Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 144 (2006) 79–94
www.elsevier.com/locate/entcs

Model Checking Linear Programs with Arrays

Alessandro Armando [1]

AI-Lab, DIST, Università di Genova, Italy

Massimo Benerecetti [2]

Dip. di Scienze Fisiche, Università di Napoli "Federico II", Italy

Jacopo Mantovani [3]

AI-Lab, DIST, Università di Genova, Italy

Abstract

In previous work we proposed Linear Programs as a fine grained model for imperative programs, and showed how the model checking procedure used in SLAM can be generalised to a model checking procedure for Linear Programs. In this paper we show that our model checking procedure for linear programs can be extended in such a way to support the analysis of linear programs featuring a symbol for undefined values and conditional expressions. This extension is particularly important as it paves the way to the construction of model checking procedures for wider classes of imperative programs such as, e.g., linear programs with arrays. We provide a detailed account of a symbolic model checking procedure for this extended class of linear programs, discuss its implementation in the EUREKA tool, and present experimental results that confirm its effectiveness in the analysis of linear programs with arrays.

Keywords: Software Model Checking, Linear Constraints, Arrays, Linear Arithmetic, Decision Procedures

1 Introduction

Counterexample-guided abstraction refinement is one of the leading approaches to software model checking (e.g. [9,5,14]). In this context, Boolean Programs

[1] Email: armando@dist.unige.it
[2] Email: massimo.benerecetti@na.infn.it
[3] Email: jacopo@dist.unige.it

1571-0661/$ – see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.entcs.2006.01.006

(i.e. programs with the usual control constructs but whose variables are restricted to range over boolean values) have been proposed, e.g., in the SLAM project [3], as possible abstractions of imperative programs, which are gradually refined into more and more concrete programs. The advantage is that effective model checking procedures for boolean programs exist (e.g. [4,12]). While the approach has proven very effective on specific application areas such as device drivers programming [5,14], its effectiveness on other, more mundane classes of programs has to be ascertained. Notice that since the detection of a spurious execution trace leads to a new iteration of the check-and-refine loop, the efficiency of the approach depends in a critical way on the number of spurious execution traces allowed by the abstract program. Of course, the closer is the abstraction to the original program the smaller is the number of spurious execution traces that may be necessary to analyse.

In a previous paper [2] we proposed Linear Programs as a finer grained model for imperative programs and showed how the model checking procedure used in SLAM can be generalised to a model checking procedure for Linear Programs. Linear Programs differ from boolean programs in that program variables can range over a numeric domain $\mathcal{D}$ (e.g. $\mathbb{Z}$ or $\mathbb{R}$); moreover, all conditions and assignments to variables involve linear expressions, i.e. expressions of the form $c_0 + c_1 * x_1 + \cdots + c_n * x_n$, where $c_0, \ldots, c_n$ are numeric constants in $\mathbb{Z}$ and $x_1, \ldots, x_n$ are program variables ranging over $\mathcal{D}$. Linear Programs can explicitly encode complex correlations between data and control that must necessarily be abstracted away when using boolean programs.

In this paper we show that our model checking procedure for linear programs can be extended in such a way to support the analysis of generalised linear programs, i.e. linear programs featuring *(i)* a symbol for undefined values and *(ii)* conditional expressions. This extension is particularly important as it paves the way to the construction of model checking procedures for wider classes of imperative programs such as, e.g., linear programs with arrays.

The paper is structured as follows. In Section 2 we show through a worked out example how linear programs with arrays can be model checked by using a model checker for generalised linear programs as a black box. In Section 3 we define the syntax and the semantics of generalised linear programs. In Section 4 we provide a detailed account of a symbolic model checking procedure for generalised linear programs. In Section 5 we discuss the implementation of our ideas in the EUREKA [2] tool and present preliminary experimental results that confirm their effectiveness.

2 Model Checking Linear Programs with Arrays

Our approach to model checking linear programs with arrays is based on the idea of abstracting away all array elements from the initial program, and then of incrementally refining the resulting abstract program by including array elements as suggested by the refinement process.

P	$\hat{P}_0$	$\hat{P}_1$
`void main() {`	`void main() {`	`void main() {`
`  int i,a[3];`	`  int i;`	`  int i,a1;`
`[1] a[1] = 1;`	`[1] ;`	`[1] a1 = (1==1)?1: u;`
`[2] i = 0;`	`[2] i = 0;`	`[2] i = 0;`
`[3] while(a[i]!=1&& i<3)`	`[3] while(u!=1&& i<3)`	`[3] while(((i==1)?a1 : u)!=1 && i<3)`
`[4]  { a[i] = 2*i;`	`[4]  { ;`	`[4]  { a1 = (i==1) ? 2*i : a1;`
`[5]  i = i+1; }`	`[5]  i = i+1; }`	`[5]  i = i+1; }`
`[6] if(i > 1)`	`[6] if(i > 1)`	`[6] if(i > 1)`
`[7]  ERROR: ; }`	`[7]  ERROR: ; }`	`[7]  ERROR: ; }`

Fig. 1. A simple program (P), the initial abstraction ($\hat{P}_0$), and its refinement ($\hat{P}_1$).

Let P be the linear program with arrays of Figure 1. We start by abstracting P into program $\hat{P}_0$ by replacing every occurrence of array expressions with the symbol `u` (denoting an arbitrary value of numeric type) and by replacing every assignment to array elements with a skip statement (`;`).

A model checker for linear programs is then applied to $\hat{P}_0$ to determine the reachability of the line labelled with `ERROR` label. The trace `1`, `2`, `3`, `4`, `5`, `3`, `4`, `5`, `3`, `4`, `5`, `3`, `6`, `7` is detected by model checker. (How this is done is explained in the following sections.) This trace corresponds to executing three iterations of the `while` loop (lines `[3]`–`[5]`) which leaves variable `i` with the value 3. Therefore, the condition of the `if` statement at line `[6]` evaluates to true and the line labelled `ERROR` is finally reached.

The feasibility check of the above trace w.r.t. P is done by generating a set of formulae whose satisfying valuations correspond to all the possible executions of the sequence of statements of P corresponding to the trace under consideration. This is done by first renaming the variables occurring in the statements in such a way that no variable is assigned twice during execution (as done in [8,10]) and then by generating formulae encoding the behaviour of the renamed statements. Table 1 shows the sequence of original statements, the renamed statements, and the associated formulae for the trace above.

