Software Model Checking Using Linear Constraints*

Alessandro Armando, Claudio Castellini, and Jacopo Mantovani

Artificial Intelligence Laboratory
DIST, Università degli Studi di Genova
Viale F. Causa 13, 16145 Genova, Italy
{armando,drwho,jacopo}@dist.unige.it

Abstract. Iterative abstraction refinement has emerged in the last few years as the leading approach to software model checking. In this context Boolean programs are commonly employed as simple, yet useful abstractions from conventional programming languages. In this paper we propose Linear Programs as a finer grained abstraction for sequential programs and propose a model checking procedure for this family of programs. We also present the eureka toolkit, which consists of a prototype implementation of our model checking procedure for Linear Programs as well as of a library of Linear Programs to be used for benchmarking. Experimental results obtained by running our model checker against the library provide evidence of the effectiveness of the approach.

1 Introduction

As software artifacts get increasingly complex, there is growing evidence that traditional testing techniques do not provide, alone, the level of assurance required by many applications. To overcome this difficulty, a number of model checking techniques for software have been developed in the last few years with the ultimate goal to attain the high level of automation achieved in hardware verification. However, model checking of software is a considerably more difficult problem as software systems are in most cases inherently infinite-state, and more sophisticated solutions are therefore needed. In this context, iterative (predicate) abstraction refinement has emerged as the leading approach to software model checking. Exemplary is the technique proposed in [2]: given an imperative program P as input,

Step 1 (Abstraction) the procedure computes a boolean program B having the same control-flow graph as P and whose program variables are restricted to range over the boolean values T and F. By construction, the execution traces of B are a superset of the execution traces of P.

* We are indebted to Pasquale De Lucia for his contribution to the development of a preliminary version of the model checker described in this paper.

Step 2 (Model Checking) The abstract program B is then model-checked and if the analysis of B does not reveal any undesired behaviour, then the procedure can successfully conclude that also P enjoys the same property. Otherwise an undesired behaviour of B is detected and scrutinised in order to determine whether an undesirable behaviour of P can be derived from it. If this is the case, then the procedure halts and reports this fact; otherwise,

Step 3 (Counterexample-driven Refinement) B is refined into a new boolean program with the help of a theorem prover. The new program does not exhibit the spurious execution trace detected in the previous step; then go to Step 2.

While the approach has proven very effective on specific application areas such as device drivers programming [2, 20], 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 it may be necessary to analyse.

In this paper we propose Linear Programs as an abstraction for sequential programs and propose a model checking procedure for this family of programs. Similarly to boolean programs, Linear Programs have the usual control-flow constructs and procedural abstraction with call-by-value parameter passing and recursion. Linear Programs differ from boolean programs in that program variables range over a numeric domain (e.g. the integers or the reals); moreover, all conditions and assignments to variables involve linear expressions, i.e. expressions of the form $c_0 + c_1x_1 + \cdots + c_nx_n$, where $c_0, \ldots, c_n$ are numeric constants and $x_1, \ldots, x_n$ are program variables ranging over a numeric domain. Linear Programs are considerably more expressive than boolean programs and can encode explicitly complex correlations between data and control that must necessarily be abstracted away when using boolean programs.

The model checking procedure for Linear Programs presented in this paper is built on top of the ideas presented in [1] for the simpler case of boolean programs and amounts to an extension of the inter-procedural data-flow analysis algorithm of [25]. We present the eureka toolkit, which consists of a prototype implementation of our model checking procedure for Linear Programs as well as of a library of Linear Programs. We also show the promising experimental results obtained by running eureka against the library.

2 Linear Programs

Most of the notation and concepts introduced in this Section are inspired by, or are extensions of, those presented in [1]. A Linear Program basically consists of global variables declarations and procedure definitions; a procedure definition is a sequence of statements; and a statement is either an assignment, a conditional (if/then/else), an iteration (goto/while), an assertion or a skip (;), much like in

ordinary C programs. Variables are of type `int`, and expressions can be built employing $+$, $-$ and the usual arithmetic comparison predicates $=, \neq, <, >, \leq, \geq$.

