Abstraction Refinement of Linear Programs with Arrays

Alessandro Armando[1], Massimo Benerecetti[2], and Jacopo Mantovani[1]

[1] AI-Lab, DIST, Università di Genova, Italy
[2] Dip. di Scienze Fisiche, Università di Napoli "Federico II", Italy

Abstract. In previous work we presented a model checking procedure for linear programs, i.e. programs in which variables range over a numeric domain and expressions involve linear combinations of the variables. In this paper we lift our model checking procedure for linear programs to deal with arrays via iterative abstraction refinement. While most approaches are based on predicate abstraction and therefore the abstraction is relative to sets of predicates, in our approach the abstraction is relative to sets of variables and array indexes, and the abstract program can express complex correlations between program variables and array elements. Thus, while arrays are problematic for most of the approaches based on predicate abstraction, our approach treats them in a precise way. This is an important feature as arrays are ubiquitous in programming. We provide a detailed account of both the abstraction and the refinement processes, discuss their implementation in the EUREKA tool, and present experimental results that confirm the effectiveness of our approach on a number of programs of interest.

1 Introduction

We present an abstraction refinement procedure for linear programs with arrays, i.e. programs in which variables and array elements range over a numeric domain and expressions involve linear combinations of variables and array elements. Unlike the approaches based on predicate abstraction in which the abstraction is relative to sets of predicates, in our approach the abstraction is relative to sets of program variables and array indexes. Thus while predicate abstraction uses Boolean programs as the target of the abstraction, in our approach we use linear programs for the same purpose. This is particularly attractive as linear programs can directly and concisely represent complex correlations among program variables and a small number of iterations of the abstraction refinement loop usually suffice to either prove or disprove that the original program enjoys the desired properties.

In previous work [1] we proposed a model checking procedure for linear programs. In [2] we extended our procedure to deal with undefined values and conditional expressions, thereby paving the way to the model checking procedure for linear programs with arrays described in this paper.

The ability to analyse linear programs with arrays is particularly important as arithmetic and arrays are ubiquitous in programming and many real-world

programs belong to this class. Moreover, most predicate abstraction techniques suffer from a severe lack of precision when dealing with arrays. For instance, SLAM [3] and BLAST [4] treat all the elements of an array as they were a single element and this makes their analysis grossly inaccurate for all programs that access or manipulate arrays even in a trivial way. In addition, as the theory of arrays does not offer suitable interpolants, the approaches in which the refinement step is based on interpolation (see, e.g., [5,6]) have difficulties in the discovery of useful predicates when traces involve arrays. On the other hand, the procedure described in this paper efficiently handles linear programs with arrays in a sound and precise way.

We have implemented our procedure in the EUREKA tool and carried out several experiments using a number of linear programs of interest including string manipulation and sorting algorithms. We compared the results of our experiments with two state-of-the-art tools that adopt predicate abstraction, namely BLAST and SATABS [7]. On all problems considered BLAST detects spurious errors, and our procedure scales better than SATABS as the size of the arrays handled by the programs increases. We also compared EUREKA with CBMC [8], a well-known bounded model checker for C programs, with largely favourable results.

In the next section we present our procedure through a worked out example. In Section 3 we define the syntax and the semantics of linear programs with arrays. In Section 4 we define our abstraction and state its fundamental properties. In Section 5 we discuss how spurious error traces are detected. In Section 6 we present the refinement process. In Section 7 we describe the implementation of our ideas in EUREKA and discuss the results of our experiments.

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 program variables and array elements from the initial program, and then incrementally refining the abstract program obtained in this way by including program variables and array elements as suggested by the refinement process.

Let P be the linear program with arrays in the leftmost column of Table 1. We start by abstracting P into program $\widehat{P}_0$ by replacing every occurrence of array expressions with the symbol $\mathtt{u}$ (denoting an arbitrary value of numeric type) and by replacing every assignment to array elements with a skip statement ($\mathtt{;}$). (For the sake of simplicity in our example we do not abstract away the program variable $\mathtt{i}$ which therefore occurs in $\widehat{P}_0$.)

By applying a model checker for linear programs to $\widehat{P}_0$ we get the execution trace $\mathtt{1, 2, 3, 4, 5, 3, 4, 5, 3, 6, 0}$ witnessing the violation of the assertion at line $\mathtt{6}$, where $\mathtt{0}$ is an additional node which is reached if and only if an assertion fails. This trace corresponds to the execution of two iterations of the $\mathtt{while}$ loop (lines $\mathtt{3}$–$\mathtt{5}$) which leaves variable $\mathtt{i}$ with value $\mathtt{2}$ and therefore leads to a violation of the assertion at line $\mathtt{6}$.

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

Line	P	$\widehat{P}_0$	$\widehat{P}_1$
	`void main() {`	`void main() {`	`void main() {`
	`  int i, a[30];`	`  int i;`	`  int i, a`^{1}`;`
[1]	`  a[1] = 9;`	`  ;`	`  a`^{1}` = (1==1)?9:a`^{1}`;`
[2]	`  i = 0;`	`  i = 0;`	`  i = 0;`
[3]	`  while(a[i]!=9) {`	`  while(u!=9) {`	`  while(((i==1)?a`^{1}`:u)!=9) {`
[4]	`    a[i] = 2*i;`	`    ;`	`    a`^{1}` = (i==1)?2*i:a`^{1}`;`
[5]	`    i = i+1; }`	`    i = i+1; }`	`    i = i+1; }`
[6]	`  assert(i<=1); }`	`  assert(i<=1); }`	`  assert(i<=1); }`

The feasibility check of the above trace w.r.t. P is done by generating a set of quantifier-free formulae whose satisfying valuations correspond to all possible executions of the sequence of statements of P corresponding to the trace under consideration. This is done by first putting the trace in Single Assignment Form [9] and then by generating quantifier-free formulæ encoding the behaviour of the statements. Table 2 shows the sequence of the original statements, the trace in Single Assignment Form, and the associated formulæ for the above trace.