The resulting set of formulae is then fed to a theorem prover. If it is

found unsatisfiable then the trace is not executable in P, whereas if it is found satisfiable then we can conclude that the trace is also executable in P. In our example the set of formulae (see rightmost column in Table 1) is found to be unsatisfiable. The formulae that contributed to the proof of unsatisfiability are those associated with steps 1, 2, 4, 5, and 6. Notice the term $a_4[i_5]$ (with $i_5 = 1$ given by the context) occurs in the formula associated with step 6. This means that in order to rule out the above trace we must refine $\hat{P}_0$ by including the element of `a` at position 1 thereby obtaining $\hat{P}_1$. In the general case if $k_1, \ldots, k_n$ are the indexes for array a to be included in the refined program, then the refinement step amounts to replacing every expression of the form $a\texttt{[}e\texttt{]}$ with the conditional expression $(e \texttt{==} k_1 \texttt{?} a_{k_1} : (e \texttt{==} k_2 \texttt{?} a_{k_2} : \ldots (e \texttt{==} k_n \texttt{?} a_{k_n} : \mathfrak{u})))$, and then every assignment of the form $a\texttt{[}e_1\texttt{]} \texttt{=} e_2\texttt{;}$ with the (parallel) assignment

$$a_{k_1}, \ldots, a_{k_n} \texttt{=} (e_1 \texttt{==} k_1 \texttt{?} e_2 : a_{k_1}), \ldots, (e_1 \texttt{==} k_n \texttt{?} e_2 : a_{k_n})\texttt{;}$$

where $a_{k_1}, \ldots, a_{k_n}$ are new variables of numeric type, each corresponding to a different array element to be included in the refined program. (If $n = 0$, the assignment above reduces to a skip (`;`) statement.) The application of the model checking procedure for linear programs to $\hat{P}_1$ reveals that the error statement cannot be reached in $\hat{P}_1$ and from that we can conclude that the error statement is not reachable in P. A technical presentation of the above abstraction technique and of its fundamental properties can be found in [1].

3 Linear Programs and Linear Programs with Arrays

Let W be a set of program variables. Let $V = \{v_1, \ldots, v_l\} \subseteq W$ be a set of program variables ranging over $\mathcal{D}$ and let $A = \{a_1, \ldots, a_m\} \subseteq W$ be a set of array variables. An *array variable* $a \in A$ is a program variable equipped with a positive integer $dim(a)$, the *size of the array*. The sets A and V form a partition of W, i.e. $W = V \cup A$ and $V \cap A = \emptyset$. Moreover, let $\mathfrak{u}$ be a distinct symbol standing for undefined. The sets E (B) of *generalised (boolean resp.) linear expressions with arrays* (henceforth, simply *linear expression with arrays*) are defined by the following BNF production rules:

$$E ::= \mathfrak{u} \mid \mathbb{Z} \mid V \mid \mathbb{Z} \texttt{*} E \mid E \texttt{+} E \mid (B \texttt{?} E : E) \mid A\texttt{[}E\texttt{]}$$

$$B ::= (E \ op \, E)$$

where $op \in \{\texttt{>=}, \texttt{=<}, \texttt{<}, \texttt{>}, \texttt{==}, \texttt{!=}\}$. The definition of *generalised linear expression without arrays* (*linear expression*, for short) is subsumed by the above. In

Step	Line	Original Statement	Renamed Statement	Formula
1	[1]	`a[1] = 1;`	$\texttt{a}_1[1] = 1;$	$a_1 = \text{store}(a_0, 1, 1)$
2	[2]	`i = 0;`	$\texttt{i}_2 = 0;$	$i_2 = 0$
3	[3]	`assume(a[i] != 1 && i < 3);`	$\texttt{assume}(\texttt{a}_1[\texttt{i}_2] \texttt{ != } 1 \texttt{ && } \texttt{i}_2 \texttt{ < } 3);$	$a_1[i_2] \neq 1 \wedge i_2 < 3$
4	[4]	`a[i] = 2 * i;`	$\texttt{a}_4[\texttt{i}_2] = 2 \texttt{ * } \texttt{i}_2;$	$a_4 = \text{store}(a_1, i_2, 2 * i_2)$
5	[5]	`i = i + 1;`	$\texttt{i}_5 = \texttt{i}_2 + 1;$	$i_5 = i_2 + 1$
6	[3]	`assume(a[i] != 1 && i < 3);`	$\texttt{assume}(\texttt{a}_4[\texttt{i}_5] \texttt{ != } 1 \texttt{ && } \texttt{i}_5 \texttt{ < } 3);$	$a_4[i_5] \neq 1 \wedge i_5 < 3$
7	[4]	`a[i] = 2 * i;`	$\texttt{a}_7[\texttt{i}_5] = 2 \texttt{ * } \texttt{i}_5;$	$a_7 = \text{store}(a_4, i_5, 2 * i_5)$
8	[5]	`i = i + 1;`	$\texttt{i}_8 = \texttt{i}_5 + 1;$	$i_8 = i_5 + 1$
9	[3]	`assume(a[i] != 1 && i < 3);`	$\texttt{assume}(\texttt{a}_7[\texttt{i}_8] \texttt{ != } 1 \texttt{ && } \texttt{i}_8 \texttt{ < } 3);$	$a_7[i_8] \neq 1 \wedge i_8 < 3$
10	[4]	`a[i] = 2 * i;`	$\texttt{a}_{10}[\texttt{i}_8] = 2 \texttt{ * } \texttt{i}_8;$	$a_{10} = \text{store}(a_7, i_8, 2 * i_8)$
11	[5]	`i = i + 1;`	$\texttt{i}_{11} = \texttt{i}_8 + 1;$	$i_{11} = i_8 + 1$
12	[6]	`assume(!(a[i] != 1 && i < 3));`	$\texttt{assume}(\texttt{!}(\texttt{a}_{10}[\texttt{i}_{11}] \texttt{ != } 1 \texttt{ && } \texttt{i}_{11} \texttt{ < } 3));$	$\neg(a_{10}[i_{11}] \neq 1 \wedge i_{11} < 3)$
13	[6]	`assume(i > 1);`	$\texttt{assume}(\texttt{i}_{11} \texttt{ > } 1);$	$i_{11} > 1$

Table 1
Checking the trace for feasibility.

the following, E_W will denote the set of linear expressions with arrays over W. A *linear program (with arrays)* is a program with the usual control-flow constructs (`if`, `while`, `assert`) and procedural abstraction with call-by-value parameter passing and recursion. Variables range over $\mathcal{D}$; moreover, all conditions and assignments to variables involve linear expressions (with arrays, resp.). We denote the set of numeric variables and array variables of P with V_P and A_P, respectively.