Given a Linear Program P consisting of n statements and p procedures, we assign to each statement a unique index from 1 to n, and to each procedure a unique index from $n+1$ to $n+p$. With s_i we denote the statement at index i. For the sake of simplicity, we assume that variable and label names are globally unique in P. We also assume the existence of a procedure called `main`: it is the first procedure to be executed.

We define $Globals(P)$ as the set of global variables of P; $Formals_P(i)$ is the set of formal parameters of the procedure that contains the statement s_i; $Locals_P(i)$ is the set of local variables and formal parameters of the procedure that contains the statement s_i, while $sLocals_P(i)$ is the set of strictly local variables (i.e. without the formal parameters) of the procedure that contains statement s_i; $InScope_P(i)$ is the set of variables that are in scope at statement s_i. Finally, $First_P(pr)$ is the index of the first statement of procedure pr and $ProcOf_P(i)$ the index of the procedure belonging to the statement s_i.

The Control Flow Graph. The *Control flow Graph* of a Linear Program P is a directed graph $G_P = (V_P, Succ_P)$, where $V_P = \{0, 1, \ldots, n, n+1, \ldots, n+p\}$ is the set of vertices, one for each statement (from 1 to n) and one $Exit_{pr}$ vertex for each procedure pr (from $n+1$ to $n+p$). Vertex 0 is used to model the failure of an `assert` statement. Vertex 1 is the first statement of procedure `main`: $First_P(\texttt{main}) = 1$. To get a concrete feeling of how a linear program and its control flow graph look like, the reader may refer to the program `parity.c`, given in Figure 1.

As one can see, `parity(n)` looks reasonably close to a small, real C procedure recursively calling upon itself 10 times (the actual parameter `x` is set to 10) whose task is to determine the parity of its argument by "flipping" the global variable `even`, which ends up being 1 if and only if `x` is even. Again Figure 1 gives an intuition of how the function $Next_P(i)$ and $sSucc_P(i)$ behave. Roughly speaking, given a vertex i, $sSucc_P(i)$ follows the execution trace of the program,[1] while $Next_P(i)$ is the *lexical* successor of i.

Valuations and transitions. We now informally define valuations and transitions for a Linear Program. The reader may find more formal and precise definitions in a longer version of this article [22]. Let $\mathcal{D}$ be a numerical domain, called the domain of computation. Given a vertex $i \in V_P$, a *valuation* is a function $\omega : InScope_P(i) \to \mathcal{D}$ (it can be extended to expressions inductively, in the usual way), and a *state* of a program P is a pair $\langle i, \omega \rangle$. A state $\langle i, \omega \rangle$ is *initial* if and only if $i = 1$. State transitions in a linear program P are denoted by $\langle i_k, \omega_1 \rangle \to_P^{\alpha} \langle i_{k+1}, \omega_2 \rangle$ where i_{k+1} is one of the successors of i and α is a label

[1] Conditional statements represent the only exception: there, $Tsucc_P(i)$ and $Fsucc_P(i)$ denote the successors in the *true* and *false* branch, respectively.

```
int x, even;

main(){
  x = 10;
  even = 1;
  parity(x);
  if ( even==1 ) { ; }
  else { ERROR: ; }
}

parity(n){
  int i;
  if( n==0 ){ ; }
  else {
    i = n - 1;
    even = -1 * even;
    parity(i);
  }
}
```

