,2}, && x_{2,1,3}, && \dots, && x_{2,1,k_2}, \\
& x_{2,2,1}, && x_{2,2,2}, && x_{2,2,3}, && \dots, && x_{2,2,k_2}, \\
& \dots \\
& x_{2,d_2,1}, && x_{2,d_2,2}, && x_{2,d_2,3}, && \dots, && x_{2,d_2,k_2}, \\
& \dots \\
& \dots \\
& x_{m-1,d_{m-1},1}, && x_{m-1,d_{m-1},2}, && x_{m-1,d_{m-1},3}, && \dots, && x_{m-1,d_{m-1},k_{m-1}}, \\
& x_{m,1,1}, && x_{m,1,2}, && x_{m,1,3}, && \dots, && x_{m,1,k_m}, \\
& x_{m,2,1}, && x_{m,2,2}, && x_{m,2,3}, && \dots, && x_{m,2,k_m}, \\
& \dots \\
& x_{m,d_m,1}, && x_{m,d_m,2}, && x_{m,d_m,3}, && \dots, && x_{m,d_m,k_m}) \quad = 1
\end{aligned}
\tag{1}
$$

gdw. es eine **realisierbare** Wertbelegung Φ an der Marke q gibt, so dass $\forall i \in \{1,2,\dots,m\}, \forall d \in \{0,2,\dots,d_i-1\}$:

$$\Phi(a_i[d]) = \sum_{b=1}^{k_i} x_{i,d,b} \cdot 2^{k_i - b}. \tag{2}$$

Φ beschreibt somit **einen** realisierbaren Kopf und f^q genau die Menge **aller** realisierbaren Köpfe an der Marke q. Um nun aus f^q entsprechende Intervallmengen zu extrahieren, wird ein Zusammenhang mit sog. Kofaktoren ausgenutzt. Dabei heißt eine Funktion $f_{x_i} := f(x_1, ...x_{i-1}, 1, x_{i+1}, ...x_n)$ **positiver Kofaktor** und $f_{x_i'} := f(x_1, ...x_{i-1}, 0, x_{i+1}, ...x_n)$ **negativer Kofaktor** von f. Man bezeichnet $f_{ab} := (f_a)_b$ als iterierten Kofaktor. Wir schreiben $f[a = \alpha]$ für den

[8] einschließlich Arrayvariablen mit entsprechender Vielfachheit und Verbunden

Kofaktor f_a, falls $\alpha = 1$ bzw. $f_{a'}$, falls $\alpha = 0$. Die **ON-Menge** von f ist eine Teilmenge des Definitionsbereichs $\{0,1\}^n$ von f und besteht aus den Elementen $\alpha \in \{0,1\}^n$, für die gilt $f(\alpha) = 1$. Weitere Details finden sich in [21, 22].

Lemma 2. *(**Exakte Wertbelegungsmenge einer Variable**)*
Für die charakteristische Funktion der Menge aller ralisierbaren Wertebelegungen $f^{a_i[d],q} : \{0,1\}^{k_i} \to \{0,1\}$ einer[9] *Variable $a_i[d]$ an Marke q, gilt: $f^{a_i[d],q}(x_1, x_2, \ldots x_{k_i}) = 1$ gdw. $f^q[x_{i,d,1} = x_1][x_{i,d,2} = x_2] \ldots [x_{i,d,k_i} = x_{k_i}] \neq \emptyset$.*

Proof. Ergibt sich aus den Definitionen. □

Nun muss die Funktion $f^{a_i[d],q}$ gar nicht explizit berechnet werden, wenn die Intervallmengen nicht als z.B. ROBDD[10] benötigt werden. Insbesondere auch dann nicht, wenn lediglich eine untere und obere Schranke für Variablen zu bestimmen ist. Statt dessen genügt im Falle der Repräsentation von f^q als ROBDD die Prüfung von $f^q[x_{i,d,1} = x_1][x_{i,d,2} = x_2] \ldots [x_{i,d,k_i} = x_{k_i}]$ auf Leerheit (also bei ROBDDs der Test auf 0). Es muss lediglich die Intervallmenge (ON-Menge) von $f^{a_i[d],q}$ bestimmt werden, welche aber auch direkt aus f^q extrahiert werden kann. Ein zusätzlicher Schritt zur Bestimmung der Intervallmengen aus $f^{a_i[d],q}$ und auch die explizite Berechnung von $f^{a_i[d],q}$ ist damit nicht nötig.

