

Äquivalenzanalysen - exakt oder nicht - im Vergleich

Dirk Richter

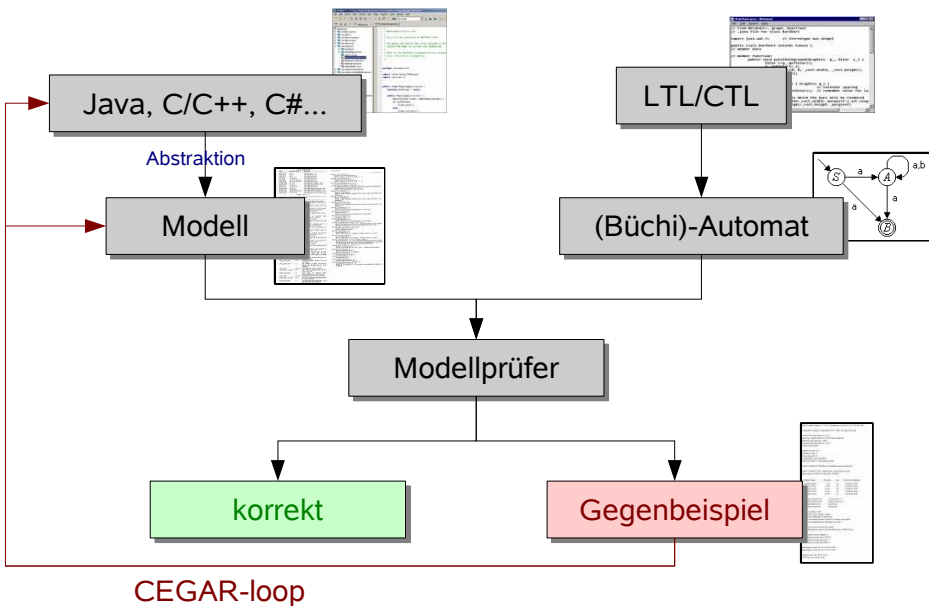
26. Workshop Programmiersprachen und Rechenkonzepte
Vortrag am 04.05.2009

Martin-Luther-Universität Halle-Wittenberg
Naturwissenschaftliche Fakultät III, Institut für Informatik,
Lehrstuhl Softwaretechnik und Programmiersprachen,
<http://swt.informatik.uni-halle.de/>

Inhaltsverzeichnis

- 1 Überblick
 - Quellcode → PDS
 - Remopla (SPDS) - Beispiel (mit JMoped generiert)
- 2 Motivation: Variablenelimination
 - nicht exakte Äquivalenzanalysen
 - exakte Äquivalenzanalysen
- 3 Experimente
- 4 weitere Techniken
- 5 vergleichbare Ansätze

Software Modellprüfung (engl. Software Model Checking)





Modellgröße und Komplexität entscheidend für

- Softwaremodellprüfung
 - Modellbasiertes Testen
 - Testdatengenerierung
 - Codegenerierung
 - Programmverstehen/-Analysen
 - Simulation
- 1 **exakte** Nachbildung von Rekursion → weniger Fehlalarme¹
 - 2 Problem: riesiger unendlicher Zustandsraum
- a priori Kompression der inneren Struktur (Source-to-Source)

¹False Negatives + Fehlabstraktionen

Kellersystem (PDS)

Kellersystem (Pushdown System)

$\mathcal{P} = (P, \Gamma, \hookrightarrow)$

$P \dots$ Zustände

$\Gamma \dots$ Kelleralphabet

$\hookrightarrow \subseteq (P \times \Gamma) \times (P \times \Gamma^*) \dots$ Transitionen

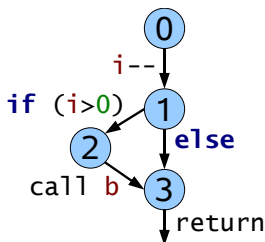
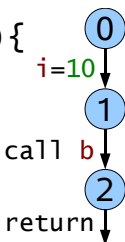
- Kellerautomat ohne Eingabe
- (p, w) Konfiguration, falls $p \in P, w \in \Gamma^*$
- $\rightarrow \subseteq (P \times \Gamma^*) \times (P \times \Gamma^*)$ Erweiterung von \hookrightarrow auf Konfigurationen:

$$(p, aw) \rightarrow (q, bw) :\Leftrightarrow (p, a) \hookrightarrow (q, b)$$

Quellcode → SPDS mit Call-By-Value-Semantik

```
void a(){
  int i=10;
  b(i);
}
```

```
void b(int i){
  i--;
  if (i>0)
    b(i);
}
```



Kontrollfluss ohne Daten...