Table 2. Checking the trace for feasibility

Step	Line	Original Statement	Renamed Statement	Formula
1	[1]	`a[1] = 9;`	`a₁[1] = 9;`	$a_1 = \text{store}(a_0, 1, 9)$
2	[2]	`i = 0;`	`i₁ = 0;`	$i_1 = 0$
3	[3]	`assume(a[i] != 9);`	`assume(a₁[i₁] != 9);`	$\text{select}(a_1, i_1) \neq 9$
4	[4]	`a[i] = 2 * i;`	`a₂[i₁] = 2 * i₁;`	$a_2 = \text{store}(a_1, i_1, 2 * i_1)$
5	[5]	`i = i + 1;`	`i₂ = i₁ + 1;`	$i_2 = i_1 + 1$
6	[3]	`assume(a[i] != 9);`	`assume(a₂[i₂] != 9);`	$\text{select}(a_2, i_2) \neq 9$
7	[4]	`a[i] = 2 * i;`	`a₃[i₂] = 2 * i₂;`	$a_3 = \text{store}(a_2, i_2, 2 * i_2)$
8	[5]	`i = i + 1;`	`i₃ = i₂ + 1;`	$i_3 = i_2 + 1$
9	[3]	`assume(!(a[i] != 9));`	`assume(!(a₃[i₃] != 9));`	$\neg(\text{select}(a_3, i_3) \neq 9)$
10	[6]	`assume(!(i <= 1));`	`assume(!(i₃ <= 1));`	$\neg(i_3 \leq 1)$

The resulting set of formulæ is then fed to a theorem prover. If it is found unsatisfiable (w.r.t. a suitably defined background theory) 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 formulæ (see rightmost column in Table 2) is found to be unsatisfiable. The formulæ that contributed to the proof of unsatisfiability are those associated with steps 1, 2, 4, 5, and 6. Moreover, the only term of the form $\text{select}(a, e)$ occurring in these formulæ is $\text{select}(a_2, i_2)$ (with $i_2 = 1$ given by the context). As we will see later in the paper, this suffice to conclude that in order to rule out the above trace we must refine $\widehat{P}_0$ by including the element of a at position 1. The resulting program, $\widehat{P}_1$, is obtained by replacing every expression of the form $a[e]$ with the conditional

expression $(e \mathop{==} 1 ? a^1 : \mathtt{u})$ and every assignment of the form $a[e_1] = e_2$; with the assignment

$$a^1 = (e_1 \mathop{==} 1 ? e_2 : a^1);$$

where a^1 is a new variable of numeric type corresponding to the array element of index 1. The application of the model checking procedure for linear programs to $\widehat{P}_1$ reveals that the error state cannot be reached in $\widehat{P}_1$ and from this it is possible to conclude that the error state is not reachable in P.

In the sequel, if P is a linear program with arrays, then by V_P and A_P we denote the set of numeric and array variables (resp.) of P. Moreover we assume that each array $a \in A_P$ is equipped with a positive integer $\dim(a)$ indicating the size of the array. Finally by R_P we denote the function mapping each $a \in A_P$ into the set $\{0, \ldots, \dim(a) - 1\}$.

A complete account of our abstraction refinement procedure for linear programs with arrays is given in Figure 1. The procedure takes as input a linear

> **procedure** AR(P,V,R)
> 1. $\widehat{P} \leftarrow \mathrm{abstract}(P, V, R)$;
> 2. $\mathit{Trace} \leftarrow \mathrm{model\text{-}check}(\widehat{P})$;
> 3. **if** $\mathit{Trace} = \mathrm{none}$ **then return** SAFE;
> 4. **if** ($V = V_P$ **and** $R = R_P$) **then return** Trace;
> 5. $\mathit{Formula} \leftarrow \mathrm{encode}(\mathit{Trace}, P)$;
> 6. $\mathit{Result} \leftarrow \mathrm{decide}(\mathit{Formula})$;
> 7. **if** $\mathit{SAT?}(\mathit{Result})$ **then return** Trace;
>   /* Result contains a proof of the unsatisfiability of $\mathit{Formula}$ */
> 8. $\langle V', R' \rangle \leftarrow \mathrm{refine}(\mathit{Trace}, \mathit{Result}, V, R)$;
> 9. **return** AR(P,V',R');

Fig. 1. Abstraction refinement of linear programs with arrays

program with arrays P, a subset $V \subseteq V_P$ and a function R mapping each array $a \in A_P$ into a subset of $R_P(a)$. Initially V is set to the empty set and R is set to the function that maps every $a \in A_P$ into the empty set. The procedure starts by abstracting P w.r.t. V and R. The resulting abstract program $\widehat{P}$ is then fed to the model checker for linear programs at line 2. If no execution trace violating an assertion is found in $\widehat{P}$, then the procedure halts at line 3 reporting that the original program is safe. Otherwise (i.e. if Trace contains an execution trace of $\widehat{P}$ that violates an assertion), the procedure checks at line 4 whether further refinement is possible. If this is not the case (this happens when $V = V_P$ and $R = R_P$), the procedure halts and returns Trace as an execution trace of P witnessing an assertion violation. Otherwise (i.e. if further refinement is possible) the procedure builds at line 5 a quantifier-free formula whose unsatisfiability implies the infeasibility of Trace w.r.t. P. The formula is then fed to a theorem prover at line 6. If the formula is found to be satisfiable by the theorem prover, then the procedure halts and returns Trace as an execution trace of P witnessing an assertion violation. Otherwise Result contains a proof of the unsatisfiability of the formula and the refinement procedure is invoked at line 8 with the task of

extending the set of program variables and the sets of array indexes to be used for the construction of a new, refined abstraction of the original program. This is done by the recursive call to the AR procedure at line 9.

3   Linear Programs and Linear Programs with Arrays

A *linear program with arrays* is a program with the usual control-flow constructs (`if`, `while`, `assert`) procedural abstraction with call-by-value parameter passing and recursion, plus an additional `assume` statement. Variables and array elements range over a numerical domain $\mathcal{D}$, e.g. $\mathbb{R}$, $\mathbb{Z}$, or $\mathbb{Z}_n$ (i.e. the integers modulo n) for $n \in \mathbb{N}$; moreover, conditions and assignments to variables and array elements involve linear expressions with arrays. The sets E of *generalised linear expressions with arrays* (henceforth, simply *linear expression with arrays*) and the set B of *Boolean linear expressions with arrays* are defined by the following BNF production rules:

$$E ::= \mathtt{u} \mid \mathbb{Z} \mid V_P \mid \mathbb{Z} * E \mid E + E \mid (B\, ?\, E : E) \mid A_P[E] \qquad B ::= (E\ op\ E)$$

where $op \in \{\mathtt{>=}, \mathtt{<=}, \mathtt{<}, \mathtt{>}, \mathtt{==}, \mathtt{!=}\}$ and $\mathtt{u}$ is a symbol representing an undefined value.

The definition of *generalised (Boolean, resp.) linear expression without arrays* (*(Boolean, resp.) linear expression*, for short) and of *linear programs without arrays* are subsumed by the above.