Theorem 3. *(**Polynomielle Laufzeit exakter Wertebereichsanalysen**)*
Die Bestimmung exakter Wertebereiche in Form von Intervallmengen eines Kellersystems $(P, \Delta, \hookrightarrow)$ benötigt polynomielle Zeit in der Größe dieses Kellersystems.

Proof. Die Berechnung des Post*-Automaten zu einem gegebenen Kellersystem $(P, \Delta, \hookrightarrow)$ benötigt die Zeit $O(|P||\Delta|(|\hookrightarrow| + |\Delta|))$ (siehe [23]). Der Post*-Automat enthält nach Konstruktion Transitionen von jedem Initialzustand s zur Marke q, welche sämtliche realisierbaren Pfade des Modells zusammenfassen. Zur Bestimmung der charakteristischen Funktionen f^q sind daher lediglich die Zielköpfe dieser Transitionen zu extrahieren, was für alle Marken zusammen insgesamt amortisiert O(Größe des Post*-Automaten als ROBDD) Zeit benötigt. Je Variable ist es dann im ROBDD von f^q nötig, die ON-Menge (repräsentiert als Intervallmenge) von $f^{a_i[d],q}$ zu bestimmen, was leicht modifiziert[11] nach [21, 22] polynomiell in der ROBDD-Größe von f^q geht. Also benötigt das Verfahren insgesamt eine Zeit: polynomiell in der Modellgröße. □

Sind lediglich untere und obere Schranken für SPDS-Variablen zu bestimmen, so kann auch die Bestimmung der ON-Menge von $f^{a_i[d],q}$ entfallen. In diesem Fall genügt ein einziger Tiefensuchlauf im ROBDD von f^q aus, um aus der ON-Menge von $f^{a_i[d],q}$ die kleinste und größte Variablenbelegung α und β zu bestimmen, so dass gilt ($\leq$ bezeichne dabei die lexikographische Ordnung):

$$f^{a_i[d],q}(\alpha) = f^{a_i[d],q}(\beta) = 1 \wedge \forall \zeta : ((f^{a_i[d],q}(\zeta) = 1) \Rightarrow \alpha \leq \zeta \leq \beta). \quad (3)$$

[9] wie in Listing 1.4 definierten
[10] reduziert geordnetes binäres Entscheidungsdiagramm [21, 22]
[11] Die ON-Menge für einen Teil der Variablen des Definitionsbereichs kann ähnlich berechnet werden, wie die ON-Menge selbst.