Kontrollfluss und Daten... $x_i \in \{0, 1\}$ jetzt: **deterministisch**

$(p, a_0) \hookrightarrow (p, a_1)$

$(p, a_0) \hookrightarrow (p, (a_1, 1010))$

$(p, a_1) \hookrightarrow (p, b_0 a_2)$

$(p, (a_1, x_1 x_2 x_3 x_4)) \hookrightarrow (p, (b_0, x_1 x_2 x_3 x_4)(a_2, x_1 x_2 x_3 x_4))$

$(p, a_2) \hookrightarrow (p, \varepsilon)$

$(p, (a_2, x_1 x_2 x_3 x_4)) \hookrightarrow (p, \varepsilon)$

$(p, b_0) \hookrightarrow (p, b_1)$

$(p, (b_0, x_1 x_2 x_3 x_4)) \hookrightarrow (p, (b_1, x_1 x_2 x_3 x_4 - 1))$

$(p, b_1) \hookrightarrow (p, b_2)$

$(p, (b_1, x_1 x_2 x_3 x_4)) \hookrightarrow (p, (b_2, x_1 x_2 x_3 x_4)) \quad x_1 x_2 x_3 x_4 > 0000$

$(p, b_1) \hookrightarrow (p, b_3)$

$(p, (b_1, 0000)) \hookrightarrow (p, (b_3, 0000))$

$(p, b_2) \hookrightarrow (p, b_0 b_3)$

$(p, (b_2, x_1 x_2 x_3 x_4)) \hookrightarrow (p, (b_0, x_1 x_2 x_3 x_4)(b_3, x_1 x_2 x_3 x_4))$

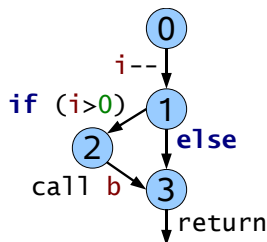
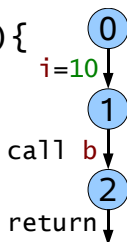
$(p, b_3) \hookrightarrow (p, \varepsilon)$

$(p, (b_3, x_1 x_2 x_3 x_4)) \hookrightarrow (p, \varepsilon)$

Quellcode \rightarrow SPDS mit Call-By-Value-Semantik

```
void a(){
  int i=10;
  b(i);
}
```

```
void b(int i){
  i--;
  if (i>0)
    b(i);
}
```



Kontrollfluss ohne Daten...

Kontrollfluss und Daten... $x_i \in \{0,1\}$ jetzt: deterministisch

$(p, a_0) \hookrightarrow (p, a_1)$

$(p, a_0) \hookrightarrow (p, (a_1, 1010))$

$(p, a_1) \hookrightarrow (p, b_0 a_2)$

$(p, (a_1, x_1 x_2 x_3 x_4)) \hookrightarrow (p, (b_0, x_1 x_2 x_3 x_4)(a_2, x_1 x_2 x_3 x_4))$

$(p, a_2) \hookrightarrow (p, \varepsilon)$

$(p, (a_2, x_1 x_2 x_3 x_4)) \hookrightarrow (p, \varepsilon)$

$(p, b_0) \hookrightarrow (p, b_1)$

$(p, (b_0, x_1 x_2 x_3 x_4)) \hookrightarrow (p, (b_1, x_1 x_2 x_3 x_4 - 1))$

$(p, b_1) \hookrightarrow (p, b_2)$

$(p, (b_1, x_1 x_2 x_3 x_4)) \hookrightarrow (p, (b_2, x_1 x_2 x_3 x_4)) \quad x_1 x_2 x_3 x_4 > 0000$

$(p, b_1) \hookrightarrow (p, b_3)$

$(p, (b_1, 0000)) \hookrightarrow (p, (b_3, 0000))$

$(p, b_2) \hookrightarrow (p, b_0 b_3)$

$(p, (b_2, x_1 x_2 x_3 x_4)) \hookrightarrow (p, (b_0, x_1 x_2 x_3 x_4)(b_3, x_1 x_2 x_3 x_4))$

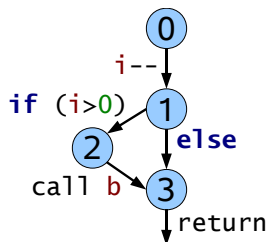
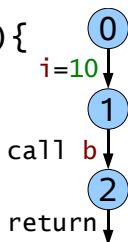
$(p, b_3) \hookrightarrow (p, \varepsilon)$

$(p, (b_3, x_1 x_2 x_3 x_4)) \hookrightarrow (p, \varepsilon)$

Quellcode → SPDS mit Call-By-Value-Semantik

```
void a(){
  int i=10;
  b(i);
}
```

```
void b(int i){
  i--;
  if (i>0)
    b(i);
}
```



Kontrollfluss ohne Daten...

