Fehlalarmfreie Abstraktion in ISO-C konformer Semantik

Dirk Richter, Roberto Hoffmann

Institut für Informatik
Martin-Luther-Universität Halle-Wittenberg
Von-Seckendorff-Platz 1
D-06099 Halle (Saale)
richterd@informatik.uni-halle.de
hoffmaro@informatik.uni-halle.de

Abstract: Viele Aussagen zum Programmverhalten sind in turingmächtigen Programmiersprachen unentscheidbar (Rice Theorem). Mittels Abstraktion vom Programmverhalten können nicht-turingmächtige Modelle automatisch erzeugt werden. Modelle in Form von symbolischen Kellersystemen (SPDS) erlauben eine so präzise Darstellung des Programmverhaltens, sodass eine ISO-C konforme Semantik möglich ist [RB12, KZR09]. Allerdings führen präzisere Darstellungen stets auch zu komplexeren und umfangreicheren Modellen. Dies erschwert Software-Modellprüfung, modellbasiertes Testen und Testdaten- und Codegenerierung. Mehr Abstraktion führt hingegen zu kleineren Modellen, allerdings auch zu *mehr* Fehlalarmen. Ziel dieser Arbeit ist es, bezüglich temporaler Aussagen unwichtige Teile eines SPDS zu identifizieren und von diesen Teilen so zu abstrahieren, dass *keine* Fehlalarme entstehen.

1 Einleitung

Im Rahmen von agilen Methoden (z.B. Extreme Programming) wird das Verhalten eines Softwaresystems durch viele Nebenbedingungen (Unit Tests, temporale Formeln oder JML) spezifiziert. Diese sind regelmäßig auf Einhaltung (Modellprüfung/Testen) zu überprüfen. Viele Modellprüfer beschränken dabei 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. Im Gegensatz zu vergleichbaren Arbeiten zur Abstraktion von Modellen durch finite state Modellprüfer wie BLAST, SPIN, NuSMV/SMV, JavaPathFinder, F-Soft oder Bogor (Bandera Projekt) betrachten wir in dieser Arbeit die Modellabstraktion *unendlicher* symbolischer Modelle in Form von symbolischen Kellersystemen (SPDS). Eine Beschränkung auf eine maximale Anzahl an Methodenaufrufen ist dann nicht nötig und vermeidet eine exponentielle Modellvergrößerung, welche z.B. durch Inlining verursacht wird. SPDS können mittels JMoped [SSE05] aus Java gewonnen und mittels des Modellprüfers Moped [ES01, Sch02, SSE05] überprüft werden und ermöglichen eine interprozedurale und kontextsensitive Weiterverarbeitung. 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.

Mit SPDS kann eine ISO-C konforme Semantik definiert werden [RB12, KZR09], so dass C-Programme (z.B. für eingebettete Systeme) direkt als Modell genutzt werden können. Abstraktionen werden (u.a. wegen nicht ausreichender Vereinfachung) oft *manuell* erzeugt [NK00], was jedoch fehlerträchtig ist. Ziel dieser Arbeit ist die *automatische* Abstraktion für SPDS. Dazu wird die Wichtigkeit von Teilen der Modellbeschreibung mit einer Heuristik bewertet, welche einerseits Programmanalysen auf der Modellbeschreibung und andererseits gegebene temporale Formeln (die Spezifikation) nutzt. So ist es möglich, unwichtige Teile der Modellbeschreibung zu erkennen und davon zu abstrahieren.

2 Grundlagen

2.1 Symbolische Kellersysteme

Die eingesetzten Modelle beschreiben sogenannte Kripkestrukturen. $M = (S, \rightarrow, L_A)$ 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. Im Gegensatz zum Testen kann mittels Software-Modellprüfung die Abwesenheit von Fehlern in Kellersystemen formal nachgewiesen werden [Sch02, ES01, Ber06, EKS02, EHRS00, Wal00, BEM97]. (p, w) heißt *Konfiguration*, falls $p \in P$ und $w \in \Gamma^*$. (p, a) heißt *Kopf* der Konfiguration (p, aw), falls $a \in \Gamma$ und $w \in \Gamma^*$. Die Anzahl der Köpfe nutzen wir, um potentiell unendlich große Zustandsräume miteinander zu vergleichen. Die Menge aller möglichen Konfigurationen sei mit $konf(\mathcal{P}) := \{(p, w) \mid p \in P, w \in \Gamma^*\}$ bezeichnet. Auf Konfigurationen wird die Transitionsrelation $\hookrightarrow$ erweitert zu $\rightarrow \subseteq (P \times \Gamma^*) \times (P \times \Gamma^*)$ mit $(p, aw) \rightarrow (q, bw) :\Leftrightarrow (p, a) \hookrightarrow (q, b)$. Bei einem *Symbolischen Kellersystem* (SPDS) werden die Transitionen nur indirekt (symbolisch) mittels Relationen beschrieben, was die Angabe des Kellersystems vereinfacht [Sch02]. Ein SPDS ist ein Tupel $S = (globs, prc)$ mit einer endlichen Menge globaler Variablen $globs$ und einer endlichen Menge an Prozeduren prc, wobei es eine ausgezeichnete Prozedur $main \in prc$ gibt. Eine Prozedur $q \in prc$ ist ein Tupel $q = (name, pars_q, loc_q, stats_q)$ mit eindeutigem Namen $name$, Parametervariablen $pars_q$, lokalen Variablen loc_q (mit $pars_q \subseteq loc_q$) und einer Liste an Anweisungen $stats_q$ (Rumpf). Jede Anweisung besitzt eine eindeutige Marke $l \in labels$. Die aktuelle Prozedur bzw. Marke einer Konfiguration s wird mit $prc(s)$ bzw. $s)$ bezeichnet. Belegungen lokaler Variablen und Methodenparameter werden in SPDS als Bitvektoren über dem Kelleralphabet zusammen mit der Aufrufhierarchie im Keller repräsentiert. Globale Variablen (z.B. Heap) werden mit Hilfe der Zustände P des Kellersystems beschrieben. Ziel dieser Arbeit ist es, das dazu nötige Kelleralphabet und die benötigten Kellerzustände bereits vorher durch Abstraktion zu verringern.