5 Rückinterpretation in die Hochsprache

Wie in Listing 1.2 am Beispiel des übergebenen Parameters n zu sehen, korrelieren bei der Modellgenerierung mittels JMoped Modellvariablen direkt mit Programmvariablen. Lokale und Paramtervariablen werden bei JMoped konventionsgemäß in den Remopla-Modellen mit $v0, v1, \ldots$ bezeichnet, wobei zuerst für alle Paramtervariablen Namen vergeben werden. Die Reihenfolge der Definitionen entspricht der im Quellprogramm. Bei Anwendung von Optimierungen durch **HalSPSI** kann diese Ordnung allerdings verloren gehen. Mittels zusätzlicher Namensverlängerung bei der Modellgenerierung ist eine problemlose Zuordnung der exakten Wertebereiche aus den SPDS zu den Variablen des Quellprogramms möglich. Dies gilt aber wiederum auch nur dann, wenn diese Variablen nicht durch **HalSPSI** wegoptimiert wurden[12]. Zu den $v0, v1, \ldots$ kommen Hilfvariablen $s0, s1, \ldots$ hinzu, welche den lokalen Operanden-Stack einer Methode in Java simulieren. Da dieser Operanden-Stack a priori bereits im Bytecode in seiner Tiefe beschränkt ist, wird zur Simulation des Operanden-Stacks kein realer Stack des Kellersystems benötigt. Die Wertebereiche der Variablen $s0, s1, \ldots$ sind insofern also für eine Rückinterpretation uninteressant. Die Wertebereiche der restlichen Variablen sind aber nur dann auch exakt für das Quellprogramm, wenn das generierte Modell das Programmverhalten zu 100% beschreibt. Für das Beispiel in Listing 1.2 trifft dies nur zu, wenn n und der Rückgabewert der Funktion fac in Listing 1.1 stets nichtnegativ und kleiner als 65536 sind. Neben vielen weiteren Beispielen, können ganze Klassen oder große Teile von Programmiersprachen direkt durch SPDS ausgedrückt werden. So wurde in [1] eine Teilsprache von ISO C direkt als SPDS formuliert. Letztendlich muss es aber für turingmächtige Programmiersprachen stets Beispiele geben, welche durch die Modellierung als SPDS Präzision in ihrer Semantik verlieren. In diesen Fällen muss bei der Modellgenerierung auf den Erhalt der Konservativität geachtet werden. Auch JMoped muss z.B. dazu derartig in der Modellgenerierung angepasst werden, dass z.B. statt auftretender Speicherüberläufe im Modell auf Grund eines zu klein modellierten Heaps die SPDS-Pfade mit z.B. undefinierten Datenwerten fortgesetzt werden. So entstehen zwar größere exakte Intervallmengen für die Modelle, die dann aber als immer noch konservativ auch für das Quellprogramm verwendet werden können.

6 Experimente

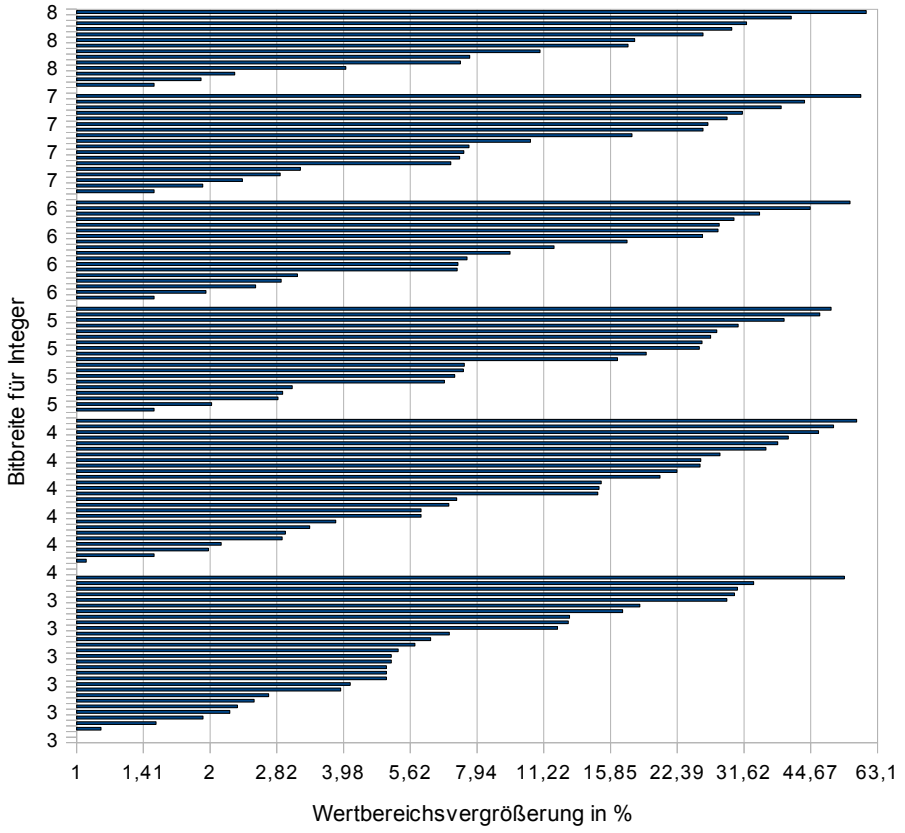
Als Grundlage für die Experimente dienten 191 von 240 Remopla-Modellen, zu denen exakte Wertebereiche berechnet werden konnten. Die Modelle wurden mittels JMoped aus Java Beispielen gewonnen, welche zum größten Teil zu den Benchmark-Instanzen der Werkzeuge JMoped bzw. Moped gehören. Zur Modellgenerierung wurden jeweils unterschiedliche Bitbreiten (bis zu 8 Bit) zur Modellierung von Ganzzahlen (Integer) verwendet. Aufgabe war es, zu diesen Modellen