Kontrollfluss und Daten... $x_i \in \{0, 1\}$ jetzt: **deterministisch**

$(p, a_0) \hookrightarrow (p, a_1)$

$(p, a_0) \hookrightarrow (p, (a_1, 1010))$

$(p, a_1) \hookrightarrow (p, b_0 a_2)$

$(p, (a_1, x_1 x_2 x_3 x_4)) \hookrightarrow (p, (b_0, x_1 x_2 x_3 x_4)(a_2, x_1 x_2 x_3 x_4))$

$(p, a_2) \hookrightarrow (p, \varepsilon)$

$(p, (a_2, x_1 x_2 x_3 x_4)) \hookrightarrow (p, \varepsilon)$

$(p, b_0) \hookrightarrow (p, b_1)$

$(p, (b_0, x_1 x_2 x_3 x_4)) \hookrightarrow (p, (b_1, x_1 x_2 x_3 x_4 - 1))$

$(p, b_1) \hookrightarrow (p, b_2)$

$(p, (b_1, x_1 x_2 x_3 x_4)) \hookrightarrow (p, (b_2, x_1 x_2 x_3 x_4)) \quad x_1 x_2 x_3 x_4 > 0000$

$(p, b_1) \hookrightarrow (p, b_3)$

$(p, (b_1, 0000)) \hookrightarrow (p, (b_3, 0000))$

$(p, b_2) \hookrightarrow (p, b_0 b_3)$

$(p, (b_2, x_1 x_2 x_3 x_4)) \hookrightarrow (p, (b_0, x_1 x_2 x_3 x_4)(b_3, x_1 x_2 x_3 x_4))$

$(p, b_3) \hookrightarrow (p, \varepsilon)$

$(p, (b_3, x_1 x_2 x_3 x_4)) \hookrightarrow (p, \varepsilon)$



```

define DEFAULT_INT_BITS 32
int heap[2^32];
int ptr(32);
module void a_main_String_V(int v0(32));
module void a_a_V();
module void a_b_I_V(int v0(32));
init jinit;

module void jinit() {
ptr=1, A i (0,2^32) heap[i]=0;
goto a_main_ALjava_lang_String_V; }

module void a_main_String_V(int v0(32)) {
a_main_ALjava_lang_String_V: a_a_V();
a_main_ALjava_lang_String_V3: return; }

module void a_a_V() {
int v0(32);
int s0(32);
a_a_V: s0=10;
a_a_V2: v0=s0;
a_a_V3: s0=v0;
a_a_V4: a_b_I_V(s0);
a_a_V7: return; }

```

```

module void a_b_I_V(int v0(32))
{
int s0(32);
a_b_I_V: v0=v0-1;
a_b_I_V3: s0=v0;
a_b_I_V4: if
:: s0<=0 -> goto a_b_I_V11;
:: else -> skip;

fi;
a_b_I_V7: s0=v0;
a_b_I_V8: a_b_I_V(s0);
a_b_I_V11: return;
}

AssertError: goto AssertError;
Exception: goto Exception;
HeapOverflow: goto HeapOverflow;
StringBuilderOverflow: goto StringBuil

```



vereinfachter Ausschnitt

```

module void a_a_V() {
  int v0(32);
  int s0(32);
  a_a_V:  s0=10;
  a_a_V2: v0=s0;
  a_a_V3: s0=v0;
  a_a_V4: a_b_I_V(s0);
  a_a_V7: return; }

```

```

module void a_b_I_V(int v0(32)) {
  int s0(32);
  a_b_I_V:  v0=v0-1;
  a_b_I_V3:  s0=v0;
  a_b_I_V4:  if
             :: s0<=0 -> goto a_b_I_V11;
             :: else -> skip;
  fi;
  a_b_I_V7:  s0=v0;
  a_b_I_V8:  a_b_I_V(s0);
  a_b_I_V11: return;}