Seien $Vars_q := globs \cup loc_q$ die Variablen im Gültigkeitsbereich einer Prozedur q und $Expr_{Vars_q}$ arithmetische Ausdrücke (Operatoren alle strikt, Assoziativitäten und Prioritäten wie in ISO-C) über diesen Variablen sowie $env^q : Vars_q \to \mathbb{Z}$ eine Variablenbelegung. Eine Variablenbelegung $env(s) \in ENV$ wird aus dem Kopf einer Konfiguration $s \in konf(S)$ erzeugt (q ist dann durch s bestimmt). Zur Veränderung eines Variablenwertes einer Variablen v an einer Konfiguration s auf den Wert c wird die Schreibweise $s[v \mapsto c]$ verwendet (dies ist ebenfalls eine Konfiguration). Sei weiter $[\![e]\!]_{env^q}$ die Auswertung eines Ausdrucks $e \in Expr_{Vars_q}$ mittels der Variablenbelegung env^q. Dann sind die SPDS-Anweisungen wie folgt definiert:

- **x = e;** mit $x \in Vars_q$ und $e \in Expr_{Vars_q}$ eine Zuweisung des Werts $[\![e]\!]_{env^q}$ an x.
- **p(e$_1$, e$_2$, ..., e$_n$);** mit e$_i \in Expr_{Vars_q}$ ein Aufruf der Prozedur p (Call-By-Value).
- **return e;** mit e $\in Expr_{Vars_q}$ ein Prozedurende mit Rückgabewert $[\![e]\!]_{env^q}$.
- **goto L;** ein unbedingter Sprung an die Marke L.
- **if (e) s;** eine bedingte Anweisung, welche s ausführt, falls $[\![e]\!]_{env^q} = 1$.

Zudem gibt es eine nichtdeterministische Funktion `rand`(e), welche fair einen zufälligen Wert c mit $0 \leq c \leq [\![e]\!]$ liefert[1]. Kommentare gelten bis zum Zeilenende und werden mit dem Symbol # eingeleitet. Weitere ISO-C Anweisungen lassen sich auf diese SPDS-Anweisungen zurückführen [RB12]. SPDS können mit Hilfe der Modellsprache Remopla [KSS06] beschrieben werden. Für Details zur Konstruktion von SPDS-Modellen aus C- und Java-Programmen sei auf [RB12, ES01, Obd02, RZ07] verwiesen.

Ein SPDS B *simuliert* ein SPDS A (in Zeichen $B \preceq A$), falls es eine Funktion $\Theta : konf(A) \to konf(B)$ gibt, so dass jeder Lauf $a1 \to a2 \to$ ... in A zu einem Lauf $\Theta(a1) \to \Theta(a2) \to$ ... in B wird. Zwei SPDS A und B heißen *bisimular* (in Zeichen $A \simeq B$), falls $A \preceq B$ und $B \preceq A$.

Beispiel 1 *(Beispiel eines SPDS)*
Abbildung 1 zeigt im linken Teil ein Beispiel eines SPDS. Die Prozedur `catch` *dient der Verarbeitung von Events. In der Schleife, welche mit* `NO_EVENT` *markiert ist, wartet* `catch`*, bis ein Event (mittels* `pick_event`*) zum Verarbeiten eintritt. Die Konstante* 0 *signalisiert dabei, dass kein Event aufgetreten ist. Die Konstante* 1 *hingegen signalisiert, dass ein Timeout stattgefunden hat, welcher gezählt wird, und ein Neuversuch (*`retry` $=$ `1`*) eingeleitet wird. In allen anderen Fällen handelt es sich um einen gültigen Event, welcher in der Reihung* `history` *als Verlauf (bis zu 100 Einträge) gespeichert wird. Ist ein Event eingetreten, welcher verschieden ist vom Parameter* e*, so wird weiter gewartet, bis der Event* e *eintritt. Erst wenn* e *eingetreten ist, dann wird rekursiv mit dem nächst kleineren Event fortgesetzt (Marke* `hit`*), wenn möglich. Der rechte Teil von Abbildung 1 stellt die zugehörige Abstraktion dar und wird später erläutert. Variablen haben darin einen deutlich kleineren Typ und deren abstrakte Variablenwerte stellen Äquivalenzklassen für viele mögliche konkrete Variablenwerte dar.*

[1] Die ISO-C Funnktion `rand`() kann als SPDS-Funktion `rand(RAND_MAX)` aufgefasst werden.

Abbildung 1: 2 SPDS-Beispiele (links: aus C-Code generiert, rechts: abstrahiert)

```
int offset(8);
int history(8)[100];
int retry(8);
int timeouts(8);

void catch(int e(8)) {
     int word(8);
     int event(8);
     event=0;
     word=1;
     retry=0;  # counts retries
pick:event = pick_event();
    if (event == 0) {   # NO_EVENT
        word=0;
        goto pick;  }
    word=1;
    if (event == 1) {   # timeout
        timeouts=timeouts+1;
        retry=1;    # enter retry mode
        goto pick; }
save:history[offset]=event;
     offset=offset+1;
     if (offset == 100) offset=0;
     if (event != e) goto pick;
hit: if (e > 2) catch(e-1);
     return; }
```

```
int offset(8)
int history(0)[100];
int retry(0);
int timeouts(0);

void catch(int e(4)) {
     int word(0);
     int event(4);
     event=0;
     word=0;
     retry=0;
pick:event = pick_event();
    if (event==0){
        word=0;
        goto pick;  }
    word=0;
    if (event == 1) {
        timeouts=0;
        retry=0;
        goto pick; }
save:history[offset]=0;
     offset=offset+1;
     if (offset == 100) offset=0;
     if (event != e) goto pick;
hit: if (e > 2) catch(e-1);
return; }
```

2.2 Syntax und Semantik temporaler Formeln

Die Logik CTL* besteht aus einer Menge von Zustandsformeln Z, welche Aussagen über einen Zustand eines Modells trifft. Innerhalb einer CTL* Formel selbst sind auch Pfadformeln (Menge P) erlaubt, welche wie folgt definiert sind. Die Quantoren A bzw. E werden verwendet, um Aussagen über *alle Pfade* bzw. *min. einen Pfad* zu treffen. Die Operatoren X, U, F sowie G repräsentieren die Operatoren *Nächster* (Next), *strenges Bis* (strong until), *Irgendwann* (future/eventually) bzw. *Immer* (always). Dabei sind E,F,G bzw. $\vee$ abkürzende Schreibweisen für $E\phi = \neg A \neg \phi, F\phi = true\ U\ \phi, G\phi = \neg F \neg \phi, \phi \vee \psi = \neg(\neg\phi \wedge \neg\psi)$, wobei $\phi, \psi \in P$ sind. Sei $AExpr = Expr \cup labels \cup prc$ die Menge an möglichen Ausdrücken, Marken- und Prozedur-Namen. Induktiv sind dann CTL* Formeln wie folgt definiert:

- $AExpr \subset Z$
- $\forall \phi, \psi \in Z : \neg\phi \in Z,\ \phi \wedge \psi \in Z$
- $Z \subseteq P$
- $\forall \phi, \psi \in P : A\phi \in Z, \neg\phi \in P,\ \phi \wedge \psi \in P, X\phi \in P,\ \phi U \psi \in P$

Gilt eine Formel ϕ an einem Zustand s einer Kripkestruktur $M = (S, \rightarrow, L)$ bzw. an einer Konfiguration s eines SPDS $M = (R, \Gamma, \hookrightarrow)$, so schreiben wir dafür $s \models_M \phi$ (bzw. $s \models \phi$, wenn M aus dem Kontext klar ist). Sei $p = s_0 \rightarrow s_1 \rightarrow \ldots$ ein unendlicher Lauf, wobei $fst(p) = s_0$ der Anfangszustand bzw. die Anfangskonfiguration des Laufs p ist und $p_i = s_i \rightarrow s_{i+1} \rightarrow \ldots$ der Suffix des Laufs beginnend bei s_i. Terminiert der Lauf p an einem Zustand bzw. einer Konfiguration s_n, so ist er endlich. Bei endlichen Läufen wird der letzte Zustand bzw. die letzte Konfiguration immer wieder wiederholt, so dass $p = \ldots \rightarrow s_{n-1} \rightarrow s_n \rightarrow s_n \rightarrow s_n \ldots$. Dann sei $\models$ neben Zuständen bzw. Konfigurationen s auch für Läufe p wie folgt erklärt:

$$
\begin{aligned}
s \models a &\quad :\Leftrightarrow \quad [\![a]\!]_{env^s} = 1 \vee a = s) \vee a = prc(s) \\
s \models \neg\phi &\quad :\Leftrightarrow \quad s \not\models \phi \\
s \models \phi \wedge \psi &\quad :\Leftrightarrow \quad s \models \phi \ \wedge \ s \models \psi \\
s \models A\phi &\quad :\Leftrightarrow \quad (\forall p = s \rightarrow s_1 \rightarrow \ldots : (p \models \phi)) \\
p \models \phi, \ mit\ \phi \in Z &\quad :\Leftrightarrow \quad fst(p) \models \phi \\
p \models \neg\phi, \ mit\ \phi \in Z &\quad :\Leftrightarrow \quad p \not\models \phi \\
p \models \phi \wedge \psi &\quad :\Leftrightarrow \quad p \models \phi \ \wedge \ p \models \psi \\
p \models \phi U \psi &\quad :\Leftrightarrow \quad \exists i \geq 0 : ((p_i \models \psi) \wedge (\forall j < i : p_j \models \phi)) \\
p \models X\phi &\quad :\Leftrightarrow \quad p_1 \models \phi
\end{aligned}
$$

Im Folgenden wird auch $K \models \phi$ für eine Menge an Konfigurationen K verwendet, falls $\forall s \in K : s \models \phi$.

Zum Beispiel 1 seien die temporalen Formeln aus Abbildung 2 gegeben. Sie beschreiben Eigenschaften des SPDS, die zu überprüfen sind. Hierbei besagt z.B. die Formel $G(0 \leq \texttt{offset} \leq 100)$, dass sich Variable `offset` stets größer oder gleich ist als 0 aber auch kleiner oder gleich 100.

Abbildung 2: Beispielhafte temporale Formeln zu Beispiel 1

Beschreibung	**temporale Formel**
Programm terminiert	$G(F\ \texttt{end})$
array out of bounds	$G(0 \leq \texttt{offset} \leq 100)$
nur gültige events	$G(0 \leq \texttt{event} < 16)$
nur gültige events	$G(2 \leq \texttt{e} < 16)$
jeder event wird geloggt	$G((\texttt{event} > 1) \Rightarrow !\texttt{pick}\ U\ \texttt{save})$
richtiger event wird behandelt	$G(\texttt{event} == \texttt{e} \Rightarrow !\texttt{pick}\ U\ \texttt{hit})$
Wiederholung bei timeout/nix	$G(\texttt{event} < 2 \Rightarrow !\texttt{save}\ U\ \texttt{pick})$
nur gültige events geloggt	$G(\texttt{save} \Rightarrow \texttt{event} > 1)$

2.3 SAT-Solving und SAT-Heuristiken

Der weit verbreitete Modelprüfer NuSMV bietet die Möglichkeit, temporale Formeln auf einem Modell nicht nur per BDD checking, sondern auch via SAT-Solving als Bounded Model Checking zu prüfen [CCG+02]. Dabei werden das Modell und die temporalen Formeln als SAT-Problem formuliert und einem SAT-Solver zur Entscheidung der Gültigkeit der Formeln auf dem gewählten Modell übergeben. In SAT-Solvern sind Heuristiken verfügbar, welche den eigentlich exponentiellen Aufwand für das Erfüllbarkeitsproblem in vielen praktischen Fällen deutlich zu reduzieren vermögen. Dies geschieht durch geeignete Variablenwahl und -belegung beim dabei angewandten Backtracking-Verfahren (Davis-Putnam-Logemann-Loveland-Algorithmus) [Hof05, Til05].

Ausgehend von dieser rein logisch-booleschen Formulierung des Modells und der temporalen Formeln in Konjunktiver Normalform (CNF) kann man weitere wichtige Informationen extrahieren: Man benutzt die Heuristiken des SAT-Solvers dazu, relevante Modellteile bezüglich gegebener temporaler Formeln zu bestimmen [MA03], analog zu Überdeckungsmaßen [HM09].

3 Abstraktionsprozess

Durch Abstraktion kann der Zustandsraum verkleinert werden. Beim Abstrahieren von einem Konfigurationenübergang (Eliminieren überflüssiger SPDS-Anweisungen) reduziert sich die Anzahl der Köpfe um $\frac{1}{n}$, wobei n die Anzahl der SPDS-Anweisungen darstellt. Andererseits führt das Abstrahieren von einer globalen 32-Bit Variable zu einer Reduktion um den Faktor 2^{32}. Wie Beispiel 1 zeigt, wird der SPDS-Konfigurationenraum bei Generierung aus höheren Programmiersprachen oft durch große Variablentypen bestimmt. Daher gelingt für solche SPDS eine Konfigurationenraumreduktion am stärksten durch Abstraktion von großen Variablentypen zu kleinen. Daher konzentrieren wir uns in dieser Arbeit auf die Abstraktion von Variablen.