[12] Änderungen durch Optimierungen sind dann rückverfolgbar zu halten.

einmal mit der konservativen und einmal mit der exakten Wertebereichsanalyse Intervallmengen zu bestimmen. Durchgeführt wurden die Experimente mit einem AMD 64 X2 4200+ mit 2 GB Hauptspeicher unter Linux (Kernel 2.6.27).

Die Laufzeiten der konservativen und exakten Wertebereichsanalyse unterscheiden sich um einige Größenordnungen. Die Berechnung der exakten Intervallmengen benötigte (wie auch theoretisch nachgewiesen in Theorem 2) oft mehr Zeit als eine Modellprüfung, während das konservative Verfahren sehr viel kleinere Laufzeiten stets im Sekundenbereich liefert und auch deutlich größere Modelle zu analysieren vermag.

Abbildung 3. Approximationsfehler des Wertebereichs der konservativen Analyse



Durch die konservative Wertebereichsanalyse werden in den Beispielen im Durchschnitt die Intervallmengen um 11.65% größer[13]. Diese Vergrößerung entspricht dem Approximationsfehler, welcher in Abbildung 3 für die einzelnen Modelle (ab 3 Bit) zu sehen ist. Desto größer die Bitbreite für Integer modelliert

[13] Im Durchschnitt beträgt die Präzision der konservativen Analyse also 88,35%.

wurde, desto seltener konnten wegen Speicherüberläufen exakte Intervallmengen bestimmt werden. In 6% der Fälle (vorranging kleine Bitbreiten) konnten die Intervallmengen bereits durch die konservative Wertebereichsanalyse exakt vorhergesagt werden (12 von 191 Fälle). Wie in Abbildung 3 auch zu erkennen, steigt bei Erhöhung der modellierten Bitbreite in einem Modell der Approximationsfehler nur leicht an. Dies läßt auf einen relativ stabil bleibenden Approximationsfehler bei größer werdenen Modellen schließen. Der Approximationsfehler ändert sich nicht signifikant, falls man lediglich die untere und obere Intervallgrenze (Minimal- und Maximalwert aus den Intervallmengen) zur Berechnung des Approximationsfehlers verwendet.

Insgesamt zeigt sich, dass mit rekursionspräzisen Analysen zur Bestimmung von Intervallmengen die Qualität des Analyseergebnises in vielen Fällen nur noch relativ wenig verbessert werden kann (man beachte die logarithmische Skala der X-Achse in Abbildung 3).