In the following we will consider only linear programs with arrays with no nested occurrences of arrays. This is without loss of generality, as nested occurrences of arrays can always be eliminated by introducing fresh variables. Moreover, we assume that the program is decorated with assertions in such a way to ensure that possible out-of-bounds array accesses always lead to an assertion violation.

For the sake of space, in the following we only present the semantics of linear programs with arrays without procedure calls and returns. We refer the reader to [2] for a complete account of the semantics that includes (recursive) procedures.

The *control flow graph* (CFG) of a program P is a directed graph $G_P = (N_P, \mathrm{Succ}_P)$, where $N_P = \{0, 1, \ldots, n\}$ is the set of vertexes[1] 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 $\mathtt{if}(e)$, $\mathtt{while}(e)$, or $\mathtt{assert}(e);$ 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 $\mathtt{assert}(e);$, then $\mathrm{Fsucc}_P(i) = 0$, while if s_i is $\mathtt{assume}(e);$, then $\mathrm{Succ}_P(i) = \{Tsucc_P(i)\}$. Finally, if $\mathrm{Succ}_P(i_1) = \{i_2\}$, we define $\mathrm{sSucc}_P(i_1) = i_2$.

Given a program P, $\mathrm{Globals}_P$ denotes the set of global variables of P and, for every $i \in N_P$, $\mathrm{Locals}_P(i)$ 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 $W_P = V_P \cup A_P$ be the set of program variables of P, a *valuation* ω *over* W_P is a total function mapping V_P into $\mathcal{D}$ and each $a \in A_P$ into a finite

---

[1] Vertexes from 1 to n are associated with program statements and vertex 0 models the failure of `assert` statements.

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

Fig. 2. Semantics of linear expressions

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 $W_P \cap \text{InScope}_P(i)$. Thus, ω is a total function over $\text{InScope}_P(i)$. The definition of *state of a linear program* is subsumed by the above.

We lift ω to a total function $\overline{\omega} : E \longrightarrow 2^{\mathcal{D}}$ over linear expressions with arrays defined as reported in Figure 2. The intuition is that $\overline{\omega}(e)$ collects the set of all the values of e which are compatible with the valuation ω. All the occurrences of the ʉ symbol, as well as those corresponding to an out-of-range access to an array, within an expression e are modelled by non-deterministically assigning an arbitrary element in $\mathcal{D}$ to the corresponding sub-expression. $\overline{\omega}$ is extended to k–tuples $\mathbf{e}$ of expressions in the obvious way.

State transitions in P are denoted by $\langle i_1, \omega_1 \rangle \longrightarrow_P \langle i_2, \omega_2 \rangle$, where the valuation ω_k is such that $\omega_k : \text{InScope}_P(i_k) \to \mathcal{D}$, for $k = 1, 2$. We use bold letters such as $\mathbf{x}$ to denote vectors of variables, elements or expressions. We also allow for parallel assignments, denoted by $\mathbf{x} = \mathbf{e};$. 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 (`;`), then $\langle i_1, \omega_1 \rangle \longrightarrow_P \langle \text{sSucc}_P(i_1), \omega_1 \rangle$;
- if s_{i_1} is a parallel assignment $\mathbf{y} = \mathbf{e};$ then $\langle i_1, \omega_1 \rangle \longrightarrow_P \langle \text{sSucc}_P(i_1), \omega_1[\mathbf{d}/\mathbf{y}] \rangle$, for $\mathbf{d} \in \overline{\omega}_1(\mathbf{e})$;
- if s_{i_1} is an assignment $a\,[e_1] = e_2;$ then $\langle i_1, \omega_1 \rangle \longrightarrow_P \langle \text{sSucc}_P(i_1), \omega_1[(\omega(a)[d_2/d_1])/a] \rangle$, for $d_1 \in \overline{\omega}_1(e_1)$ and $d_2 \in \overline{\omega}_1(e_2)$;
- if i_1 corresponds to $\texttt{assume}(e);$, then $\langle i_1, \omega_1 \rangle \longrightarrow_P \langle i_2, \omega_1 \rangle$, where $i_2 = \text{sSucc}_P(i_1)$ if $d \in \overline{\omega}_1(e)$ for some $d \neq 0$;
- if s_{i_1} is a statement of the form $\texttt{if}(e)$, $\texttt{while}(e)$, or $\texttt{assert}(e);$, then $\langle i_1, \omega_1 \rangle \longrightarrow_P \langle i_2, \omega_1 \rangle$, where $i_2 = \text{Fsucc}_P(i_1)$ if $0 \in \overline{\omega}_1(e)$ and $i_2 = \text{Tsucc}_P(i_1)$ if $d \in \overline{\omega}_1(e)$ for some $d \neq 0$.

Let $\longrightarrow_P^*$ denote the reflexive and transitive closure of $\longrightarrow_P$. A state $\langle i, \omega \rangle$ is *reachable* if and only if there exists a valuation ω_0 such that $\langle 1, \omega_0 \rangle \longrightarrow_P^* \langle i, \omega \rangle$. A vertex $i \in N_P$ is *reachable* if and only if there exist two valuations ω_0 and ω such that $\langle 1, \omega_0 \rangle \longrightarrow_P^* \langle i, \omega \rangle$. A *trace of P* is a sequence of nodes $i_0 i_1 \cdots i_n$ such that $\langle 1, \omega_0 \rangle \longrightarrow_P \langle i_1, \omega_1 \rangle \longrightarrow_P \cdots \longrightarrow_P \langle i_n, \omega_n \rangle$ for some valuations $\omega_0, \omega_1, \ldots, \omega_n$. By traces($P$) we denote the set of traces of P. An *error trace of P* is any trace of P ending with vertex 0.

In [1] we proposed a symbolic model checking procedure for linear programs based on the tabulation algorithm defined in [10]. In [2] we extended our procedure to support the analysis of linear programs with the $\mathfrak{u}$ symbol and conditional expressions. The resulting procedure can be used to implement the model checker (model-check) invoked by the AR procedure of Figure 1. For the purpose of this paper it suffices to know that model-check(P_0) is capable to detect and return an error trace (if any) of the linear program P_0 given as input.

It is worth pointing out that the framework described in this paper is independent from the domain of computation. On the other hand, the decidability of the model checking problem clearly depends on it: if $\mathcal{D} = \mathbb{R}$ or $\mathcal{D} = \mathbb{Z}$, then the problem is undecidable, whereas if $\mathcal{D} = \mathbb{Z}_n$ (for $n \geq 0$), then the problem is decidable.