3.1 Bestimmung Wichtigkeit γ von Variablenbits

Man kann mithilfe der Heuristiken eines SAT-Solvers für ein gegebenes Modell und dessen temporale Formeln unter Ausnutzung der logischen Zusammenhänge die "Wichtigkeit" einzelner Variablenbits ermitteln. Dazu seien $Vars^*$ die Variablenbits der Variablen $Vars$ des SPDS S. Dann werden als erstes das Modell und die temporalen Formeln ins NuSMV-Format überführt. Dabei wird via Def-Use-Analysen [Muc97] (als Programmanalysen bei höheren Programmiersprachen gut bekannt) der Datenfluss nachgebildet, um diesen unabhängig vom Kontrollfluss zu analysieren. In Abbildung 3 wurde Beispiel 1 als NuSMV-Modell umgesetzt, wobei wir aus Effizienzgründen die konkrete NuSMV-Datei symbolisch mittels Parameter beschreiben (Parameter p). Die Variablen wurden mit einem Präfix (`DR_`) versehen, damit eine Kollision mit reservierten Schlüsselworten vermieden

Abbildung 3: NuSMV-Sourcecode des Beispiels

```
PARAM p: 0..100;
VAR DR_retry: 0..255; DR_offset: 0..255; DR_history: array 0..100
  of 0..255; DR_event: 0..255; DR_timeouts: 0..255;
  DR_word: 0..255; DR_e: 0..255;
ASSIGN init(DR_e):= 15; init(DR_retry):= 0; init(DR_word):= 1;
  init(DR_offset):= 0;init(DR_history[p]):=0;init(DR_timeouts):=0;
  init(DR_i):=15;init(DR_event):={0..15};

next(DR_event)       := { 0 .. 15 };
next(DR_retry)       := case DR_event = 1: 1; TRUE: DR_retry; esac;
next(DR_word)        := case DR_event = 0: 0; TRUE: 1; esac;
next(DR_timeouts)    := case DR_event=1: DR_timeouts+1 mod 255;
                             TRUE: DR_timeouts; esac;
next(DR_history[p])  := case DR_offset = p & DR_event > 1: DR_event;
                             TRUE: DR_history[p]; esac;
next(DR_e)           := case DR_e > 2 & DR_event = DR_e: DR_e - 1;
                             DR_e <= 2 & DR_i > 2:DR_i; TRUE:DR_e; esac;
next(DR_offset)      := case DR_event>1 & DR_offset<100: DR_offset+1;
                        DR_event>1&DR_offset=100:0; TRUE:DR_offset; esac;
```

wird. Der in [HM09] vorgestellte modifizierte NuSMV wird dann dazu benutzt, einerseits das Modell und andererseits die temporalen Formeln in konjunktive Normalformrepräsentation zu überführen (CNF_S und CNF_ϕ), welche der charakteristischen Funktion der dem Modell bzw. der den temporalen Formeln entsprechenden Kripkestruktur entspricht. Aufgrund der Endlichkeit dieser Darstellung sind nur k Zeitschritte (frei wählbar) der auftretenden Systemübergänge enthalten. Wir haben für unser Beispiel $k = 5$ gewählt. Ausgehend von diesen Repräsentationen wird die "Wichtigkeit" der einzelnen Bits anhand deren logischer Komplexität und Vernetzung bestimmt. Dies geschieht analog wie beim SAT-Solving. Für das positive und das negative Literal einer Variablen berechnet die Heuristik anhand der in Form einer CNF-Datei vorliegenden Problemstruktur einen Zahlenwert als Maß zur Wichtigkeit $\gamma_S(v.i) \in \mathbb{R}$ (Modell CNF_S) bzw. $\gamma_\phi(v.i) \in \mathbb{R}$ (Formel CNF_ϕ) für alle Variablenbits $v.i \in Vars^*(S)$. Normalerweise wird dann im Laufe des SAT-Solving diejenige Variable zuerst belegt, welche über den höchsten Wert verfügt. An dieser Stelle bricht der modifizierte SAT-Solver ab und gibt stattdessen die ermittelten Werte für jedes Literal aus.

Die CNF-Darstellung des Modells beziehungsweise der temporalen Formeln enthält alle Bits aller Variablen für alle Zeitschritte $k = 0..5$. Hieraus wird schließlich das Maß zur Wichtigkeit $\gamma(v.i) \in \mathbb{R}$ für alle Variablenbits $v.i \in Vars^*$ bestimmt. Dabei werden Variablen entlang der Zeitachse sowie positive und negative Literale zusammengefasst. Aufgrund bisheriger Erfahrungen verwenden wir dafür statt Summenbildung das Maximum, um selten vorkommende, aber dafür wichtige Variablen angemessen zu berücksichtigen. Die Wichtigkeiten aus Modell und Formeln werden normiert (Skalierung auf 50%) und addiert. So sind Variablenbits, welche in beiden Teilen von Bedeutung sind, insgesamt umso wichtiger. Für Beispiel 1 ergeben sich die Wichtigkeiten aus Abbildung 4. Von den Variablen sind `DR_event0..3, DR_e0..3` und `DR_offset` am wichtigsten

($\gamma > 0$). Eher unwichtig sind hingegen `DR_word*`, `DR_event4..7`, `DR_e4..7`, `DR_history*`, `DR_retry*` und `DR_timeouts*`, weil sie unbedeutend im Modell und unnötig für die temporalen Formeln sind ($\gamma \approx 0$). Die ermittelte Wichtigkeit wird dann genutzt, um den Abstraktionsprozess zu leiten. Wichtige Teile werden beibehalten, unwichtige abstrahiert. Die Grenze kann dabei beliebig variiert werden, um sie dem gewünschten Abstraktionsgrad anzupassen.

Abbildung 4: Ermittelte Wichtigkeit der Variablenbits (Auszug), k=5

Variablenbit	γ_S	γ_ϕ	$\gamma \in [0,1]$
`DR_event.0`	1687.29	18.00	0.54
`DR_event.1`	1750.69	16.80	0.54
`DR_event.2`	1748.82	11.80	0.48
`DR_event.3`	1735.15	4.80	0.40
`DR_event.4`	0.00	0.00	0.00
...			
`DR_offset.0`	2434.98	43.85	0.98
`DR_offset.1`	2529.55	20.32	0.73
`DR_offset.2`	2161.85	18.98	0.65
`DR_offset.3`	1760.80	5.00	0.41
`DR_offset.4`	1558.04	5.00	0.37
`DR_offset.5`	1547.79	5.00	0.37
`DR_offset.6`	2201.81	5.00	0.49
`DR_offset.7`	2011.01	5.00	0.46
`DR_word.0`	113.62	0.00	0.03
`DR_word.1`	0.00	0.00	0.00
...			
`DR_history[0].0`	691.09	8.74	0.24
`DR_history[0].1`	20.44	8.00	0.10
`DR_history[0].2`	18.67	6.00	0.07
...			