```

- Kelleralphabetgröße: $2 * 10^{20}$ versch. Kellersymbole



eine nicht exakte Äquivalenzanalyse

Realisierbarkeit

Variablenbelegung f^q **realisierbar** $\Leftrightarrow \exists$ Konfigurationsfolge aus Startkonfiguration p zur Marke q im SPDS (**Run/Lauf**)

Äquivalenz

$a \equiv_q b \Leftrightarrow \forall$ realisierbaren $f^q : f^q(a) = f^q(b)$

- analog Konstanten
- unentscheidbar für Hochsprachen
- konservativ, interprozedural, kontextinsensitiv
- ⇒ schnell + unpräzise (nicht alle Äquivalenzen werden erkannt)



eine nicht exakte Äquivalenzanalyse (Beispiel)

```

module void a_a_V() {
int v0(32);
int s0(32);
a_a_V:  s0=10;      # {s0=10}
a_a_V2: v0=s0;     # {v0=s0=10}
a_a_V3: s0=v0;     # {v0=s0=10}
a_a_V4: a_b_I_V(s0); # {v0=s0=10}
a_a_V7: return; }

```

```

module void a_b_I_V(int v0(32)) {
int s0(32);
a_b_I_V:  v0=v0-1;    # {}
a_b_I_V3:  s0=v0;     # {s0=v0}
a_b_I_V4:  if
           :: s0<=0 -> goto a_b_I_V11;
           :: else -> skip;    # {s0=v0}
fi;
a_b_I_V7:  s0=v0;     # {s0=v0}
a_b_I_V8:  a_b_I_V(s0); # {s0=v0}
a_b_I_V11: return;}

```

Variablenersetzungen mittels Äquivalenzinformationen

```

module void a_a_V() {
  int v0(32);
  int s0(32);
  a_a_V:  s0=10;      # {s0=10}
  a_a_V2: v0=10;      # {v0=s0=10}
  a_a_V3: s0=10;      # {v0=s0=10}
  a_a_V4: a_b_I_V(10); # {v0=s0=10}
  a_a_V7: return; }

```

```

module void a_b_I_V(int v0(32)) {
  int s0(32);
  a_b_I_V:  v0=v0-1;      # {}
  a_b_I_V3:  s0=v0;      # {s0=v0}
  a_b_I_V4:  if
             :: v0<=0 -> goto a_b_I_V11;
             :: else -> skip;      # {s0=v0}
  fi;
  a_b_I_V7:  s0=v0;      # {s0=v0}
  a_b_I_V8:  a_b_I_V(v0); # {s0=v0}
  a_b_I_V11: return;}

```

- keine **lesende** Verwendung von s0 (beide Methoden)
- keine **lesende** Verwendung von v0 in a_a_V

Resultat der Variablenelimination

```

module void a_a_V() {
a_a_V:  skip;
a_a_V2: skip;
a_a_V3: skip;
a_a_V4: a_b_I_V(10);
a_a_V7: return; }

```

```

module void a_b_I_V(int v0(32)) {
a_b_I_V:  v0=v0-1;
a_b_I_V3: skip;
a_b_I_V4:  if
           :: v0<=0 -> goto a_b_I_V11;
           :: else -> skip;
fi;
a_b_I_V7:  skip;
a_b_I_V8:  a_b_I_V(v0);
a_b_I_V11: return;}

```

- Kelleralphabetgröße: von $2 * 10^{20}$ auf $3 * 10^{10}$ Kellersymbole
- Reduktion um **Faktor**: $7,4 * 10^9$
- Anzahl Transitionen: quadratisch (Effekt noch größer)
- genauere Äquivalenzanalyse \Rightarrow mehr Variableneliminationen



exakte Äquivalenzanalysen

exakt

Äquivalenzanalyse heißt **exakt**, wenn alle auftretenden Äquivalenzen durch Analyse erkannt werden.