4   Abstracting Linear Programs with Arrays into Linear Programs

Let R be a function mapping each array $a \in A_P$ into a subset of $R_P(a)$, and let $V \subseteq V_P$. The *set of abstractions of P w.r.t. R and V*, in symbols abstract(P, V, R), is the set of linear programs defined as follows. The set of program variables of $\widehat{P} \in$ abstract(P, V, R) is $\widehat{V} = V \cup \{a^k : a \in A_P, k \in R(a)\}$. Intuitively, a^k is a new variable representing in $\widehat{P}$ the $(k+1)$-th element of the array a. Given any linear expression e in P, an abstract version $\widehat{e}$ is obtained from e by replacing *(i)* every occurrence of the variables not in V with the undefined symbol $\mathfrak{u}$, *(ii)* every expression of the form $a\,[e]$ with abs($a\,[e], [k_1, \ldots, k_n]$), where $[k_1, \ldots, k_n]$ is some permutation of $R(a)$ and:

$$\text{abs}(a\,[e], []) = \mathfrak{u}$$
$$\text{abs}(a\,[e], [k_1, k_2, \ldots, k_n]) = (\widehat{e} \texttt{==} k_1 \texttt{?}\, a^{k_1} : \text{abs}(a\,[e], [k_2, \ldots, k_n])),$$

and *(iii)* every conditional expression of the form $e' \texttt{?}\, \mathfrak{u} : \mathfrak{u}$, possibly occurring in the expression resulting from step (i), with the equivalent expression $\mathfrak{u}$.

Fixed a permutation of $R(a)$ for each $a \in A_P$, the linear program $\widehat{P} \in$ abstract(P, V, R) is then obtained from P by replacing all the expressions e occurring in P with $\widehat{e}$, and then by replacing each assignment of the form $x \texttt{=} e\,\texttt{;}$ with the skip statement ($\texttt{;}$) if $x \notin V$, with $x \texttt{=} \widehat{e}\,\texttt{;}$ otherwise, and by replacing each assignment of the form $a\,[i] \texttt{=} e\,\texttt{;}$ with the (parallel) assignment

$$a^{k_1}, \ldots, a^{k_n} \texttt{=} (\widehat{i} \texttt{==} k_1 \texttt{?}\, \widehat{e} : a^{k_1}), \ldots, (\widehat{i} \texttt{==} k_n \texttt{?}\, \widehat{e} : a^{k_n})\,\texttt{;}$$

that we abbreviate with $a[i] \mathrel{\hat{=}} e;$. If $n = 0$ (i.e. if $R(a) = \emptyset$), the assignment above reduces to a skip (;) statement.

If θ and θ' are two permutations of $R(a)$,then $\overline{\omega}(\text{abs}(a[e],\theta))=\overline{\omega}(\text{abs}(a[e],\theta'))$. From this it readily follows that all programs in abstract(P, V, R) are semantically equivalent.

It is worth noticing that computing an abstraction of a program P w.r.t. V and R can be done in time linear in the size of P and the cardinality of R, where by cardinality of a mapping R we mean the cardinality of the set $\bigcup_{a \in A_P} R(a)$. This contrasts with approaches based on predicate abstraction, where theorem provers [3] or SAT solvers [7] are needed to compute abstractions.

We now show that the abstraction defined above is conservative (i.e. sound), namely that every node reachable in the concrete program P is also reachable in the abstract program $\widehat{P}$. The *abstraction of ω w.r.t. R and V* is the valuation $\widehat{\omega}$ over $\widehat{V}$ such that $\widehat{\omega}(v) = \omega(v)$ for all $v \in V$ and $\widehat{\omega}(a^k) = \omega(a)(k)$ for all $a \in A_P$ and $k \in R(a)$. The following result states the relation between abstract and concrete valuations on linear and Boolean expressions.

Lemma 1. *The following facts hold:*

1. *if ω is a valuation over V_P and A_P, then $\overline{\omega}(e) \subseteq \overline{\widehat{\omega}}(\widehat{e})$, for every expression e;*
2. *if ω is a valuation over V_P and A_P and $\widehat{\omega}$ is a valuation over $V_P \cup \{a^k : a \in A_P, k \in R_P(a)\}$, then $\overline{\omega}(e) = \overline{\widehat{\omega}}(\widehat{e})$, for every expression e.*

The first statement of the lemma ensures that when a concrete expression e is abstracted to its corresponding abstract expression $\widehat{e}$ all concrete values compatible with the concrete valuations are preserved by the abstract ones. This is the key property in order to prove that the abstraction is conservative. The second statement of the lemma guarantees the equivalence of the concrete and abstract semantics when the abstraction is relative to all the variables V_P and all the array indexes R_P of the concrete program.

Let S_P be the set of states of program P. The abstraction of valuations is lifted to abstraction of states by means of the function $h : S_P \to S_{\widehat{P}}$ such that $h(\langle i, \omega \rangle) = \langle i, \widehat{\omega} \rangle$ for all $\langle i, \omega \rangle \in S_P$.

Let $S \subseteq S_P$, we define the abstraction $\alpha[h](S) = \{h(s) : s \in S\}$; conversely, for all $\widehat{S} \subseteq S_{\widehat{P}}$, the concretisation is defined by $\gamma[h](\widehat{S}) = \{s \in S_P : h(s) \in \widehat{S}\}$. It can be proved that the pair $\langle \alpha[h], \gamma[h] \rangle$ forms a Galois connection.

We define $R \subseteq R'$ if and only if $R(a) \subseteq R'(a)$ for all $a \in A_P$. We define $\langle V, R \rangle \preceq \langle V', R' \rangle$ if and only if $V \subseteq V'$ and $R \subseteq R'$. Moreover we define $\langle V, R \rangle \prec \langle V', R' \rangle$ if and only if $\langle V, R \rangle \preceq \langle V', R' \rangle$, and $V' \neq V$ or $R' \neq R$. If P and P' are programs with the same control-flow graph, then we define $P' \sqsubseteq P$ if and only if traces$(P') \subseteq$ traces(P), $P' \sqsubset P$ if and only if traces$(P') \subset$ traces(P), and $P' \equiv P$ if and only if $P' \sqsubseteq P$ and $P \sqsubseteq P'$, i.e. traces$(P') =$ traces(P).