We assume that every occurrence of $(\texttt{!}\, b_1)$, $(b_1 \texttt{ \&\& } b_2)$, and $(b_1 \texttt{ || } b_2)$ in a program is replaced by the equivalent expression $(b_1 \texttt{ ? } 0 : 1)$, $(b_1 \texttt{ ? } b_2 : 0)$, and $(b_1 \texttt{ ? } 1 : b_2)$ respectively. Moreover, every boolean linear expression b occurring outside the guard of a conditional expression, is replaced by the following equivalent linear expression $(b \texttt{ ? } 1 : 0)$. Therefore, after the rewriting, boolean expressions will only occur within the guards of conditional expressions. We also assume that a skip statement (`;`) is added immediately after each procedure call in the original program. In this way the return point of a procedure call is unique, explicit, and labelled within the control flow graph of the program. Finally, as in [4] and without loss of generality, we assume that all variable names and statement labels are globally unique.

The *control flow graph* of a program P is a directed graph $G_P = (N_P, \text{Succ}_P)$,

where $N_P = \{0, 1, \ldots, n, n+1, n+p\}$ is the set of vertices [4] and $\mathrm{Succ}_P : N_P \to 2^{N_P}$ maps each vertex in the set of its successors. For every vertex i such that $1 \leq i \leq n$, s_i denotes the program statement corresponding to i. If s_i is $\texttt{if}(e)$, $\texttt{while}(e)$, or $\texttt{assert}(e)$ then $\mathrm{Succ}_P(i) = \{\mathrm{Tsucc}_P(i), \mathrm{Fsucc}_P(i)\}$, where $\mathrm{Tsucc}_P(i)$ ($\mathrm{Fsucc}_P(i)$) denotes the successor of i when e evaluates to true (false, resp.). If s_i is $\texttt{assert}(e)$, then $\mathrm{Fsucc}_P(i) = 0$. If pr is a procedure in P, then $\mathrm{First}_P(pr)$ is the vertex corresponding to the first statement in pr, and $\mathrm{Exit}_P(pr)$ is the exit vertex of pr. If s_i is a procedure call $pr(\mathbf{e})$; then $\mathrm{Succ}_P(i) = \{\mathrm{First}_P(pr)\}$ and $\mathrm{RetPt}_P(i)$ is the vertex of the ; statement immediately following the procedure call. Moreover, if s_i is a statement occurring in the body of a procedure pr, then $\mathrm{ProcOf}_P(i) = pr$. If s_i is a return statement in a procedure pr, then $\mathrm{Succ}_P(i) = \{\mathrm{Exit}_P(pr)\}$, and $\mathrm{Succ}_P(\mathrm{Exit}_P(pr)) = \{j \in N_P : s_i = pr(\boldsymbol{e}); \text{ and } j = \mathrm{RetPt}_P(i)\}$. Finally, if $\mathrm{Succ}_P(i_1) = \{i_2\}$, we define $\mathrm{sSucc}_P(i_1) = i_2$. We also assume the existence of a procedure called $\texttt{main}$ and that $\mathrm{First}_P(\texttt{main}) = 1$.

Given a program P, $\mathrm{Globals}_P$ denotes the set of global variables of P and, for every $i \in N_P$, $\mathrm{Formals}_P(i)$ is the set of formal parameters of the procedure containing i, while $\mathrm{Locals}_P(i)$ is the set of the local variables in scope at vertex i. Moreover, we define $\mathrm{InScope}_P(i) = \mathrm{Globals}_P \cup \mathrm{Locals}_P(i)$.

Let W be a set of program variables, a *valuation* ω *over* W is a total function mapping the numeric variables in $V \subseteq W$ into $\mathcal{D}$ and each array variable $a \in A \subseteq W$ into a finite mapping from $\{0, \ldots, dim(a) - 1\}$ into $\mathcal{D}$. A *state of a linear program with arrays* P is a pair $\langle i, \omega \rangle$, where i is a vertex of the control flow graph of P and ω is a valuation over $W \cap \mathrm{InScope}_P(i)$. Thus, ω is a total function over $\mathrm{InScope}_P(i)$. The definition of *state of a linear program* is subsumed by the above. We lift ω to a total function $\overline{\omega} : E_W \longrightarrow 2^{\mathcal{D}}$ over

[4] Vertexes from 1 to n are associated with program statements, vertex 0 models the failure of $\texttt{assert}$ statements, and vertexes from $n+1$ to $n+p$ are the "exit" vertexes of procedures, where p is the number of procedures defined in P.

linear expressions with arrays defined as follows:

$$\overline{\omega}(e) = \begin{cases} \{e\} & \text{if } e \in \mathbb{Z} \\ \{\omega(e)\} & \text{if } e \in V \\ \{d \in \omega(a)(d_1) : d_1 \in \overline{\omega}(e_1)\} & \text{if } e = a\texttt{[}e_1\texttt{]} \\ & \text{and } \overline{\omega}(e_1) \subseteq \{0, \ldots, dim(a) - 1\} \\ \{c \cdot d_1 : d_1 \in \overline{\omega}(e_1)\} & \text{if } e = c \texttt{*} e_1 \\ \{d_1 \; op \, d_2 : d_1 \in \overline{\omega}(e_1) \text{ and } d_2 \in \overline{\omega}(e_2)\} & \text{if } e = e_1 \; op \, e_2 \\ & \text{with } op \in \{\texttt{>=}, \texttt{=<}, \texttt{<}, \texttt{>}, \texttt{==}, \texttt{!=}, \texttt{+}\} \\ \overline{\omega}(e_1) \cup \overline{\omega}(e_2) & \text{if } e = (b \texttt{?} e_1 : e_2) \text{ and } \{0, d\} \subseteq \overline{\omega}(b) \\ & \text{for some } d \neq 0 \\ \overline{\omega}(e_1) & \text{if } e = (b \texttt{?} e_1 : e_2) \text{ and } 0 \notin \overline{\omega}(b) \\ \overline{\omega}(e_2) & \text{if } e = (b \texttt{?} e_1 : e_2) \text{ and } \overline{\omega}(b) = \{0\} \\ \mathcal{D} & \text{otherwise} \end{cases}$$

The intuition is that $\overline{\omega}(e)$ is the set of all the values of e that are compatible with ω. All the occurrences of the $\mathfrak{u}$ symbol, as well as those corresponding to an out-of-range access to an array, within an expression e are modelled by nondeterministically assigning an arbitrary element in $\mathcal{D}$ to the corresponding subexpression. $\overline{\omega}$ is extended to k–tuples $\mathbf{e}$ of expressions in the obvious way.