3.2 α-Abstraktion von unwichtigen Variablenbits

Gegeben sei neben dem Quellmodell S der Abstraktionsgrad $\alpha \in [0,1]$. Dieser wird später automatisch bestimmt und quantifiziert, wie viel der Wichtigkeit in das Zielmodell $T_\alpha(S)$ einfließen soll. T_α sei dabei die Abstraktionsfunktion. Man wähle dann sukzessive die wichtigsten Variablenbits[2] $v.i \in W$ und konstruiere daraus minimal nötige Typen

[2]Z.B. indem die Variablenbits nach der Wichtigkeit γ sortiert werden.

$bits^{\alpha}(v) := \max\{i \mid v.i \in W\}$ für die Variablen $v \in Vars$ des Zielmodell, so dass

$$|\alpha - \sum_{v \in Vars} bits^{\alpha}(v) \,/\, \sum_{v \in Vars} bits(v)| \tag{1}$$

minimal ist. Alle Variablenbits $v.i \notin W$ mit $i > bits^{\alpha}(v)$ sind unwichtig und werden vom Quellmodell abstrahiert. Seien im folgenden abkürzend die Konstanten $r(v) := 2^{bits(v)}$ und $r_{\alpha}(v) := 2^{bits^{\alpha}(v)}$ verwendet. Ist $r_{\alpha}(v) = 1$ und damit $bits^{\alpha}(v) = 0$, so ist v unwichtig im Modell. v braucht dann nicht in einer Typdeklaration auftreten. Allerdings kann es dann noch lesende und schreibende Variablenverwendungen geben. Zum Zwecke der Veranschaulichung sei in dieser Arbeit eine solche Typdeklaration mit dem Typ 0 Bits erlaubt. Derartig definierte SPDS-Variablen mit dem Typ 0 Bits haben keinen Einfluss auf den Konfigurationenraum. Das abstrahierte Zielmodell $T_{\alpha}(S)$ besteht aus den verkleinerten Typen $bits^{\alpha}$ und bildet über die Variablenwerte aus S Äquivalenzklassen für $T_{\alpha}(S)$. Der konkrete Variablenwert $[\![v]\!]$ in S wird in $T_{\alpha}(S)$ abstrakt durch die Äquivalenzklasse $[\![v]\!] \% r_{\alpha}(v)$ repräsentiert. Ist z.B. $bits(v) = 5$ und $bits^{\alpha}(v) = 2$, so wird v von den höheren 3 Bits abstrahiert, so dass die abstrakten Werte 0 bis 3 jeweils die konkreten Werte aus Abbildung 5 darstellen.

Abbildung 5: Beispiel einer 3-Bit-Abstraktion von $bits(v) = 5$ auf $bits^{\alpha}(v) = 2$.

abstrakt	**konkrete mögliche Werte**
0	$0, 4, 8, 12, 16, 20, 24, 28$
1	$1, 5, 9, 13, 17, 21, 25, 29$
2	$2, 6, 10, 14, 18, 22, 26, 30$
3	$3, 7, 11, 15, 19, 23, 27, 31$

Schreibende Verwendungen einer SPDS-Variable v (also eine SPDS-Zuweisung) müssen von den konkreten Werten aus S in abstrakte Werte in $T_{\alpha}(S)$ konvertiert werden. Dazu werden SPDS-Zuweisungen der Form $v = e$ durch $v = e \% r_{\alpha}(v)$ aus gedrückt. Dabei fallen die Werte entsprechend ihrer Äquivalenzklasse zusammen. Lesende Verwendungen von v in S müssen für $T_{\alpha}(S)$ analog zu konkreten Werten der Äquivalenzklasse konvertiert werden. Dies wird erreicht, indem lesende Verwendungen von v in S für $T_{\alpha}(S)$ mittels $v + \texttt{rand}(\frac{r(v)}{r_{\alpha}(v)} - 1) * r_{\alpha}(v)$ ausgedrückt werden. Dadurch werden aus abstrakten Variablenwerten sämtliche mögliche konkrete Werte. Der Abstraktionsprozess wird durch eine Intervallanalyse [Muc97] ergänzt. Diese bestimmt statisch, dass manche Variablenwerte nie realisiert werden können. Dies wird genutzt, um nicht alle konkreten Belegungen eines abstrakten Wertes zu erzeugen, sondern nur diejenigen, welche nötig sind. Weiter werden Ausdrücke durch eine Konstantenfaltung ausgewertet und vereinfacht. Dadurch vereinfachen sich abstrakte Werte im Idealfall so stark, dass die Funktion `rand()` keiner Verwendung bedarf. Der Konfigurationenraum und die zu Grunde liegende Kripkestruktur verkleinern sich bei der Abstraktion erheblich.

Abbildung 6: Abstraktion temporaler Formeln

Für eine Zustandsformel p ist $\alpha_\tau^+(p) = \alpha^+(p)$ und $\alpha_\tau^-(p) = \alpha^-(p)$.
Für eine Formel $\phi \in \{\neg p, p \wedge q, Xp, pUq, Ap\}$ ist:

$$\begin{array}{lcllcl}
\alpha_\tau^-(\neg p) & = & \neg\alpha_\tau^+(p) & \alpha_\tau^+(\neg p) & = & \neg\alpha_\tau^-(p) \\
\alpha_\tau^-(p \wedge q) & = & \alpha_\tau^-(p) \wedge \alpha_\tau^-(q) & \alpha_\tau^+(p \wedge q) & = & \alpha_\tau^+(p) \wedge \alpha_\tau^+(q) \\
\alpha_\tau^-(Xp) & = & X\alpha_\tau^-(p) & \alpha_\tau^+(Xp) & = & X\alpha_\tau^+(p) \\
\alpha_\tau^-(pUq) & = & \alpha_\tau^-(p)U\alpha_\tau^-(q) & \alpha_\tau^+(pUq) & = & \alpha_\tau^+(p)U\alpha_\tau^+(q) \\
\alpha_\tau^-(Ap) & = & A\alpha_\tau^-(p) & \alpha_\tau^+(Ap) & = & A\alpha_\tau^+(p)
\end{array}$$

3.3 Temporale Abstraktion

Bei Abstraktionsgraden $\alpha < 1$ können ohne Abstraktion der temporalen Formel ϕ Fehlalarme entstehen, welche nicht erwünscht sind (false positives). In diesen Fällen erfüllt das abstrahierte Modell $T_\alpha(S)$ die Formel ϕ ($T_\alpha(S) \models \phi$), wo hingegen das ursprüngliche Modell S diese nicht erfüllt ($S \not\models \phi$). Daher wird für kleinere Abstraktionsgrade ($\alpha < 1$) ϕ zu ϕ^α abstrahiert, so dass aus $T_\alpha(S) \models \phi^\alpha$ stets auch $S \models \phi$ folgt. Dann kann es lediglich unechte Negativbeispiele (false negatives) geben, welche am Ausgangsmodell S auf Echtheit überprüft und für eine bessere Abstraktion genutzt werden können (CEGAR).