7 Verwandte Arbeiten

Ein Problem gängiger Intervallanalysen ist die Fehlabstraktion der Semantik der Ringarithmetik mittels unbeschränkter Datentypen, welches sich gerade bei kleineren Bitbreiten, wie sie bei der Modellprüfung eingesetzt werden, zum Tragen kommt. So missachten z.B. Intervallanalysen, wenn sie mittels linearen Programmen Variablen-Grenzen bestimmen, Ringarithmetik [8]. Sind aber im Modell unbeschränkte Ganzzahlen erlaubt, so kann für Fixpunktalgorithmen keine Konvergenz mehr garantiert werden [7]. Mittels Aufweitung (widening) und Einengung (narrowing) [16, 7] oder Beschränkung der Fixpunktiterationen [24] kann Konvergenz nur noch mit einhergehendem Qualitätsverlust erreicht werden. So erlangt man aber nur noch eine ungenaue sichere Überapproximation der Analyseergebnisse [25]. Die hier betrachteten SPDS verfügen über beschränkte Ganzzahlen, womit Konvergenz auch ohne Aufweitung und Einengung möglich ist. Nach [26] werden zudem z.B. Aufweitungs-Operatoren auch von Hand erstellt und müssen bei unzureichender Genauigkeit immer wieder neu angepasst werden. Daher verwendet **HalSPSI** einen Fixpunktansatz (wie auch Delzanno [6]) für die konservative Intervallanalyse mit präzisen Ergebnissen ohne Aufweitung bzw. Einengung, welche es zudem gegenüber von Ungleichungssystemen (z.B. Polyeder[14] [26]) ermöglicht, mehrere Intervalle je Variable gleichzeitig in Form von Intervallmengen zu behandeln und nicht nur eine untere und eine obere Schranke liefert [24]. Auch unterliegt der Ansatz keinen Monotonie-Einschränkungen wie in [25] oder Einschränkungen auf eine geringe Anzahl spezieller Operationen. Bodik, Gupta und Sarkar [27] berücksichtigen z.B. nur Zuweisung einer Konstante und Addition einer Konstante. Vielmehr wird die Ausdrucksauswertung hier ohne Einschränkungen der Operanden oder Operationen wie Multiplikation, Division und Modulo (Restbestimmung) interpretiert. Eingesetzt werden Intervallanalysen u.A. bei der Elimination überflüssiger Prüfungen von Feldzugriffen (Array Bound Checks) [24]. Cousot, Halbwachs und Schwarz (et al.) nutzen dazu

[14] engl. polyhedra

Abstrakte Interpretation mit statischer Datenfluss-Analyse [6], um überflüssige Prüfungen von Feldzugriffen zu identifizieren.

Insgesamt ist aber die maximal mögliche Bestimmung von Wertebereichen von Variablen in all diesen Publikationen nicht das Hauptziel. Unsere Ansätze sind zudem weitgehend unabhängig von der Quellsprache und können im Gegensatz zu Hochsprachen exakt durchgeführt werden [3]. Dies führt zu sehr präzisen Analysen für die Ausgangssprache, welche nicht auf endliche Modelle beschränkt sind und beliebige Rekursion berücksichtigen.