State transitions in P are denoted by $\langle i_1, \omega_1\rangle \xrightarrow{\sigma}_P \langle i_2, \omega_2\rangle$ where σ is either the symbol ϵ or an expression of the form $\textsc{call}(i, \omega)$ or $\textsc{ret}(i, \omega)$ for $i \in N_P$ such that there exists a procedure call s_j with $i = \mathrm{RetPt}(j)$ and $\omega : \mathrm{Locals}_P(j) \to \mathcal{D}$ (Terminals of the form $\textsc{call}(i, \omega)$ and $\textsc{ret}(i, \omega)$ represent the entry and exit points of the procedure invoked by s_j respectively). We use bold letters such as $\mathbf{x}$ to denote a vector of variables, elements or expressions. We also allow for parallel assignment, denoted by $\mathbf{x} \texttt{=} \mathbf{e}\texttt{;}$. Moreover, let $\mathbf{c} = \langle c_1, c_2, \ldots, c_n\rangle$ and $\mathbf{d} = \langle d_1, d_2, \ldots, d_n\rangle$ be n–tuples of values in a set X and in $\mathcal{D}$ respectively; for any function $f : X \to \mathcal{D}$ by $f[\mathbf{d}/\mathbf{c}]$ we denote the function f' such that $f'(c_k) = d_k$ for all $k = 1, 2, \ldots, n$, and $f'(c) = f(c)$ for all $c \neq c_k$ and $k = 1, 2, \ldots, n$. State transitions of a program P are defined as follows:

- if s_{i_1} is a skip (`;`), or a `return` statement, then

$$\langle i_1, \omega_1\rangle \xrightarrow{\epsilon}_P \langle \mathrm{sSucc}_P(i_1), \omega_1\rangle;$$

- if s_{i_1} is a parallel assignment $\mathbf{y} \texttt{=} \mathbf{e}\texttt{;}$ then

$$\langle i_1, \omega_1\rangle \xrightarrow{\epsilon}_P \langle \mathrm{sSucc}_P(i_1), \omega_1[\mathbf{d}/\mathbf{y}]\rangle,$$

  for $\mathbf{d} \in \overline{\omega}_1(\mathbf{e})$;

- if s_{i_1} is an assignment $a\texttt{[}e_1\texttt{]} = e_2\texttt{;}$ then

$$\langle i_1, \omega_1\rangle \xrightarrow{\epsilon}_P \langle \mathrm{sSucc}_P(i_1), \omega_1[(\omega(a)[d_2/d_1])\,/a]\rangle,$$

  for $d_1 \in \overline{\omega}_1(e_1)$ and $d_2 \in \overline{\omega}_1(e_2)$;
- if s_{i_1} is a statement of the form $\texttt{if}(e)$, $\texttt{while}(e)$, or $\texttt{assert}(e)$, then

$$\langle i_1, \omega_1\rangle \xrightarrow{\epsilon}_P \langle i_2, \omega_1\rangle,$$

  where $i_2 = \mathrm{Fsucc}_P(i_1)$ if $0 \in \overline{\omega}_1(e)$ and $i_2 = \mathrm{Tsucc}_P(i_1)$ if $d \in \overline{\omega}_1(e)$ for some $d \neq 0$;
- if $s_{i_1} = pr(\mathbf{e})\texttt{;}$ then

$$\langle i_1, \omega_1\rangle \xrightarrow{\textsc{call}(\mathrm{RetPt}_P(i_1),\omega)}_P \langle \mathrm{First}_P(pr), \omega_2\rangle,$$

  where $\omega : \mathrm{Locals}_P(i_1) \to \mathcal{D}$ is such that $\omega(\mathbf{x}) = \omega_1(\mathbf{x})$, $\omega_2(\mathbf{y}) \in \overline{\omega}_1(\mathbf{e})$, and $\omega_2(\mathbf{g}) = \omega_1(\mathbf{g})$ where $\mathbf{x} = \mathrm{Locals}_P(i_1)$, $\mathbf{y} = \mathrm{Formals}_P(i_2)$, and $\mathbf{g} = \mathrm{Globals}_P$;
- if i_1 is the exit vertex of pr then

$$\langle i_1, \omega_1\rangle \xrightarrow{\textsc{ret}(i_2,\omega)}_P \langle i_2, \omega_2\rangle,$$

  where $i_2 \in \mathrm{Succ}_P(i_1)$, $\omega_2(\mathbf{g}) = \omega_1(\mathbf{g})$ for all $\mathbf{g} = \mathrm{Globals}_P$ and $\omega_2(\mathbf{x}) = \omega(\mathbf{x})$ for all $\mathbf{x} = \mathrm{Locals}_P(i_2)$.

A *path* of a program P is a sequence $\langle i_0, \omega_0\rangle \xrightarrow{\sigma_1}_P \langle i_1, \omega_1\rangle \xrightarrow{\sigma_2}_P \cdots \xrightarrow{\sigma_n}_P \langle i_n, \omega_n\rangle$ such that $\langle i_k, \omega_k\rangle \xrightarrow{\sigma_{k+1}}_P \langle i_{k+1}, \omega_{k+1}\rangle$ for $k = 0, \ldots, n-1$. (In the sequel we will abbreviate $\langle i_0, \omega_0\rangle \xrightarrow{\sigma_1}_P \cdots \xrightarrow{\sigma_n}_P \langle i_n, \omega_n\rangle$ with $\langle i_0, \omega_0\rangle \xrightarrow{\sigma_1\cdots\sigma_n}_P \langle i_n, \omega_n\rangle$.) Notice that not all paths represent potential execution paths: in a transition like $\langle i_1, \omega_1\rangle \xrightarrow{\textsc{ret}(i_2,\omega)}_P \langle i_2, \omega_2\rangle$ where $i_1 = \mathrm{Exit}_{pr}$, the valuation ω can be chosen arbitrarily and therefore ω_2 is not guaranteed to coincide with ω_1 on the locals of the caller, as required by the semantics of procedure calls. A *valid path from* $\langle i_0, \omega_0\rangle$ *to* $\langle i_n, \omega_n\rangle$ describes the transmission of effects from $\langle i_0, \omega_0\rangle$ to $\langle i_n, \omega_n\rangle$ via a sequence of execution steps during which the call stack may temporarily grow deeper (because of procedure calls) but never shallower than its original depth. An *initialised path* is a path whose initial state is $\langle 1, \omega_0\rangle$ for some ω_0. A state $\langle i, \omega\rangle$ is *reachable* if and only if there exists an initialised valid path to $\langle i, \omega\rangle$. A vertex $i \in N_P$ is *reachable* if and only if there exists a valuation ω such that $\langle i, \omega\rangle$ is reachable.