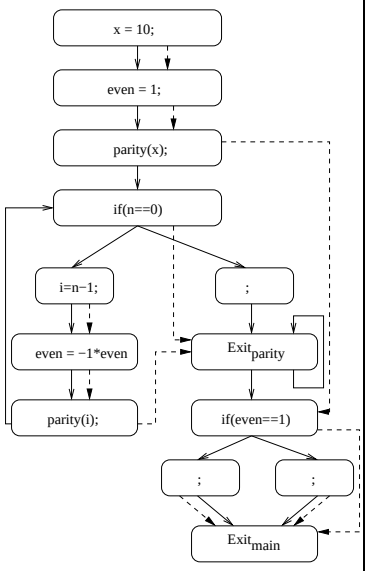

Fig. 1. `parity.c` and its control flow graph. The dashed lines show the $Next_P$ function, while the continuous lines show the successor relations between the vertices.

ranging over the set of terminals:

$$\begin{aligned}\Sigma(P) = \{\sigma\} \cup \{&\langle \texttt{call}, i, \delta\rangle, \langle \texttt{ret}, i, \delta\rangle : \exists j \in V_P \text{ s.t.} \\ &s_j = \texttt{call}, i = \mathit{Next}_P(j), \delta : \mathit{Locals}_P(j) \to \mathcal{D}\}.\end{aligned}$$

Terminals of the form $\langle \texttt{call}, i, \delta\rangle$ and $\langle \texttt{ret}, i, \delta\rangle$ represent, respectively, entry and exit points of the procedure invoked by s_j. A *path* is a sequence $\langle i_0, \omega_0\rangle \to_P^{\alpha_1} \langle i_1, \omega_1\rangle \to_P^{\alpha_2} \cdots \to_P^{\alpha_n} \langle i_n, \omega_n\rangle$ such that $\langle i_k, \omega_k\rangle \to_P^{\alpha_{k+1}} \langle i_{k+1}, \omega_{k+1}\rangle$ for $k = 0, \ldots, n-1$. Notice that not all paths represent potential execution paths: in a transition like $\langle \mathit{Exit}_{pr}, \omega_1\rangle \to_P^{\langle \texttt{ret}, i_2, \delta\rangle} \langle i_2, \omega_2\rangle$, the valuation δ 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. To rectify this, the notion of same-level valid path is introduced. 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 which may end with some number of activation records on the call stack; these correspond to "unmatched" terminals of the form $\langle \texttt{ret}, i, \delta\rangle$ in the string associated with the path. A same-level 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 $\langle i_n, \omega_n\rangle$—where $\langle i_0, \omega_0\rangle$ and $\langle i_n, \omega_n\rangle$ are in the same procedure—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, before eventually returning to its original depth. A state $\langle i, \omega\rangle$ is *reachable* iff there

exists a valid path from some initial state to $\langle i, \omega \rangle$. A vertex $i \in V_P$ is *reachable* iff there exists a valuation ω such that $\langle i, \omega \rangle$ is reachable.

3 Symbolic Model Checking of Linear Programs

The reachability of a line in a program can be reduced to computing the set of valuations Ω_i such that $\langle i, \omega \rangle$ is reachable iff $\omega \in \Omega_i$, for each vertex i in the control flow graph of the program: the statement associated to vertex i is reachable iff Ω_i is not empty. Following [1], our model checking procedure computes *(i)* "path edges" to represent the reachability status of vertices and *(ii)* "summary edges" to record the input/output behaviour of procedures.

Let $i \in V_P$ and $e = \mathit{First}_P(\mathit{ProcOf}_P(i))$. A *path edge* $\pi_i = \langle \omega_e, \omega_i \rangle$ of i is a pair of valuations such that there is a valid path $\langle 1, \omega_0 \rangle \rightarrow_P^{\alpha_1} \cdots \rightarrow_P^{\alpha_k} \langle e, \omega_e \rangle$ and a same-level valid path $\langle e, \omega_e \rangle \rightarrow_P^{\alpha_{k+1}} \cdots \rightarrow_P^{\alpha_n} \langle i, \omega_i \rangle$ for some valuation ω_0. In other words, a path edge represents a suffix of a valid path from $\langle 1, \omega_0 \rangle$ to $\langle i, \omega_i \rangle$.