Der Abstraktionsgrad α induziert eine Abstraktionsrelation $R^\alpha \subseteq \mathit{konf}(S) \times \mathit{konf}(T_\alpha(S))$ zwischen den Konfigurationen von S und $T_\alpha(S)$. Nach Konstruktion ist $\mathit{konf}(T_\alpha(S)) \subseteq \mathit{konf}(S)$. Unwichtige Konfigurationen wurden vom Konfigurationenraum abstrahiert. Die Variablenbelegungen $[\![v]\!]$ von Konfigurationen $\mathit{konf}(S) \setminus \mathit{konf}(T_\alpha(S))$ werden mittels der Abstraktionsrelation R^α abgebildet auf deren Äquivalenzklassen $[\![v]\!]\%r_\alpha(v)$. Sei $p \in \mathit{AExpr}$ eine Bedingung (Prädikat) innerhalb einer temporalen Formel. Da Variablentypen verkleinert wurden, könnte es eine Konfiguration in S, aber nicht in $T_\alpha(S)$ mit Variablenbelegung $\mathit{env} \in \mathit{ENV}$ geben, so dass p erfüllt ist in S aber nicht in $T_\alpha(S)$. Wir wollen daher die sichere Abstraktion $p^\alpha \in \mathit{AExpr}$ so definieren, dass stets gilt $p^\alpha \Rightarrow p$. Damit kann der Operator α^- definiert werden als $\alpha^-(p) := \forall T_\alpha(p)$, wobei $\forall q$ bedeutet, dass q für jeden Funktionswert von `rand` erfüllt sein muss. $\alpha^-(p)$ ist damit für eine abstrakte Konfiguration s^α erfüllt, wenn p für alle zugehörigen konkreten Konfigurationen s erfüllt ist. Analog definiert man den Operator α^+ als $\alpha^+(p) := \exists T_\alpha(p)$. $\alpha^+(p)$ ist dann für eine abstrakte Konfiguration s^α erfüllt, wenn p für mindestens eine zugehörige konkrete Konfiguration s erfüllt ist. Ist $\alpha^+(p) = \alpha^-(p)$, so heißt α *präzise* bezüglich p. α^+ und α^- werden auf temporale Formeln erweitert zu α_τ^+ und α_τ^-. Diese sind induktiv gemäß Abbildung 6 definiert und werden als universelle bzw. existenzielle temporale Abstraktion bezeichnet. Als temporale Abstraktion für ϕ dient dann $\phi^\alpha := \alpha_\tau^-(\phi)$. Man beachte, dass äquivalente temporale Formeln verschiedene Abstraktionen besitzen können.

3.4 Eigenschaften

Bei einem Abstraktionsgrad $\alpha \approx 1$ werden nur sehr wenige unwichtige Variablen abstrahiert. In diesem Fall ist die Abstraktion typischerweise noch präzise. Mit sinkendem Abstraktionsgrad $\alpha \approx 0.01$ jedoch wird zunehmend auch mehr von den wichtigen Variablen abstrahiert, so dass dann die Abstraktion nicht mehr präzise ist. Wegen Abstraktion der gegebenen temporalen Formeln kommt es nicht zu Fehlalarmen (False Negatives, False Positives), wenn sich das Ergebnis der Modellprüfung für ϕ verändern würde.

Satz 1 *Für eine temporale Formel ϕ und ein SPDS S gilt:* $(T_\alpha(S) \models \phi^\alpha) \Rightarrow (S \models \phi)$.

Beweis (Skizze): Ergibt sich nach Konstruktion von T_α und ϕ^α analog [KP00].

□

Die Abstraktion T_α heißt *präzise* bezüglich S und α, falls auch die Umkehrung gilt. Dann ist $(T_\alpha(S) \models \phi^\alpha) \Leftrightarrow (S \models \phi)$. Der minimale Abstraktionsgrad, welcher noch präzise abstrahiert, heißt *optimal* und wird später näherungsweise bestimmt. Für präzise und optimale Abstraktionsgrade sind Fehlalarme ausgeschlossen.

Satz 2 *Es gilt $S \simeq T_\alpha(S)$, falls die in $T_\alpha(S)$ für lesende Variablenverwendungen aus den abstrakten Werten v generierten konkreten Werte „$v + \texttt{rand}(\frac{r(v)}{r_\alpha(v)} - 1) * r_\alpha(v)$“ sich mittels Programmanalysen/-Transformationen zu v vereinfachen lassen.*

Beweis (Skizze):
zu zeigen: $T_\alpha(S) \preceq S$
Hierzu konstruiere man eine entsprechende Funktion $\Theta : konf(S) \rightarrow konf(T_\alpha(S))$ gemäß obiger Abstraktion als $\Theta(s) := s[v \mapsto [\![v]\!]_{env(s)} \% r_\alpha(v)]$. Konkrete Werte in s werden dabei abgebildet auf ihre abstrakten Äquivalenzklassen. Seien nun lesende Verwendungen abstrakter Werte v in S betrachtet. Diese werden zu konkreten Werten „$v + \texttt{rand}(\frac{r(v)}{r_\alpha(v)} - 1) * r_\alpha(v)$“ in $T_\alpha(S)$ konvertiert. Durch die `rand`-Funktion entstehen potentiell neue Kontrollflüsse. Wird durch eine Programmanalysen (z.B. Intervallanalyse) in S die (konservative) Nebenbedingung $v \leq c$ für ein $c < r_\alpha(v)$ erkannt, so kann „$v + \texttt{rand}(\frac{r(v)}{r_\alpha(v)} - 1) * r_\alpha(v)$“ zu v vereinfacht werden, da größere Werte als c in S nicht realisierbar sind. Jede Äquivalenzklasse (abstrakter Wert) repräsentiert dann genau einen konkreten Wert. Ein Lauf $s1 \rightarrow s2...$ in S wird daher zu einem gültigen Lauf $\Theta(s1) \rightarrow \Theta(s2)...$ in $T_\alpha(S)$.
zu zeigen: $S \preceq T_\alpha(S)$
Der Kontrollfluss und damit auch der Lauf hängt nicht von toten Werten ab. Daher kann analog $\Theta(s) := s'$ gewählt werden, wobei s' aus s hervorgeht, indem Belegungen toter Variablen in S entsprechend passend gewählt werden. Nach obigen Betrachtungen ist dies stets möglich, da jede Äquivalenzklasse (abstrakter Wert) genau einen konkreten Wert repräsentiert.