4 Symbolic Model Checking of Linear Programs

Our model checking procedure extends the one described in [4], which is in turn an adaptation of the tabulation algorithm defined by Reps, Horwitz, and Sagiv in [17]. [5]

Let $i \in N_P$ and e be the node associated with the first statement of the procedure containing s_i. A *path edge* $\pi_i = \langle \omega_e, \omega_i \rangle$ *incident in* i is a pair of valuations such that there is a valid path $\langle 1, \omega_0 \rangle \xrightarrow{\sigma_1 \cdots \sigma_k}_P \langle e, \omega_e \rangle \xrightarrow{\sigma_l \cdots \sigma_m}_P \langle i, \omega_i \rangle$ for some valuation ω_0 and $0 \leq k \leq l \leq m$. In other words, a path edge represents a suffix of a valid path from $\langle 1, \omega_0 \rangle$ to $\langle i, \omega_i \rangle$.

For each node i in the control flow of the program, our procedure incrementally builds a symbolic representation of the set of path edges incident in i. Reachability of a statement s_i then boils down to checking whether the set of path edges incident in i is not empty. We present the procedure in three stages: we start with the definition of formulae of Linear Arithmetics encoding the semantics of parallel assignments (i.e. $\alpha(\mathbf{y}, \mathbf{e})$) and of linear expressions (i.e. $\beta^+(e)$ and $\beta^-(e)$); next we introduce the data structures (called abstract disjunctive linear constraints) that we use to symbolically represent sets of path edges; finally we provide an abstract characterisation of our model checking procedure as a (inductively defined) relation over N_P-indexed family of abstract disjunctive linear constraints.

Let e and ne be linear expressions, B a set of atomic boolean linear expressions. We define the relation $e \rightarrow (B, ne)$ to be the smallest relation such that $e \rightarrow (\emptyset, e)$ for all $e \in V \cup \mathbb{Z} \cup \{\mathtt{u}\}$ and closed under the following inference rules:

$$\frac{e_1 \rightarrow (B_1, ne_1) \quad e_1 \rightarrow (B_2, ne_2)}{(e_1 \; op \; e_2) \rightarrow (B_1 \cup B_2, (ne_1 \; op \; ne_2))} \text{ if } op \in \{\mathtt{*}, \mathtt{+}, \mathtt{>=}, \mathtt{=<}, \mathtt{<}, \mathtt{>}, \mathtt{==}, \mathtt{!=}\}$$

$$\frac{b \rightarrow (B, nb) \quad e_1 \rightarrow (B_1, ne_1)}{(b \,\mathtt{?}\, e_1 : e_2) \rightarrow (B \cup B_1 \cup \{nb^+\}, ne_1)} \qquad \frac{b \rightarrow (B, nb) \quad e_2 \rightarrow (B_2, ne_2)}{(b \,\mathtt{?}\, e_1 : e_2) \rightarrow (B \cup B_2 \cup \{nb^-\}, ne_2)}$$

where b^+ (b^-) is $b\,\mathtt{!=}\,0$ ($b\,\mathtt{==}\,0$, resp.) if b is a linear expression and b ($\mathtt{!}\,b$, resp.) if b is a boolean expression. It can be shown that if $e \rightarrow (B, ne)$, then both the linear expression ne and the set B of linear expressions do not contain conditionals.

Let U be an infinite set of variables such that $U \cap V = \emptyset$. If e is a linear

[5] For the sake of brevity here we describe a simplified version of the procedure the does not make use of Summary Edges, i.e. data structures that cache the result of the analysis of procedure calls thereby avoiding redundant work in subsequent computations.

expression, by e^* we indicate the expression obtained from e by replacing every occurrence of $\texttt{!}\, b$, $e_1 \texttt{!=} e_2$, and $e_1 \texttt{==} e_2$, with $\neg b$, $\neg(e_1 = e_2)$, and $e_1 = e_2$, respectively, and every occurrence of $\mathtt{u}$ in e with a distinct variable in U. If w is a formula and $u_1, \ldots, u_k$ the variables from U occurring in w, by $\exists U.w$ we denote the formula $\exists u_1. \ldots \exists u_k.w$. Notice that if e is a linear expression without conditionals, then e^* is a expression of the language of Linear Arithmetics.

Let $\mathbf{y}$ and $\mathbf{e}$ be k-tuples of variables and linear expressions with $k > 0$ such that no variable in $\mathbf{y}$ occurs in $\mathbf{e}$. We define

$$\alpha(\mathbf{y}, \mathbf{e}) = \bigvee \left\{ \bigwedge_{i=1}^{k} \left(\exists U.(y_i = ne_i \wedge \bigwedge B_i)^* \right) : e_i \rightarrow (B_i, ne_i) \text{ for } i = 1, \ldots, k \right\}.$$

Moreover, for a linear expression e we define

$$\beta^+(e) = \bigvee \{ \exists U.(ne^+ \wedge \bigwedge B)^* : e \rightarrow (B, ne) \}$$
$$\beta^-(e) = \bigvee \{ \exists U.(ne^- \wedge \bigwedge B)^* : e \rightarrow (B, ne) \}.$$

For example, if $e = (3 \texttt{*} \mathtt{x} \texttt{+} (\mathtt{y} \texttt{<} 0 \texttt{?} \mathtt{u} \texttt{:} \mathtt{y}))$ then $e \rightarrow (\{\mathtt{y} \texttt{<} 0\}, 3 \texttt{*} \mathtt{x} \texttt{+} \mathtt{u})$ and $e \rightarrow (\{\texttt{!}\, \mathtt{y} \texttt{<} 0\}, 3 \texttt{*} \mathtt{x} \texttt{+} \mathtt{y})$. Therefore $\alpha(z, e) = (\exists u_1.(y < 0 \wedge z = 3 * x + u_1) \vee (\neg(y < 0) \wedge z = 3 * x + y))$ which can be simplified to the logically equivalent formula $(y < 0 \vee (\neg(y < 0) \wedge z = 3 * x + y))$. Similarly, $\beta^+(e) = (\exists u_1.(y < 0 \wedge \neg(3 * x + u_1 = 0)) \vee (\neg(y < 0) \wedge \neg(3 * x + y = 0)))$ which can be simplified to $(y < 0 \vee (\neg(y < 0) \wedge \neg(3 * x + y = 0)))$.

Let e be an expression, by e' we denote the expression obtained from e by replacing each variable, say v, with the corresponding primed version (i.e. v'). This notation is extended over sets (tuples) of expressions in the obvious way, i.e. $E' = \{e' : e \in E\}$ ($\mathbf{e}' = \langle e'_1, \ldots, e'_n \rangle$ if $\mathbf{e} = \langle e_1, \ldots, e_n \rangle$, resp.).