Let $i \in V_P$ be such that $s_i = pr(e_1, \ldots, e_n)$, let $y_1, \ldots, y_n$ be the formal parameters of pr associated to the actuals $e_1, \ldots, e_n$ respectively, and let $\pi = \langle \omega_i, \omega_o \rangle$ be a path edge of a vertex Exit_{pr}. A *summary edge* $\sigma = \langle \omega_1, \omega_2 \rangle$ of π is a pair of valuations such that

1. $\omega_1(g) = \omega_i(g)$ and $\omega_2(g) = \omega_o(g)$ for all $g \in \mathit{Globals}(P)$, and
2. $\omega_1(y_j) = \omega_i(e_j) = \omega_o(e_j)$ for $j = 1, \ldots, n$.

The computation of the summary edges is one of the most important parts of the algorithm. Summary edges record the output valuation ω_2 of a procedure for a given input valuation ω_1. Therefore, there is no need to re-enter a procedure for the same input, since the output is known already. In some cases of frequently called procedures and of recursion, this turns into a great improvement in performance. We represent path edges and summary edges symbolically, by means of Abstract Disjunctive Linear Constraints. In the rest of this section we give the formal definitions needed and define the operations over them.

3.1 Representing path edges and summary edges symbolically.

A *linear expression* over $\mathcal{D}$ is an expression of the form $c_0 + c_1 x_1 + \ldots + c_n x_n$, where $c_0, c_1, \ldots, c_n$ are constants and $x_1, x_2, \ldots, x_n$ are variables, both ranging over $\mathcal{D}$. A *linear constraint* is a relation of the form $e \leq 0$, $e = 0$, $e \neq 0$, where e is a linear expression over $\mathcal{D}$. A *linear formula* is a boolean combination of linear constraints. A *Disjunctive Linear Constraint* D (DLC for short) is a linear formula in disjunctive normal form (that is, a disjunction of conjunctions of linear constraints). Formally, $D = \bigvee_i \bigwedge_j c_{ij}$, where c_{ij} are linear constraints. The symbol $\top$ stands for a tautological linear formula, while $\bot$ stands for an unsatisfiable linear formula. An *Abstract Disjunctive Linear Constraint* (ADLC for short) is an expression of the form $\lambda \mathbf{x} \lambda \mathbf{x}'.D$, where D is a DLC, and $\mathbf{x}, \mathbf{x}'$

are all and the only variables in D.[2] The following operations over DLCs are defined:

- *Application.* Let $\lambda \mathbf{x}\mathbf{x}'.D$ be an ADLC and $\mathbf{s}$ and $\mathbf{t}$ be vectors of linear expressions with the same number of elements as $\mathbf{x}$. The *application of* $\lambda \mathbf{x}\mathbf{x}'.D$ *to* $(\mathbf{s}, \mathbf{t})$ is the DLC obtained by simultaneously replacing the i-th element of $\mathbf{x}$ ($\mathbf{x}'$) with the i-th element of $\mathbf{s}$ ($\mathbf{t}$ resp.).
- *Conjunction.* Let D_1 and D_2 be two DLCs, then $D_1 \sqcap D_2$ is any DLC for $D_1 \wedge D_2$. Conjunction is extended to ADLCs as follows. Let δ_1 and δ_2 be two ADLCs. Then, $\delta_1 \sqcap \delta_2 = \lambda \mathbf{x}\mathbf{x}'.(\delta_1(\mathbf{x}, \mathbf{x}') \sqcap \delta_2(\mathbf{x}, \mathbf{x}'))$.
- *Disjunction.* Let D_1 and D_2 be two DLCs, then $D_1 \sqcup D_2$ is $D_1 \vee D_2$. Disjunction is extended to ADLCs in the following way. Let δ_1 and δ_2 be two ADLCs. Then, $\delta_1 \sqcup \delta_2 = \lambda \mathbf{x}\mathbf{x}'.(\delta_1(\mathbf{x}, \mathbf{x}') \sqcup \delta_2(\mathbf{x}, \mathbf{x}')) = \lambda \mathbf{x}\mathbf{x}'.(\delta_1(\mathbf{x}, \mathbf{x}') \vee \delta_2(\mathbf{x}, \mathbf{x}'))$.
- *Negation.* Let D be a DLC. Then $^{\sim}D$ is obtained by putting the negation $\neg D$ of D in disjunctive normal form. Negation is extended to ADLCs in the following way: $^{\sim}\delta = \lambda \mathbf{x}\mathbf{x}'.(^{\sim}\delta(\mathbf{x}, \mathbf{x}'))$.
- *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 $\mathbf{x}$.
- *Entailment.* Let δ_1 and δ_2 be two ADLCs, $\delta_1 \sqsubseteq \delta_2$ iff all the pairs of valuations satisfying δ_1 satisfy also δ_2: $\delta_1 \sqsubseteq \delta_2$ iff $\delta_1(\mathbf{x}, \mathbf{x}') \models_{\mathcal{D}} \delta_2(\mathbf{x}, \mathbf{x}')$. With the subscript $\mathcal{D}$ in $\models_{\mathcal{D}}$ we denote that assignments over variables range over $\mathcal{D}$ and that the symbols $+$, $-$, $*$, $=$, $\neq$, $\leq$ have the intended interpretation.