Damit gilt $S \simeq T_\alpha(S)$.

□

4 Fehlalarmfreie Abstraktion

Der Abstraktionsprozess approximiert nun für gegebenes SPDS S, temporale Formel ϕ und maximalem Abstraktionsfehler $eps > 0$ den kleinsten Abstraktionsgrad, welcher noch präzise ist:

1 Berechne Wichtigkeit von Variablenbits und Intervallanalyse im Modell S.

2 Setze $\alpha_0 := 0$ und $\alpha_1 := 1$.

3 Setze $\alpha := \frac{\alpha_0+\alpha_1}{2}$.

4 Berechne α-Abstraktion $T_\alpha(S)$ von Modell S.

5 Setzte $\alpha_1 = \alpha$, falls $\phi^\alpha = \phi$ und $S \simeq T_\alpha(S)$. Andernfalls setzte $\alpha_0 = \alpha$.

6 Gehe zu 3, falls $\alpha_1 - \alpha_0 > eps$.

Schritt 5 wird dabei mittels der hinreichenden Bedingung aus Satz 2 konservativ bestimmt.

Wie man leicht sieht, stellt der Abstraktionsgrad α_1 eine Approximation des kleinsten präzisen Abstraktionsgrads dar (Intervallschachtelung):

Satz 3 *Sei α^* der optimale Abstraktionsgrad und α_1 wie oben bestimmt. Dann ist α_1 ein präziser Abstraktionsgrad für S und es gilt: $|\alpha_1 - \alpha^*| \leq eps$.*

Zur Veranschaulichung sei wieder Beispiel 1 aus Abbildung 1 betrachtet. Es ergibt sich eine Berechnungsfolge von $[0.5, 0.25, ..]$ für α_1, welche bei $\alpha_1 = 0.02$ terminiert. Dies ist eine Reduktion auf 2% der bisher verwendeten Variablenbits. Dabei ergeben sich die neuen minimal nötigen Typen aus Abbildung 7. Der Konfigurationenraum, gemessen in der Anzahl der möglichen Köpfe, wird dabei um den Faktor $2^{832} = 2.86 \cdot 10^{250}$ verkleinert. In Abbildung 1 rechts ist dazu das abstrahierte Modell zu sehen. Darin wurden

Abbildung 7: Berechnete Minimaltypen für Beispiel 1 bei Abstraktionsgrad $\alpha_1 = 0.02$.

	offset	history[*]	retry	timeouts	e	word	event	Gesamt
$bits$	8	800	8	8	8	8	8	848
$bits^{\alpha_1}$	8	0	0	0	4	0	4	16

Ausdrücke zu Konstanten wie z.B. $(0\%2\hat{}4) \equiv 0$ bzw. $(1\%2\hat{}0) \equiv 0$ sowie $(\texttt{timeouts}+1)\%1 \equiv 0$, welche in den Zuweisungen $\texttt{event} = 0$ bzw. $\texttt{word} = 1$ bzw. $\texttt{timeouts} = \texttt{timeouts} + 1$ usw verwendet wurden. Durch Konstantenfaltung werden die Ausdrücke $(\texttt{event} + \texttt{rand}(15) * 16) == 0$ bzw. $(\texttt{e} + \texttt{rand}(15) * 16) == 0$ im Beispiel 1 zu $\texttt{event} == 0$ bzw. $\texttt{e} == 0$ vereinfacht, da statisch festgestellt wird, dass stets $\texttt{event} < 16$ bzw. $\texttt{e} < 16$ im Modell gilt. Das entstandene SPDS-Modell in Abbildung 1 (rechts) enthält dann noch diverse Artefakte wie Zuweisungen an Variablen des Typs 0 Bits. Diese können später eliminiert werden mittels weiterer statischer Analysen (z.B. Slicing [RZ07]).

5 Verwandte Arbeiten

Das in dieser Arbeit vorgestellte Verfahren ist ähnlich zu [DHJ+01]. Auch dort werden Modell und temporale Eigenschaften derart abstrahiert, dass bei Modellprüfung der Abstraktionen Rückschlüsse auf das Ursprungsmodell möglich sind. Allerdings werden dort nur endliche Modelle sowie LTL betrachtet und es wird die Abstraktion manuell vom Nutzer durch die Bandera Abstraction Specification Language (BASL) bestimmt. Methoden dieser Arbeit sind auch für unendliche Strukturen (SPDS) nutzbar und abstrahieren vollautomatisch mittels eines frei wählbaren Abstraktionsgrads α. Anders als die Predikatabstraktion in [GS97] operieren wir direkt in der symbolischen Beschreibung und nicht auf dem zu Grunde liegendem ggf. sehr großen oder unendlichem Transitionssystem. In [MA03], [CCK+02] und [HJMS02] werden Beweise bzw. Gegenbeispiele für das Erfülltsein bzw. Unerfülltsein von SAT-Formeln verwendet, um relevante Modellteile zu identifizieren und zu abstrahieren. Wir hingegen lösen das aufgestellte SAT-Problem nicht (weil aufwändig) und untersuchen es lediglich mittels effizienter Heuristiken auf wichtige Modellteile. In [NK00] werden als Prädikatabstraktion wichtige Prädikate bezüglich der temporalen Spezifikation berechnet. Im Gegensatz zu unserem Verfahren terminiert deren Verfahren nicht immer, es wird nur Bisimularität betrachtet (daher Restmodell sehr groß und kein Einfluss auf Abstaktionsgrad), die Betrachtungen sind im wesentlichen auch mit Slicing realisierbar und es werden nur endliche Modelle (vom unendlichem Programm) betrachtet, weshalb viele unnötige Fehlalarme entstehen können. Letzteres ist wegen der ISO-C konformen Semantik in SPDS [KZR09] bei unserem Verfahren reduziert. Die in [SUM96] vorgestellte Deduktive Modellprüfung generiert wie unsere Methode Abstraktionen basierend auf gegebenen LTL Formeln. Diese ist allerdings auf LTL und endliche Modelle beschränkt und erfordert signifikanten manuellen Eingriff im Gegensatz zu unserer Methode. In [KP00] werden Techniken vorgestellt zur Abstraktion von Kontrollfluss und Daten. Auch diese ist beschränkt auf LTL, endliche Modelle und operieren auf Diskreten Kripkestrukturen statt in der symbolischen Beschreibung eines SPDS. Zudem muss im Gegensatz zu unserem Verfahren für die Abstraktion eine Zuordnung (mapping) von abstrakten auf konkrete Zustände manuell erfolgen.