Let S be a set of states S of a program P. We define post$_P^*(S) = \{s' : s \longrightarrow_P^* s'$ and $s \in S\}$, i.e. the set of states reachable from S. We can now state the soundness of the abstraction.

Theorem 1 (Soundness). *Let* $\langle V, R \rangle \preceq \langle V_P, R_P \rangle$ *and* $\widehat{P} \in \mathrm{abstract}(P, V, R)$. *Then,* $\mathrm{post}_P^* \subseteq (\gamma[h] \circ \mathrm{post}_{\widehat{P}}^* \circ \alpha[h])$ *and* $P \sqsubseteq \widehat{P}$. *Moreover if* $\widehat{P} \in \mathrm{abstract}(P, V_P, R_P)$ *then* $\mathrm{post}_P^* = (\gamma[h] \circ \mathrm{post}_{\widehat{P}}^* \circ \alpha[h])$ *and* $P \equiv \widehat{P}$.

The following result is key to prove the completeness of the AR procedure.

Theorem 2. *Let* $\widehat{P} \in \mathrm{abstract}(P, V, R)$ *and* $\widehat{P}' \in \mathrm{abstract}(P, V', R')$. *If* $\langle V, R \rangle \preceq \langle V', R' \rangle$, *then* $\widehat{P}' \sqsubseteq \widehat{P}$.

5   Checking Trace Feasibility

We now turn our attention to the problem of determining whether a trace $\tau = i_0 \cdots i_n$ of the abstract program $\widehat{P}$ is also a trace of the corresponding concrete program P. We show how this problem can be reduced to the problem of determining the satisfiability of a set of quantifier-free formulae (henceforth called *trace formulae*) in the decidable theory resulting from the combination of Linear Arithmetic and the theory of arrays. By Linear Arithmetic we mean standard arithmetic (over $\mathcal{D}$) with addition (i.e. $+$) and the usual relational operators (e.g. $=, <, \leq, >, \geq$) but without multiplication. (Multiplication by a constant, say $n * x$ where n is a numeral, is usually allowed but it is just a notational shorthand for the (linear) expression $x + \cdots + x$ with n occurrences of the variable x.). The theory of arrays we consider models arrays as data structures representing arbitrary associations of elements to a set of indexes. Unlike arrays available in standard programming languages, the arrays modelled by the theory of arrays need not have finite size. Let INDEX, ELEM and ARRAY be sorts for indexes, elements, and arrays (resp.), and select : ARRAY × INDEX → ELEM and store : ARRAY × INDEX × ELEM → ARRAY be function symbols. We also assume that the language of the theory of arrays includes a conditional term constructor that allows for terms of the form $(w \ ? \ t_1 : t_2)$, for every formula w and terms t_1 and t_2. Then the following is a concise presentation of the theory of arrays:

$$\forall a, i, j, e. \ \mathrm{select}(\mathrm{store}(a, i, e), j) = (j = i \ ? \ e : \mathrm{select}(a, j)) \tag{1}$$

In the sequel we will denote Linear Arithmetic with $\mathcal{T}_0$ and the union of Linear Arithmetic with the theory of arrays with $\mathcal{T}_1$.

Let $s_{i_1} \cdots s_{i_n}$ the sequence of statements associated with $\tau = i_0 \cdots i_n$. The sequence of statements is put in Single Assignment Form [9], i.e. the program variables are renamed in such a way that each variable is assigned exactly once in the resulting program. This is done in the following way. Let v be a program variable and i a program location. We define $\alpha(v, i)$ to be the number of assignments made to v prior to location i. Let e be a program expression. With $\varrho(e)$ we denote the expression obtained from e by substituting every variable v in e with $v_{\alpha(v,i)}$. Every assignment to a variable x at a given location i, say $x = e;$, is replaced by $x_{\alpha(x,i)+1} = \varrho(e);$. Every assignment to an array element, say $a[e_1] = e_2;$, is replaced by $a_{\alpha(a,i)+1}[\varrho(e_1)] = \varrho(e_2);$. Every condition c (also called *guard*) is replaced by $\varrho(c)$.

The set of *trace formulae for τ w.r.t.* P is the set of quantifier-free formulae $\Phi(\tau, P) = \bigcup_{k=1}^{n} \phi(s_{i_k})$, where $\phi(s_{i_k})$ is defined in Table 3. We define $\Phi_{\mathcal{T}_i}(\tau, P) = \mathcal{T}_i \cup \Phi(\tau, P)$ for $i = 0, 1$. The following theorem holds:

Table 3. Encoding

s_{i_k}	$\phi(s_{i_k})$	condition
`if(c)`, `assert(c);`, `while(c);`	$\{c\}$	if $i_{k+1} = \mathrm{Tsucc}_P(i_k)$
`if(c)`, `assert(c);`, `while(c);`	$\{\neg c\}$	if $i_{k+1} = \mathrm{Fsucc}_P(i_k)$
`v`$_{j+1}$ `= e;`	$\{v_{j+1} = e'\}$	
`a`$_{j+1}$`[e`$_1$`] = e`$_2$`;`	$\{a_{j+1} = \mathrm{store}(a_j, e'_1, e'_2)\}$	
`;`	$\emptyset$	

Theorem 3. *Let P_0 be a linear program and let P_1 be a linear program with arrays, then $\tau \in \mathrm{traces}(P_i)$ if and only if $\Phi_{\mathcal{T}_i}(\tau, P_i)$ is satisfiable, for $i = 0, 1$.*

In the AR procedure of Figure 1 the task of checking whether the trace for $\widehat{P}$ found by the model-checker is also a trace for P is jointly carried out by the functions encode and decide. If the variable *Trace* is set to τ, then the function call encode(*Trace*, P) at line 5 computes and returns the set of trace formulae $\Phi(\tau, P)$. If the variable *Formula* is set to the set of trace formulae $\Phi(\tau, P)$, then the function call decide(*Formula*) at line 6 checks the satisfiability of $\Phi_{\mathcal{T}_1}(\tau, P)$.

If $\Phi_{\mathcal{T}_1}(\tau, P)$ is unsatisfiable, then decide(*Formula*) returns a proof of this fact in a sequent calculus for first order logic with equality, i.e. it returns a proof of the sequent $\Phi_{\mathcal{T}_1}(\tau, P) \vdash \bot$, namely a tree whose root is labelled by the sequent $\Phi_{\mathcal{T}_1}(\tau, P) \vdash \bot$ and whose leaves are labelled by sequents of the form $\Phi_{\mathcal{T}_1}(\tau, P) \vdash \varphi$ with $\varphi \in \Phi_{\mathcal{T}_1}(\tau, P)$.