8 Zusammenfassung und Ausblick

Es wurde eine Methode vorgestellt, die in Experimenten durch Speicherbegrenzung in Java-Prgrammen sehr genaue Wertebereiche der Variablen bestimmt. Die Java-Programme wurden dazu zunächst in ein Rekursionsmodell überführt, welches präzise das rekursive Verhalten des Programms beschreibt. Für diese Rekursionsmodelle konnten exakte Wertebereiche berechnet werden. Je nach Art der Modellgenerierung lassen diese Wertebereiche im Modell Rückschlüsse auf die Wertebereiche des ursprünglichen Java-Programms zu. Da die Bestimmung exakter Wertebereiche (nicht nur komplexitätstheoretisch) aufwändig ist, sind möglichst kleine Modelle für die Durchführbarkeit entscheidend. Die Modellgenerierung mittels JMoped ermöglicht die Modellierung ganzzahliger Datentypen mit nur wenigen Bits und beschränkt somit automatisch den adressierbaren Speicher, in welchem sich die zur Laufzeit erzeugten Objekte befinden, was bereits in 80% (191 von 240) der untersuchten Fälle genügte. Eine Modell-Verkleinerung bzw. -Verbesserung führt in diesen Fällen dann nur noch zu einer Beschleunigung der rekursionspräzisen Intervallanalyse. Wie auch in den Experimenten zu erkennen war, können mit konservativen Analysen, welche um Größenordnungen schneller sind als exakte Verfahren[15], bereits 88% Präzision erreicht werden. Dennoch können nicht nur approximative, sondern auch exakte Methoden genutzt werden, um präziser das Verhalten und Eigenschaften von SPDS und damit von C, Java oder anderen Programmen vorherzusagen. Zur Beschleunigung der exakten Wertebereichsanalyse können zusätzlich Reduktionsverfahren unseres Werkzeugs **HalSPSI** eingesetzt werden, wie z.B. die Wertebereichsreduktion, Stotterreduktion oder Slicing [2–5]. So können dann noch größere Modelle analysiert und Intervallmengen der Modelle schneller berechnet werden.

Genauer zu betrachten bleibt eine bezüglich Wertebereichsanalyse nachweislich konservative Modellgenerierung aus höheren Programmiersprachen (und nicht nur Java wie in Abschnitt 5). Für z.B. eingebette Systeme, welche u.U. nur einen Teil von ISO C verwenden (wie in [1] erläutert), degeneriert eine solche Modellgenerierung im Idealfall zur Identität. Ein Nachweis zum Erhalt von Konservativität kann dann entfallen.

[15] Obwohl die vorgestellte exakte Wertebereichsanalyse nur polynomielle Laufzeit benötigt.

Literatur

1. R. Kirner, W. Zimmermann, D. Richter: On Undecidability Results of Real Programming Languages. Im Rahmen des 15. Kolloquium Programmiersprachen und Grundlagen der Programmierung (KPS) (2009)
2. D. Richter, W. Zimmermann: Slicing zur Modellreduktion von symbolischen Kellersystemen. Proc. of the 24. Workshop of GI-section 'Programmiersprachen und Rechenkonzepte', University Kiel (2007)
3. D. Richter: Modellreduktionstechniken für symbolische Kellersysteme. Proc. of the 25. Workshop 'Programmiersprachen und Rechenkonzepte', University Kiel (2008)
4. D. Richter, W. Zimmermann: Variablenelimination für symbolische Modelle. Im Rahmen der 39. GI-Jahrestagung zum 4. Workshop 'Modellbasiertes Testen', Lübeck (2009)
5. D. Richter: Äquivalenzanalysen - exakt oder nicht - im Vergleich. Im Rahmen des 26. Workshops 'Programmiersprachen und Rechenkonzepte', University Kiel (2009)
6. Thi Viet Nga Nguyen, Francois Irigoin: Interprocedural program analyses for efficient array bound checking. In: 2nd International Workshop on Automated Program Analysis, Testing and Verification (WAPATV), ACM Press New York (2001) http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.11.1279.
7. Patrick Cousot, Radhia Cousot: Comparing the galois connection and widening/narrowing approaches to abstract interpretation. In: Programming Language Implementation and Logic Programming. Volume 631 of Lecture Notes in Computer Science (LNCS)., Springer-Verlag Berlin Heidelberg (1992) 269–295 http://www.springerlink.com/content/p869x76457527706/.
8. Jeffery Von Ronne, Kleanthis Psarris, David Niedzielski: Verifiable Range Analysis Annotations for Array Bounds Check Elimination. 13th International Workshop on Compilers for Parallel Computers (CPC). Lisbon, Portugal. (2007)
9. S. Muchnick: Advanced Compiler Design and Implementation. Morgan Kaufmann (1997)
10. S. Schwoon: Model-Checking Pushdown Systems. Technische Universität München (2002)
11. S. Kiefer, S. Schwoon, D. Suwimonteerabuth: Introduction to Remopla. Institute of Formal Methods in Computer Science, University of Stuttgart (2006)
12. J. Esparza, S. Schwoon: A BDD-based model checker for recursive programs. LNCS Volume 2102, 324-336, Springer (2001)
13. J. Obdrzalek: Formal verification of sequential systems with infinitely many states. Master's Thesis, FI MU Brno, Masaryk University (2001)
14. J. Obdrzalek. In: Model Checking Java Using Pushdown Systems. LFCS, University of Edinburgh (2002)
15. William H. Harrison: Compiler Analysis of the Value Ranges for Variables. In IEEE Transactions on Software Engineering, Vol 3, Issue 3, 243-250 (1977)
16. Patrick Cousot, Nicolas Halbwachs: Automatic discovery of linear restraints among variables of a program. In: Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, ACM Press New York (1978) 84–96 http://portal.acm.org/citation.cfm?id=512770.
17. Thomas Gawlitza, Jan Reineke, Helmut Seidl, Reinhard Wilhelm: Polynomial Precise Interval Analysis Revisited. Technical Report, TU Muenchen (2006)
18. J. Whaley, M.S.L.: Cloning-Based Context-Sensitive Pointer Alias Analysis Using Binary Decision Diagrams. Proc. of the SIGPLAN 2004 conference on Programming language design and implementation, 131-144, ACM (2004)