6 Zusammenfassung und Ausblick

Es wurde ein Verfahren vorgestellt, das zu gegebenen temporalen Formeln ϕ_i und einem Modell S selektiv unwichtige Variablenbits identifiziert und davon abstrahiert. Dabei wurde die Wichtigkeit von Variablenbits im Modell heuristisch bestimmt, und schließlich das Modell von unwichtigen Variablenbits so abstrahiert, dass keine Fehlalarme entstehen (präzise Abstraktion). In früheren Betrachtungen wurde der Grad der Verkleinerung über einen Parameter (Abstraktionsgrad α) gesteuert [RH11]. Bei der Überführung des Quellmodells in ein weniger detailliertes Zielmodell können Fehlalarme (false negatives) entstehen, wenn der Abstraktionsgrad α zu klein ist. Der Abstraktionsgrad α wird in dieser Arbeit automatisch so bestimmt, dass die Abstraktion präzise ist und *keine Fehlalarme* entstehen.

Zwar ist es möglich auch eine optimale Abstraktion zu berechnen, jedoch übersteigt deren Komplexität die der Modellprüfung [RH11], was den vorgestellten heuristischen Ansatz rechtfertigt.

Positive und negative Literale einer Variablen x_i in der Repräsentation des Modells wurden in der vorgestellten Methode zusammengefasst. Eine feinkörnige Betrachtung der einzelnen Literale einer Variablen, z.B. wenn nur das positive als sehr wichtig identifiziert wurde, kann sicherlich weitere Informationen über das Modell enthüllen und zukünftig noch bessere Abstraktionen liefern, da dann z.B. alle Verbindungen mit Variablen, die nur mit dem negativen Literal von x_i verknüpft sind, auch reduziert werden können.

Literatur

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

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

[CCG+02] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani und Armando Tacchella. NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In *Computer-Aided Verification*, Jgg. 2404 of *LNCS*. Springer, 2002.

[CCK+02] Pankaj Chauhan, Edmund Clarke, James Kukula, Samir Sapra, Helmut Veith und Dong Wang. Automated Abstraction Refinement for Model Checking Large State Spaces Using SAT Based Conflict Analysis. In Mark Aagaard und John OâLeary, Hrsg., *Formal Methods in Computer-Aided Design*, Jgg. 2517 of *Lecture Notes in Computer Science*, Seiten 33–51. Springer Berlin / Heidelberg, 2002. 10.1007/3-540-36126-X3.

[DHJ+01] Matthew B. Dwyer, John Hatcliff, Roby Joehanes, Shawn Laubach, Corina S. Păsăreanu, Hongjun Zheng und Willem Visser. Tool-supported program abstraction for finite-state verification. In *Proc. of the 23rd International Conference on Software Engineering*, Seiten 177–187, Washington, DC, USA, 2001. IEEE Computer Society.

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

[EKS02] Javier Esparza, Antonin Kucera und Stefan 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] Javier Esparza und Stefan Schwoon. *A BDD-based model checker for recursive programs*. LNCS Volume 2102, 324-336, Springer, 2001.

[GS97] Susanne Graf und Hassen Saidi. Construction of abstract state graphs with PVS. In Orna Grumberg, Hrsg., *Computer Aided Verification*, Jgg. 1254 of *LNCS*, Seiten 72–83. Springer, 1997.

[HJMS02] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar und Grégoire Sutre. Lazy abstraction. In *Proc. of the 29th symposium on principles of programming languages*, Seiten 58–70, New York, USA, 2002. ACM.

[HM09] Roberto Hoffmann und Paul Molitor. Guiding property development with SAT-based coverage calculation. *Midwest Symposium on Circuits and Systems*, Seiten 1199–1202, 2009.

[Hof05] Roberto Hoffmann. A SAT Solving Framework: Conflict Analysis and Learning. Diplomarbeit, Institute of Computer Sciences, MLU Halle-Wittenberg, 2005.

[KP00] Yonit Kesten und Amir Pnueli. Control and data abstraction: cornerstones of practical formal verification. *Int. Journal Software Tools for Technology Transfer*, 2:328–342, 2000.

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

[KZR09] Raimund Kirner, Wolf Zimmermann und Dirk Richter. On Undecidability Results of Real Programming Languages. In *15. colloquium prog. lang. (KPS)*, 2009.

[MA03] Kenneth L. McMillan und Nina Amla. Automatic Abstraction without Counterexamples. In Hubert Garavel und John Hatcliff, Hrsg., *Tools and Algorithms for the Construction and Analysis of Systems*, Jgg. 2619 of *LNCS*, Seiten 2–17. Springer Berlin / Heidelberg, 2003.

[Muc97] Steven Muchnick. *Advanced Compiler Design and Implem.* Morgan Kaufmann, 1997.

[NK00] Kedar S. Namjoshi und Robert P. Kurshan. Syntactic Program Transformations for Automatic Abstraction. In *Computer Aided Verification*, Jgg. 1855 of *LNCS*, Seiten 435–449. Springer, 2000.

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

[RB12] Dirk Richter und Christian Berg. Exact Gap Computation for Code Coverage Metrics in ISO-C. In *Proc. of 7th international Workshop on Model Based Testing*, 2012.

[RH11] Dirk Richter und Roberto Hoffmann. Spezifikationsgetriebene Abstraktion fuer Kellersysteme. In *16. colloquium prog. lang. (KPS)*, 2011.

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

[Sch02] Stefan Schwoon. *Model-Checking Pushdown Systems*. TU München, 2002.

[SSE05] Dejvuth Suwimonteerabuth, Stefan Schwoon und Javier Esparza. *jMoped: A Java Bytecode Checker Based on Moped*. Tools and Alg. for Construction and Analysis of Systems, LNCS, Springer, 2005.

[SUM96] Henny Sipma, Tomas Uribe und Zohar Manna. Deductive model checking. In Alur und Henzinger, Hrsg., *Computer Aided Verification*, Jgg. 1102 of *LNCS*, Seiten 208–219. Springer, 1996.

[Til05] Daniel Tille. A SAT Solving Framework: Splitting Strategies. Diplomarbeit, Institute of Computer Sciences, MLU Halle-Wittenberg, 2005.

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