A *Disjunctive Linear Constraint* D (DLC for short) is a formula of the form $D = \bigvee_i \exists U. \bigwedge_j c_{ij}$, where c_{ij} are linear constraints. The symbol $\perp$ stands for an unsatisfiable linear constraint. An *Abstract Disjunctive Linear Constraint of arity* n (ADLC for short) is an expression of the form $\lambda\mathbf{x}.\lambda\mathbf{x}'.D$, where D is a DLC and $\mathbf{x}$ is an n-tuple comprising the free variables occurring in D.[6] We define the following operations on DLCs:

- *Application.* Let $\lambda\mathbf{x}\mathbf{x}'.D$ be an ADLC of arity n and $\mathbf{s}$ and $\mathbf{t}$ be n-tuples of linear expressions. The *application of* $\delta = \lambda\mathbf{x}\mathbf{x}'.D$ *to* $(\mathbf{s}, \mathbf{t})$, in symbols $\delta(\mathbf{s}, \mathbf{t})$, is the DLC obtained by simultaneously replacing from D the variables in $\mathbf{x}$ ($\mathbf{x}'$) with the corresponding element of $\mathbf{s}$ ($\mathbf{t}$ resp.).

[6] In the sequel we will abbreviate $\lambda\mathbf{x}.\lambda\mathbf{x}'.D$ with $\lambda\mathbf{x}\mathbf{x}'.D$.

- *Conjunction and Disjunction.* Let D_1 and D_2 be DLCs, then $D_1 \sqcap D_2$ and $D_1 \sqcup D_2$ are any DLCs logically equivalent to $D_1 \wedge D_2$ and $D_1 \vee D_2$ respectively. These operations are extended to ADLCs as follows. Let δ_1 and δ_2 be ADLCs of the same arity, then $\delta_1 \sqcap \delta_2 = \lambda \mathbf{x}\mathbf{x}'.(\delta_1(\mathbf{x},\mathbf{x}') \sqcap \delta_2(\mathbf{x},\mathbf{x}'))$ and $\delta_1 \sqcup \delta_2 = \lambda \mathbf{x}\mathbf{x}'.(\delta_1(\mathbf{x},\mathbf{x}') \sqcup \delta_2(\mathbf{x},\mathbf{x}'))$.
- *Quantifier Elimination.* Let D be a DLC, then $\exists \mathbf{x}.D$ is any DLC equivalent to D obtained by eliminating from D the variables in $\mathbf{x}$.
- *Entailment.* Let δ_1 and δ_2 be ADLCs of the same arity, then $\delta_1 \sqsubseteq \delta_2$ iff all the pairs of valuations satisfying δ_1 satisfy also δ_2, i.e. $[\![\delta_1]\!] \subseteq [\![\delta_2]\!]$ where $[\![\delta]\!] = \{\langle \omega, \omega' \rangle : \models_{\omega \cup \omega'} \delta(\mathbf{x},\mathbf{x}'), \omega : V \to \mathcal{D}, \omega' : V' \to \mathcal{D}\}$.

By $\mathcal{A}(P)$ we denote the N_P-indexed family of sets of ADLCs such that $\mathcal{A}_i(P)$ is the set of ADLCs of arity $|\,\mathrm{InScope}_P(i)|$ for all $i \in N_P$. Let $\Delta, \Delta^1 \in \mathcal{A}(P)$[7], we define the binary relation $\Delta \rightarrowtriangle \Delta^1$ as follows. Let $i \in N_P$, $\Delta^1_j = \Delta_j$ for all $j \notin \mathrm{Succ}_P(i)$ and

- if i corresponds to a skip (`;`) or `return` statement, then

$$\Delta^1_{\mathrm{sSucc}_P(i)} = \Delta_{\mathrm{sSucc}_P(i)} \sqcup \Delta_i.$$

- if i corresponds to an assignment **y=e**, then

$$\Delta^1_{\mathrm{sSucc}_P(i)} = \Delta_{\mathrm{sSucc}_P(i)} \sqcup \lambda \mathbf{x}\mathbf{x}''.\exists \mathbf{x}'.(\Delta_i(\mathbf{x},\mathbf{x}') \sqcap \alpha(\mathbf{y}'',\mathbf{e}') \sqcap \mathbf{z}'' = \mathbf{z}'),$$

  where $\mathbf{x} = \mathrm{InScope}_P(i)$ and $\mathbf{z} = \mathrm{InScope}_P(i) \setminus \mathbf{y}$;[8]
- if i corresponds to an `if`(b), `while`(b), or `assert`(b) statement, then

$$\Delta^1_{\mathrm{Tsucc}_P(i)} = \Delta_{\mathrm{Tsucc}_P(i)} \sqcup \lambda \mathbf{x}\mathbf{x}'.(\Delta_i(\mathbf{x},\mathbf{x}') \sqcap \beta^+(b')),$$
$$\Delta^1_{\mathrm{Fsucc}_P(i)} = \Delta_{\mathrm{Fsucc}_P(i)} \sqcup \lambda \mathbf{x}\mathbf{x}'.(\Delta_i(\mathbf{x},\mathbf{x}') \sqcap \beta^-(b')\}),$$

  with $\mathbf{x} = \mathrm{InScope}_P(i)$;
- if i corresponds to a procedure call $pr(\mathbf{e})$, then

$$\Delta^1_{\mathrm{sSucc}_P(i)} = \Delta_{\mathrm{sSucc}_P(i)} \sqcup \lambda \mathbf{w}\mathbf{w}''.(\exists \mathbf{x}'.(\Delta_i(\mathbf{x},\mathbf{x}') \sqcap \alpha(\mathbf{y}'',\mathbf{e}') \sqcap \mathbf{g}'' = \mathbf{g}') \sqcap \mathbf{w} = \mathbf{w}''),$$

  with $\mathbf{x} = \mathrm{InScope}_P(i)$, $\mathbf{y} = \mathrm{Formals}_P(pr)$, $\mathbf{w} = \mathrm{InScope}_P(\mathrm{sSucc}_P(i))$, $\mathbf{g} = \mathrm{Globals}_P$, and $\mathbf{l} = \mathrm{Locals}_P(\mathrm{sSucc}_P(i))$;
- if i is the exit vertex of pr then

$$\Delta^1_j = \Delta_j \sqcup \lambda \mathbf{w}\mathbf{w}''.\exists \mathbf{w}'''.(\Delta_k(\mathbf{w},\mathbf{w}''') \sqcap \mathbf{y}'' = \mathbf{y}''' \sqcap \exists \mathbf{x}'\mathbf{z}''.(\Delta_i(\mathbf{x}',\mathbf{x}'') \sqcap \alpha(\mathbf{f}',\mathbf{e}''') \sqcap \mathbf{g}' = \mathbf{g}'''))$$

[7] With an abuse of notation we write $\Delta \in \mathcal{A}(P)$ for $\Delta_i \in \mathcal{A}_i(P)$, for each $i \in N_P$.
[8] We abbreviate $x'_1 = x_1 \wedge \ldots \wedge x'_n = x_n$ with $\mathbf{x}' = \mathbf{x}$, where $\mathbf{x} = \langle x_1, \ldots, x_n \rangle$.

if $j \in \mathrm{Succ}_P(i)$ and k such that $s_k = pr(\mathbf{e})$ and $\mathrm{RetPt}(k) = j$, $\mathbf{w} = \mathrm{InScope}_P(k)$, $\mathbf{y} = \mathrm{Locals}_P(k)$, $\mathbf{x} = \mathrm{InScope}_P(i)$, $\mathbf{z} = \mathrm{Locals}_P(i)$.