19. Florian Martin: Experimental comparison of call string and functional approaches to interprocedural analysis. In: Compiler Construction. Volume 1575., Lecture Notes in Computer Science (LNCS) (1999) http://www.springerlink.com/content/u5xkkgvf40tnaq9u/.
20. Bageshri Karkare, Uday P. Khedker: An improved bound for call strings based interprocedural analysis of bit vector frameworks. In: ACM Transactions on Programming Languages and Systems (TOPLAS). Volume 29(6)., ACM Press New York (2007) http://portal.acm.org/citation.cfm?id=1286829.
21. P. Molitor, C. Scholl: Datenstrukturen und effiziente Algorithmen für die Logiksynthese kombinatorischer Schaltungen. Teubner Verlag (1999)
22. P. Molitor, J. Mohnke: Equivalence Checking of Digital Circuits: Fundamentals, Principles, Methods. Springer Netherlands (2004)
23. J. Esparza, D. Hansel, P. Rossmanith, S. Schwoon: Efficient algorithms for model checking pushdown systems. Proc. of the 12th International Conference on Computer Aided Verification, LNCS 1855 (2000)
24. Radu Rugina, Martin C. Rinard: Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In: ACM Transactions on Programming Languages and Systems (TOPLAS). Volume 27(2)., ACM Press New York (2005) 185–235 http://portal.acm.org/citation.cfm?id=1057388.
25. Thomas Gawlitza, Jan Reineke, Helmut Seidl, Reinhard Wilhelm: Polynomial precise interval analysis revisited. Technical Report, TU München, Germany (2006) http://rw4.cs.uni-sb.de/publications.shtml.
26. Chao Wang, Zijiang Yang, Aarti Gupta, Franjo Ivancic: Using counterexamples for improving the precision of reachability computation with polyhedra. In: Proceedings of the 19th International Conference on Computer Aided Verification. Volume 4590 of Lecture Notes in Computer Science (LNCS)., Springer-Verlag Berlin Heidelberg (2007) 352–365 http://www.springerlink.com/content/83835rn353287342/.
27. Rastislav Bodik, Rajiv Gupta, Vivek Sarkar: Abcd: eliminating array bounds checks on demand. In: Proceedings of the ACM SIGPLAN conference on Programming language design and implementation, ACM Press New York (2000) 321–333 http://portal.acm.org/citation.cfm?id=349342.