6   Refinement

Let $\widehat{P} \in \mathrm{abstract}(P, V, R)$ with $V \subseteq V_P$ and $R \subseteq R_P$, let τ be a trace of $\widehat{P}$ such that $\Phi_{\mathcal{T}_1}(\tau, P)$ is unsatisfiable and let Π be a proof of $\Phi_{\mathcal{T}_1}(\tau, P) \vdash \bot$. The procedure refine($\tau, \Pi, V, R$) computes V' and R' such that $\langle V, R \rangle \prec \langle V', R' \rangle$ and $\tau \notin \mathrm{traces}(\widehat{P}')$ for all $\widehat{P}' \in \mathrm{abstract}(P, V', R')$. From this is it is easy to conclude that $\widehat{P}' \sqsubset \widehat{P}$. This fact, which is formally stated and proved below, is key to establish the completeness of the AR procedure of Figure 1.

The definition of the procedure refine(τ, Π, V, R) is given in Figure 3. The procedure exploits the fact that every term of the form select(a, e) occurring in Π has a corresponding representation in the abstract program $\widehat{P}'$ only if $e = k$ for $k \in R'(a)$. The set V' is obtained by extending V with all the program variables occurring in Π. The computation of R' is based on the idea of turning Π into a proof of the unsatisfiability of $\Phi_{\mathcal{T}_0}(\tau, \widehat{P}')$. This is done in step 1 by adding to the premises of each leaf sequent $\Phi_{\mathcal{T}_1}(\tau, P) \vdash \varphi$ of Π a formula $Q(e, a)$ for each term of the form select(a_k, e) occurring in φ. Informally a formula of the form $Q(e, a)$ is a placeholder for the formula $\bigvee_{k \in R'(a)} e = k$. However, since $R'(a)$ is unknown

procedure refine(τ, Π, V, R)

1. $V' \leftarrow V \cup \{x \in V_P : x_j$ occurs in Π for some $j \geq 0\}$;
2. $\Pi' \leftarrow$ the sequent tree obtained from Π by
   (a) replacing every leaf sequent $\Phi_{\mathcal{T}_1}(\tau, P) \vdash \varphi$ with $\Phi_{\mathcal{T}_1}(\tau, P), \{Q(e, a) \;:\; \mathrm{select}(a, e)$ occurs in $\varphi\} \vdash \varphi$, where Q is a newly introduced binary predicate symbol added to the signature of $\mathcal{T}_1$ and
   (b) updating the sequents associated with the non-leaves nodes of the proof by re-applying all the inference rules.
3. Let $\Phi_{\mathcal{T}_1}(\tau, P), Q(e'_1, a), \ldots, Q(e'_q, a) \vdash \bot$ be the root node of Π'. Choose an R' such that $R \subseteq R'$ and

$$\Phi_{\mathcal{T}_0}(\tau, \widehat{P}') \models \bigvee_{k \in R'(a)} e'_j = k \qquad (2)$$

   for all $\widehat{P}' = \mathrm{abstract}(P, V', R')$ and $j = 1, \ldots, q$.
4. **return** $\langle V', R' \rangle$

Fig. 3. The refinement procedure

at this stage, we use $Q(e, a)$ in place of its expanded version $\bigvee_{k \in R'(a)} e = k$. The sequent tree obtained in this way is then updated by re-applying all the inference rules of Π on the new leaf sequents. This leaves us with a sequent tree Π' whose root sequent is of the form $\Phi_{\mathcal{T}_1}(\tau, P), Q(e'_1, a_{k_1}), \ldots, Q(e'_q, a_{k_q}) \vdash \bot$, where $a_{k_i} \in A_P$ for $i = 1, \ldots, q$. We are then left with the problem of defining R' in such a way that (2) holds. This is the task of step 2 of the procedure. Notice that (2) always admits $R' = R_P$ as (trivial) solution. However, as the size of $\widehat{P}'$ grows (linearly) with the cardinality of R', we are interested in finding a solution R' with the smallest possible cardinality. Since this problem is intractable in the general case, an alternative approach which works well in practice is to choose R' in such a way that $R'(a_j) = R(a_j) \cup \{e'_i : k_i = j$ and $i = 1, \ldots, q\}$ if $e'_1, \ldots, e'_q$ are all numerals and $R' = R_P$ otherwise.

The following result states that if V' and R' are computed as described above, then the sequent tree Π' can be turned into a proof of the unsatisfiability of $\Phi_{\mathcal{T}_0}(\tau, \widehat{P}')$. From this it readily follows the unsatisfiability of $\Phi_{\mathcal{T}_0}(\tau, \widehat{P}')$.

Lemma 2. *The sequent tree Π' computed at step 2 of the* refine(τ, Π, V, R) *procedure of Figure 3 can be transformed into a proof of the unsatisfiability of $\Phi_{\mathcal{T}_0}(\tau, \widehat{P}')$ for all $\widehat{P}' \in \mathrm{abstract}(P, V', R')$. Hence $\Phi_{\mathcal{T}_0}(\tau, \widehat{P}')$ is unsatisfiable for all $\widehat{P}' \in \mathrm{abstract}(P, V', R')$.*

Example 1. If τ is the trace of Table 2 relative to the program P of Table 1, then $\Phi_{\mathcal{T}_1}(\tau, P)$ comprises the set of formulae in the rightmost column of Table 2. The sequent tree corresponding to a proof of the unsatisfiability of $\Phi_{\mathcal{T}_1}(\tau, P)$ after applying step 2 of the refine procedure and omitting the formulae $\Phi_{\mathcal{T}_1}(\tau, P)$ in the left hand sides of the sequents is as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{Q(i_2,a) \vdash \text{select}(a_2,i_2) \neq 9 \quad \vdash i_2 = i_1 + 1}{Q(i_1+1,a) \vdash \text{select}(a_2,i_1+1) \neq 9 \qquad \vdash a_2 = \text{store}(a_1,i_1,2 * i_1)}}{Q(i_1+1,a) \vdash \text{select}(\text{store}(a_1,i_1,2 * i_1),i_1+1) \neq 9 \qquad (1)}}{Q(i_1+1,a) \vdash (i_1+1 = i_1 \; ? \; 2 * i_1 : \text{select}(a_1,i_1+1)) \neq 9}}{Q(i_1+1,a) \vdash \text{select}(a_1,i_1+1) \neq 9 \qquad \qquad \vdash i_1 = 0}}{Q(1,a) \vdash \text{select}(a_1,1) \neq 9 \qquad \qquad \vdash a_1 = \text{store}(a_0,1,9)}}{Q(1,a) \vdash \text{select}(\text{store}(a_0,1,9),1) \neq 9 \qquad (1)}}{Q(1,a) \vdash (1 = 1 \; ? \; 9 : \text{select}(a_0,1)) \neq 9}}{Q(1,a) \vdash \bot}$$

In this case—as we anticipated in Section 2—it thus suffices to refine the program using an R' such that $R'(a) = \{1\}$.