- keine Über- oder Unterapproximation des Modellverhaltens
- für SPDS berechenbar (bei Hochsprachen nicht)
- exakte Analyseinformationen zum Modell
- kleinere Modelle, kompaktere Testdaten
- genauere Analyseinformationen für Originalprogramm

eine exakte Äquivalenzanalyse - Idee

Kofaktor f_a einer Funktion f ; iteriert: $f_{ab} = (f_a)_b$

$f_{x_i} := f(x_1, \dots, x_{i-1}, 1, x_{i+1}, \dots, x_n)$ 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 .

gegeben: SPDS P

1. Berechnung symbolischer Automat $Post^*(P)$ (Esparza et al.)
→ akzeptiert erreichbare Konfigurationen
2. Extraktion char. Funktionen f^q für Konfigurationsköpfe an q
3. Kofaktortests: Bestimmung der Variablenäquivalenzen aus f^q

Kofaktortest + Beschleunigung exakter Äquivalenzanalysen

Äquivalenztest für $x = (x_1 x_2 \dots x_n), y = (y_1 y_2 \dots y_n) \in \text{Vars}_q$

$$x \equiv_q y \Leftrightarrow \forall i : f_{x_i y_i}^q = \emptyset \wedge f_{x_i' y_i}^q = \emptyset$$

- Kofaktorbildungen in BDDs effizient
 - dennoch: $O(n \cdot m^2)$ Kofaktorbildungen ($m = \text{Anzahl Vars}$)
- Kofaktorbildungen vermeiden:
- x an Marke q konstant gdw. $\forall i : f_{x_i} = \emptyset \vee f_{x_i'} = \emptyset$.
 - Symmetrie x_i, y_i notwendige Bedingung
 - Asymmetrietests (per Signaturen): $(\exists i : \tau(x_i, y_i)) \Rightarrow x \not\equiv_q y$

191 Instanzen-Benchmark@ AMD 64 X2 4200+, 2GB RAM

- durch approximative Äquivalenzanalyse erkannte Äquivalenzen:

Code-Beispiel	$\varnothing d_{v,L}$
ParameterRestrictions.java (7Bits)	96%
ConcreteFieldClass.java (8Bit)	92%
While.java(8Bit)	85%
Durchschnitt (alle 191 Instanzen)	76%

Table: Gleichheit der approximativen und exakten Äquivalenzklassen (großes $d_{v,L}$ besser)

Reduktionsergebnis für Variablenelimination

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)	$8,6 \cdot 10^{89}$	$5,2 \cdot 10^{52}$	$2,0 \cdot 10^{52}$

Table: Vergleich der Modellgröße in der Anzahl der Köpfe vor der Reduktion (hd^-) und nach der Reduktion für approximative (hd^+) und exakte Variante (hd^*)

Modellprüfzeiten nach Variablenelimination

- exakte Analysen so aufwändig wie MC
- hier: unexakte Version

Beispiel	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
Concrete...	0 s	0 s	6 s	0 s	39 s	0 s	111 s	1 s	329 s	1 s
Dispatching	329 s	12 s	MOut	758 s	MOut	MOut	MOut	MOut	MOut	MOut
Exceptions...	5 s	0 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	0 s	1 s	1 s	1 s	4 s	4 s	30 s	11 s	355 s	44 s
LinkedList	2 s	0 s	12 s	1 s	59 s	5 s	234 s	12 s	702 s	34 s
MemoFib	4 s	0 s	MOut	9 s	MOut	113 s	MOut	502 s	MOut	MOut
Parameter...	5 s	1 s	68 s	9 s	340 s	110 s	1538 s	500 s	MOut	MOut
RecFib	7 s	0 s	7 s	0 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	0 s	0 s	1 s	0 s	1 s	1 s	2 s	1 s	3 s	2 s
false_neg...	339 s	15 s	MOut	741 s	MOut	MOut	MOut	MOut	MOut	MOut
total	1417 s	76 s	334 s	32 s	1565 s	273 s	6278 s	1239 s	2034 s	255 s

Modellprüfzeiten für Moped. 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.

Reduktionstechniken im Überblick

Nicht-Parasitäre Techniken (Kontroll-/Datenfluss-Struktur erhaltend)

- Variablenreorganisation
- Kontrollfluss-Slicing
- SMT-Reduktion
- Konstantenpropagation/-Faltung
- weitere triviale Transformationen

Parasitäre Techniken (Kontroll-/Datenfluss-Struktur verändernd)

- Variablenelimination mittels Äquivalenzanalysen
- Richtungs-Entscheidungsreduktion
- Stotterreduktion
- Wertebereichreduktion mittels Intervallanalysen

vergleichbare Ansätze

1. Spin (Bell, Promela): dead-code/var-elimination, constant prop./folding, partial order red., etc.
→ endl. Zustandsraum wegen Tiefensuche (keine Rekursion)
2. Alfred/Moped: zwar PDS aber nur triviale Modellanalysen

Mittagessen!