For example, let s_i be the parallel assignment $\mathtt{y_1, y_2 = ((y_1 > 0)\,?\,y_1 + 1 : u), y_2 + 1}$ and $\mathrm{InScope}_P(i) = \{\mathtt{x, y_1, y_2}\}$. Hence, $\mathbf{e} = \langle((y_1 > 0)\,?\,y_1 + 1 : \mathtt{u}), y_2 + 1\rangle$, $\mathbf{y} = \langle y_1, y_2\rangle$ and $\mathbf{z} = \langle x\rangle$. Then, $\alpha(\mathbf{y}'', \mathbf{e}') = ((y_1'' = y_1' + 1 \wedge y_2'' = y_2' + 1 \wedge y_1' > 0) \vee \exists u_1.(y_1' = u_1 \wedge y_2'' = y_2' + 1 \wedge \neg y_1' > 0))$. This can be simplified to $\alpha(\mathbf{y}'', \mathbf{e}') = ((y_1'' = y_1' + 1 \wedge y_2'' = y_2' + 1 \wedge y_1' > 0) \vee (y_2'' = y_2' + 1 \wedge \neg y_1' > 0))$. Therefore, the new set of path edges added to $\Delta_{\mathrm{sSucc}_P(i)}$ is represented by the following ADLC $\lambda x, y_1, y_2, x'', y_1'', y_2''.\exists x', y_1', y_2'.(\Delta_i(\langle x, y_1, y_2\rangle, \langle x', y_1', y_2'\rangle) \sqcap ((y_1'' = y_1' + 1 \wedge y_2'' = y_2' + 1 \wedge y_1' > 0 \wedge x'' = x') \vee (y_2'' = y_2' + 1 \wedge \neg y_1' > 0 \wedge x'' = x')))$.

As a further example, let s_i be the statement $\mathtt{if(y > 0\ \&\&\ y == 2 * u)}$ and $\mathrm{InScope}_P(i) = \{\mathtt{x, y}\}$, then (after some simplifications) $\beta^+(b') = \exists u_1.(y' > 0 \wedge y' = 2 * u_1)$ and $\beta^-(b') = \exists u_1.(y' > 0 \wedge \neg y' = 2 * u_1) \vee (\neg y' > 0)$. The new set of path edges added to $\Delta_{\mathrm{Tsucc}_P(i)}$ (to $\Delta_{\mathrm{Fsucc}_P(i)}$, resp.) is represented by the following ADLC $\lambda x.x'.y.y'.\exists u_1.(\Delta_i(\langle x, y\rangle, \langle x', y'\rangle) \sqcap (y' > 0 \wedge y' = 2 * u_1 \wedge x' = x))$ $(\lambda x.x'.y.y'.\exists u_1.(\Delta_i(\langle x, y_1, y_2\rangle, \langle x', y_1', y_2'\rangle) \sqcap ((y' > 0 \wedge \neg y' = 2 * u_1 \wedge x' = x) \vee (\neg y' > 0 \wedge x' = x)))$, resp.).

Let $\Delta \in \mathcal{A}(P)$, we define $[\![\Delta]\!]$ as the N_P–indexed family of sets $[\![\Delta]\!]_i$ such that $[\![\Delta]\!]_i = [\![\Delta_i]\!]$, for all $i \in N_P$.

The following result states the correctness of our procedure. A proof of is available in [1]. Let $\Delta^0 \in \mathcal{A}(P)$ such that $\Delta_1^0 = \lambda \mathbf{x}\mathbf{x}'.(\mathbf{x}' = \mathbf{x})$ with $\mathbf{x} = \mathrm{InScope}_P(1)$ and $\Delta_i^0 = \lambda \mathbf{x}_i\mathbf{x}_i'.\bot$ with $\mathbf{x}_i = \mathrm{InScope}_P(i)$ for all $i \in N_P \backslash \{1\}$.

Theorem 4.1 *Let Δ^0 be defined as above. The following fact holds: i is reachable if and only if there exists Δ^1 such that $\Delta^0 \rightarrowtriangle^* \Delta^1$ and $[\![\Delta_i^1]\!] \neq 0$.*

5 Implementation and experimental results

We have extended the model checking procedure for linear programs of EUREKA [2] along the lines discussed in Section 4 so that it now supports the analysis of linear programs extended with the $\mathtt{u}$ symbol and conditional expressions. Quantifier elimination is implemented by means of the Fourier-Motzkin method [16]. Entailment checking is done by using ICS v2.0 [11] as a decision procedure for the boolean combination of linear arithmetic constraints. This is done by reducing the problem of determining whether $\delta_1 \sqsubseteq \delta_2$ holds to that of determining the unsatisfiability of the formula $\delta_1(\mathbf{c}, \mathbf{c}') \wedge \neg\delta_2(\mathbf{c}, \mathbf{c}')$, where δ_1 and δ_2 are ADLCs of arity n and $\mathbf{c}$ is an n-tuple of distinct constants. In order to assess the effectiveness of the approach we have also developed in EUREKA a prototype implementation of the counterexample-guided abstraction

refinement procedure outlined in Section 2. In order to check the satisfiability of the formulae encoding the feasibility of the traces generated by the model checker (see Section 2) we use CVC Lite [6] as it is a complete prover for the union of the theory of arrays and linear arithmetics.

Preliminary experiments confirm the effectiveness of the proposed approach. For instance, EUREKA readily concludes that the program in Figure 1 is safe. Quite interestingly Blast [14] (which does support arrays and implements lazy boolean abstraction) incorrectly reports that the same program is unsafe. The reasons lie in that in Blast (as in the C language), `a[i]` is a shorthand for `*(a+i)`, and in that Blast models any "[...] expression $p + i$, where p is a pointer and i is an integer, as yielding a pointer value that points to the object pointed to by p" [14]. This means that all array elements are indistinguishable for Blast during analysis. Since SLAM deals with pointers in the same way, we expect it should exhibit the same behaviour, but we could not verify this since the SLAM toolkit is not publicly available. On other linear programs with arrays (successfully analysed by EUREKA) Blast reports an error due to the incapacity to discover new predicates for the abstraction.