The following result states the key properties of the refinement process: if $\widehat{P}' \in \text{abstract}(P, V', R')$, where V' and R' are the sets of variables returned by the procedure $\text{refine}(\tau, \Pi, V, R)$, then $\widehat{P}'$ is a refinement of $\widehat{P}$ and $\langle V, R \rangle \prec \langle V', R' \rangle \preceq \langle V_P, R_P \rangle$.

Theorem 4. *Let* $\widehat{P} \in \text{abstract}(P, V, R)$, $\tau \in \text{traces}(\widehat{P})$ *such that* $\Phi_{\mathcal{T}_1}(\tau, P)$ *is unsatisfiable,* Π *be a proof of* $\Phi_{\mathcal{T}_1}(\tau, P) \vdash \bot$ *and* $\widehat{P}' \in \text{abstract}(P, V', R')$, *where* V' *and* R' *are the sets of variables returned by the procedure* $\text{refine}(\tau, \Pi, V, R)$. *Then* $\widehat{P}' \sqsubset \widehat{P}$ *and* $\langle V, R \rangle \prec \langle V', R' \rangle \preceq \langle V_P, R_P \rangle$.

We are now in a position to prove the soundness and (relative) completeness of the AR procedure.

Corollary 1 (Soundness). *Let* $V \subseteq V_P$ *and* $R \subseteq R_P$. *If* $\text{AR}(P, V, R)$ *returns* SAFE, *then* P *has no error trace.*

Corollary 2 (Relative Completeness). *Let* $V \subseteq V_P$ *and* $R \subseteq R_P$. *If* P *has no error trace and all the calls to the* model-check *procedure terminate, then* $\text{AR}(P, V, R)$ *terminates and returns* SAFE.

7 Implementation and Experimental Results

We have developed a prototype implementation of the techniques described in this paper in the EUREKA tool [1,2]. The EUREKA tool itself has been completely re-engineered: now it features *(i)* a new implementation of our model checking procedure for linear programs based on the Parma Polyhedra Library [11] for handling linear arithmetic constraints in an efficient way and *(ii)* a tight interface with CVC Lite [12] which is used as a decision procedure for the combination of linear arithmetic and the theory of arrays.

We have tested the abstraction refinement procedure described in this paper by running EUREKA against a variety of C programs featuring a non trivial interplay between array manipulation and arithmetic.[2] The programs we considered are implementations of well-known algorithms for string manipulation such as

---

[2] The current version of our tool checks for reachability properties and/or assertion violations.

string-copy, for coding (i.e. the (n, k)-Gray code, a generalisation of the binary Gray code [13]), and sorting (namely bubble sort, selection sort, and the partitioning phase of the quick sort). The string copy algorithm copies an array of `char` into another and checks that the '\0' character is eventually reached in the source array. The (n, k)-Gray code algorithm is a significantly more complex benchmark as it involves the simultaneous manipulation of four arrays and four loops, two of which are nested. The (n, k)-Gray code encodes integers using n different values and k digits (the length of the code). Adjacent elements of a (n, k)-Gray code differ in only one digit and the difference is either $+1$ or -1. (An assertion in the source code verifies this property.) The partitioning of the quick sort involves a single array and four loops, two of which—as in the Gray code algorithm—are nested. Finally, both sorting algorithms we considered involve a single array, two nested loops, and a third loop that checks the correctness through a sequence of assertions.

For each algorithm considered we have automatically generated a family of programs parametric in a positive integer N such that the size of the arrays occurring in the programs and/or the number of iterations carried out by the loops increase as N increases. Thus the higher is the value of N, the bigger is the search space to be analysed. This has allowed us to assess quantitatively the scalability of the tools we experimented with.

Besides EUREKA we have also run BLAST, SATABS, and CBMC on our benchmark programs. All the experiments have been carried out by using a 2.4GHz Pentium IV, running Linux with memory limit set to 800MB and time limit set to 30 minutes. The results of our experiments are summarised in Table 4. EUREKA performs very well on the string copy program: it scales up smoothly to program instances with arrays comprising hundreds of elements, the reason lying in that only a single element of the arrays out of the $2N$ is introduced by the refinement step (namely the one involved by the property, that is, the '\0' character). EUREKA is also able to analyse all the instances of the Gray code

Table 4. A summary of the results of our experiments. Every element of the table shows the greatest instance the tools are able to analyse and, in brackets, the time in seconds. The EUREKA column also shows the number of array elements found during the refinement and the sum of the sizes of the arrays involved in the programs. Numbers with [*] indicate that the tool can analyse greater instances than the one shown. Numbers with [1] and [2] are obtained by enabling interpolants with option `-craig 1` (that eliminates variables not in scope) and `-craig 2` (that applies a precise analysis) respectively. The choices were made in order to obtain the best results from the tool.

Benchmark	EUREKA			BLAST		SATABS		CBMC	
	Inst.	(Time)	refined/total array elements	Inst.	(Time)	Inst.	(Time)	Inst.	(Time)
String copy	1000[*]	(153.78)	1/2N	Incorrect[2]		10	(144.69)	221	(32.5)
Gray code	25	(230.26)	16/28	Incorrect[2]		Inconclusive		48	(83.65)
Partition	40	(178.02)	1/N	Incorrect[2]		Inconclusive		7	(157.14)
Bubble sort	8	(91.92)	N/N	Incorrect[1]		2	(30.42)	12	(1213.18)
Selection sort	6	(104.42)	N/N	Incorrect[2]		2	(115.86)	6	(432.60)

algorithm and of the partitioning algorithm up to $N = 25$ and $N = 40$ respectively. For the Gray code 16 out of the 28 elements of the arrays are introduced by the refinement step, whereas for the partitioning only 1 out of the N array elements suffices. The bubble sort and the selection sort algorithms proved more challenging as the largest instances EUREKA succeeded to analyse are with $N = 8$ and $N = 6$ respectively. This is due to the fact that all the N array elements are introduced by the refinement step. As a matter of fact, the assertions in the code check that every pair of adjacent elements is sorted. Therefore, every element of the array counts and needs to be modelled. In general, in the refinement step EUREKA introduces as many new *array variables* as the number of array elements required by the property to check. When all arrays are fully *expanded* then the abstraction is the most precise in the sense that the linear program is an exact approximation of the linear program with arrays.

BLAST has been used with all the recommended[3] optimisations (option `-predH 7`) and Craig interpolation [5] (options `-craig 1` and `-craig 2`) enabled. BLAST wrongly returns error traces on all benchmarks. This is due to its lack of precision in handling arrays: all the elements of an array are indistinguishable for BLAST [14].

SATABS has been used with Cadence SMV as symbolic model checker. While the default maximum number of iterations of the abstraction refinement loop is set to 50, we increased this number to 100 (option `--iterations 100`) as this was necessary in most cases for the tool to succeed.[4] SATABS handles arrays with more precision than BLAST and thus it never returns a wrong answer. However, our experiments indicate that it scales poorly on all benchmarks considered.

CBMC generates a boolean formula that encodes all the computation paths of bounded length. It therefore explicitly represents all the elements of the arrays occurring in the input program. Being a *bounded* model checker, CBMC may return incomplete results. By default CBMC adopts so-called *unwinding assertions* in order to try to automatically determine the minimum bound for a complete analysis of the input program. However, this is an undecidable problem, as shown by the fact that on three families of benchmarks out of five the tool was not able to compute the proper bound and the analysis diverged. The minimum bound had to be set by hand on the string copy, the Gray code and the partitioning benchmarks (option `--unwind`). That said, CBMC shows good scalability results wrt. BLAST and SATABS, and compares favourably with EUREKA.

In order to overcome the limitations of BLAST in handling arrays, we generated a new set of benchmarks obtained by abstracting the programs of Table 4 w.r.t. all array indexes. These are linear programs having a distinguished variable for each array element in the corresponding original programs. Since in this case the input programs do not contain arrays anymore, BLAST performs slightly

---

[3] In BLAST's user manual, `http://mtc.epfl.ch/software-tools/blast/doc`

[4] SATABS also features an option for the detection of looping counterexamples from the abstract model. This option, when enabled, heavily affected the performance of SATABS on all benchmarks, and hence we disabled it.

better by returning correct results on some benchmarks (e.g. in the string copy and in the partitioning benchmarks), but it still suffers from scalability issues as it handles very small instances (e.g. it already fails for $N = 14$ and $N = 15$ of the string copy and the partitioning algorithms resp.). We also run CBMC and SATABS on these new problems. CBMC exhibits a varied behaviour: for example, while the largest instance of the original string copy it analyses is $N = 221$ in 32.5 s. (cf. Table 4), in this case the largest instance is $N = 47$ in $241, 46$ s.. On the other hand, CBMC analyses the Bubble sort algorithm up to $N = 38$, while in the original version it fails for $N = 13$. The reason of this behaviour needs to be further investigated. SATABS fails on most benchmarks (e.g. the string copy, the bubble sort, the Gray coding, and the partitioning), reporting the inability of discovering new predicates.

In conclusion, our experiments indicate the potential of EUREKA in handling a variety of linear programs with arrays. At the same time they confirm the difficulties that the approaches based on predicate abstraction have in handling this important class of programs. The experiments also confirm that the effectiveness of our procedure does not depend on the size of the arrays manipulated by the input programs, but it depends on the number of elements introduced by the refinement step.

8   Conclusions

Most of the procedures based on predicate abstraction refinement show difficulties when dealing with arrays: either the abstractions built are too coarse, or the refinements fail to determine suitable predicates for building a new, more precise abstract version of the program. In this paper we have proposed a novel abstraction refinement scheme for software analysis that employs sets of variables and array indexes instead of predicates as done traditionally. We have showed that our approach allows for a precise and efficient analysis of a wide class of programs. Moreover, we presented a number of experimental results that indicate that a prototype implementation of our ideas compares favourably with—and on a number of programs of interest outperforms—state-of-the-art software model checkers based on predicate abstraction refinement.

References

1. Armando, A., Castellini, C., Mantovani, J.: Software model checking using linear constraints. In: ICFEM'04. Volume 3308 of LNCS., Springer (2004)
2. Armando, A., Benerecetti, M., Mantovani, J.: Model checking linear programs with arrays. In: SoftMC'05. Volume 144 of ENTCS., Elsevier (2005)
3. Ball, T., Rajamani, S.K.: Automatically validating temporal safety properties of interfaces. In: Proc. of SPIN'01, Springer New York, Inc. (2001) 103–122
4. Henzinger, T., Jhala, R., Majumdar, R., Sutre, G.: Software Verification with Blast. In: Proc. of SPIN '03. Volume 2648 of LNCS., Springer (2003) 235–239
5. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: POPL'04, New York, NY, USA, ACM Press (2004) 232–244

6. Jhala, R., McMillan, K.L.: A practical and complete approach to predicate re-
   finement. In Hermanns, H., Palsberg, J., eds.: TACAS. Volume 3920 of LNCS.,
   Springer (2006) 459–473
7. Clarke, E., Kroening, D., Sharygina, N., Yorav, K.: SATABS: SAT-based predicate
   abstraction for ANSI-C. In: TACAS'05. Volume 3440 of LNCS., Springer (2005)
   570–4
8. Kroening, D., Clarke, E., Yorav, K.: Behavioral consistency of C and Verilog
   programs using bounded model checking. In: Proc. of DAC 2003, ACM Press
   (2003) 368–371
9. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools.
   Addison-Wesley, Reading, MA (1986)
10. Reps, T., Horwitz, S., Sagiv, M.: Precise interprocedural dataflow analysis via
    graph reachability. In: Proc. of POPL '95, ACM Press (1995) 49–61
11. Bagnara, R., Ricci, E., Zaffanella, E., Hill, P.M.: Possibly not closed convex poly-
    hedra and the Parma Polyhedra Library. In Hermenegildo, M.V., Puebla, G., eds.:
    SAS'02. Volume 2477 of LNCS., Madrid, Spain, Springer (2002) 213–229
12. Barrett, C., Berezin, S.: CVC Lite: A new implementation of the cooperating
    validity checker. In: CAV. (2004) 515–518
13. Black, P.E.: Gray code, in dictionary of algorithms and data structures. See
    http://www.nist.gov/dads/HTML/graycode.html (2005)
14. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: POPL
    2002. (2002) 58–70