We also compared EUREKA with CBMC [15]. [9] At the core of CBMC lies a procedure that encodes the (bounded) verification problem for C programs into a SAT formula. The number of propositional variables in the SAT formula generated by CBMC grows linearly with the size of the arrays defined in the input program. In contrast EUREKA, by considering the array elements in an incremental and counterexample-driven manner, uses—in many cases of interest—time and memory resources independent from the size of the arrays defined in the input program. (The program in Figure 1 is a typical example of this.)

More in general, we tested EUREKA, Blast, and CBMC on a number of C programs that make use of arrays and that we use as regression tests for EUREKA.

The results are summarised in Table 2. [10] The reader may refer to the EUREKA project webpage (URL: `http://www.ai.dist.unige.it/eureka`) for a detailed description of the experimental results. Array out of bounds properties are not shown here, but it is easy to see that they can be checked by instrumenting the code by putting `assert`$(e$ `<` $dim(a))$ `&&`$(e$ `>=` `0`$)$ before every occurrence of a`[`e`]`, for any $a \in A$ and $e \in E$.

[9] CBMC uses a "physical" model of memory: every integer variable is modelled as a word of n bits, where n is either 8, 16, 32, or 64. The default (used for our experiments) is $n = 32$.
[10] The *mout* results of CBMC depend on both time and resources spent in finding the proper unwinding assertions in order to automatically determine a safe bound for model checking the program. This limitation can be overcome by setting a bound at the command line. Of course then, bounded model checking may lead to incomplete answers from the tool.

Program	EUREKA	BLAST	CBMC
`array_init.c`	safe	error	safe
`array_init_assign.c`	safe	unsafe	safe
`complex_guard.c`	safe	unsafe	safe
`simple_swap.c`	safe	safe	safe
`sequential_swap_call.c`	safe	error	safe
`simple_array_inversion.c`	safe	unsafe	safe
`bubblesort_inner_loop.c`	unsafe	error	unsafe
`bubblesort.c`	unsafe	safe	unsafe
`array_max.c`	safe	error	safe
`wrong_loop.c`	unsafe	error	mout
`loop_on_input.c`	unsafe	unsafe	mout
`simple_control_on_input.c`	unsafe	unsafe	mout

Legenda:
- *error*: the tool halts and reports that it cannot find a suitable refinement.
- *safe*: no error state can be reached by the program.
- *unsafe*: an error state may be reached by the program.
- *mout*: the tool runs out of memory.

Table 2
Results of the experimental analysis

An alternative approach to model checking of infinite-state programs is proposed in [13], whereby the correctness of the original program is expressed as a single logical query of an extended constraint logic which is then decided using a combination of lazy abstraction, counterexample-based predicate inference, and proof-based explication. The work is closely related to ours, but the lack of a publicly available implementation prevented us from carrying out a comparative analysis.

6 Conclusions

The work presented in this paper relates to the body of work developed within the counterexample-guided (predicate) abstraction refinement paradigm.

We have proposed an alternative abstraction and refinement schema for a subset of the C programming language that employs linear arithmetics (with variables ranging over a numerical domain) and arrays, which allows for the analysis of a wide class of imperative programs.

We have provided a formal semantics and a symbolic model checking procedure for this extended class of Linear Programs. Preliminary experimental

results obtained with our prototype model checker EUREKA indicate that our procedure correctly analyses programs on which other tools either fail or provide incorrect answers.

In the future we plan to explore the application of widening techniques along the lines proposed in [7] to enforce termination of our model checking procedure. We also plan to extend the abstraction refinement procedure (e.g. by treating pointers) in order to be able to verify an even wider class of C programs.

References

[1] Armando, A., M. Benerecetti and J. Mantovani, *Abstracting linear programs with arrays into linear programs*, Technical report, available at: http://www.ai.dist.unige.it/eureka. University of Genova (2005).

[2] Armando, A., C. Castellini and J. Mantovani, *Software model checking using linear constraints.*, in: *ICFEM*, 2004, pp. 209–223.

[3] Ball, T., R. Majumdar, T. D. Millstein and S. K. Rajamani, *Automatic predicate abstraction of c programs*, in: *SIGPLAN PLDI Conference*, 2001, pp. 203–213.

[4] Ball, T. and S. K. Rajamani, *Bebop: A symbolic model checker for boolean programs*, in: *Proc. of SPIN*, 2000, pp. 113–130.

[5] Ball, T. and S. K. Rajamani, *Automatically validating temporal safety properties of interfaces*, in: *Proc. of SPIN* (2001), pp. 103–122.

[6] Barrett, C. and S. Berezin, *Cvc lite: A new implementation of the cooperating validity checker category b.*, in: *CAV*, 2004, pp. 515–518.

[7] Bultan, T., R. Gerber and W. Pugh, *Model-checking concurrent systems with unbounded integer variables: symbolic representations, approximations, and experimental results*, ACM Transactions on Programming Languages and Systems **21** (1999), pp. 747–789.

[8] Clarke, E., D. Kroening and F. Lerda, *A tool for checking ANSI-C programs*, in: K. Jensen and A. Podelski, editors, *Proc. of TACAS 2004*, LNCS **2988** (2004), pp. 168–176.

[9] Clarke, E., D. Kroening, N. Sharygina and K. Yorav, *Predicate abstraction of ansi-c programs using sat*, Form. Methods Syst. Des. **25** (2004), pp. 105–127.

[10] Clarke, E. M., D. Kroening, N. Sharygina and K. Yorav, *Predicate abstraction of ansi-c programs using sat.*, Formal Methods in System Design **25** (2004), pp. 105–127.

[11] de Moura, L., S. Owre, H. Ruess, J. Rushby and N. Shankar, *The ICS decision procedures for embedded deduction* (2004).

[12] Esparza, J. and S. Schwoon, *A BDD-based model checker for recursive programs*, in: *Proc. of CAV*, number 2102 in LNCS (2001), pp. 324–336.

[13] Flanagan, C., *Software model checking via iterative abstraction refinement of constraint logic queries* (2004), cP+CV'04.

[14] Henzinger, T. A., R. Jhala, R. Majumdar and G. Sutre, *Lazy abstraction*, in: *POPL 2002*, 2002, pp. 58–70.

[15] Kroening, D., E. Clarke and K. Yorav, *Behavioral consistency of C and Verilog programs using bounded model checking*, in: *Proc. of DAC 2003* (2003), pp. 368–371.

[16] Lassez, J.-L. and M. Maher, *On Fourier's Algorithm's for Linear Arithmetic Constraints*, Journal of Automated Reasoning **9** (1992), pp. 373–379.

[17] Reps, T., S. Horwitz and M. Sagiv, *Precise interprocedural dataflow analysis via graph reachability*, in: *Proc. of POPL '95* (1995), pp. 49–61.