Summary Edges. Let c be a vertex of V_P for a procedure call, say $s_c = pr(\mathbf{a})$, and let i be the exit vertex of pr. Let $\mathbf{y} = Formals_P(i)$, $\mathbf{x} = InScope_P(i)$, $\mathbf{z} = sLocals_P(i)$, $\mathbf{g} = Globals(P)$; then $Lift_c(\delta) = \lambda \mathbf{g}\mathbf{g}'\mathbf{y}.\exists \mathbf{z}\mathbf{z}'\mathbf{y}'.\delta(\mathbf{x}, \mathbf{x}')$. A summary edge of a procedure pr records the "behaviour" of pr in terms of the values of the global variables just before the call (i.e. $\mathbf{g}$) and after the execution of pr (i.e. $\mathbf{g}'$). These valuations depend on the valuations of the formal parameters (i.e. $\mathbf{y}$) of pr.

Transition Relations. From here on, we will use the expression $\mathbf{x}' = \mathbf{x}$ as a shorthand for $x'_1 = x_1 \wedge \ldots \wedge x'_n = x_n$. Let $Exit_P$ be the set of exit vertices in V_P, $Cond_P$ be the set of conditional statements in V_P, and $Call_P$ be the set of all procedure calls in V_P. We associate with each vertex i of $V_P \setminus Exit_P$ a transfer function, defined as follows: if s_i is a `;` or a `goto` statement, then $\tau_i = \lambda \mathbf{x}\mathbf{x}'.(\mathbf{x}' = \mathbf{x})$ where $\mathbf{x} = InScope_P(i)$; if s_i is an assignment of the form **y=e**, then $\tau_i = \lambda \mathbf{x}\mathbf{x}'\lambda \mathbf{y}\mathbf{y}'.(\mathbf{y}' = \mathbf{e} \wedge \mathbf{x} = \mathbf{x}')$ where $\mathbf{x} = InScope_P(i) \setminus \mathbf{y}$; if $i \in Call_P$, i.e. s_i is of the form pr**(a)**, then $\tau_i = \lambda \mathbf{y}'\mathbf{g}\mathbf{g}'.\exists \mathbf{y}\mathbf{z}\mathbf{z}'.(\mathbf{y}' = \mathbf{a} \wedge \mathbf{x} = \mathbf{x}')$ where $\mathbf{y} = Formals_P(First_P(pr))$, $\mathbf{z} = Locals_P(i)$, $\mathbf{x} = InScope_P(i)$ and $\mathbf{g} = Globals(P)$; finally, if $i \in Cond_P$, that is, s_i is of the form `if`$(d(\mathbf{x}))$, `while`$(d(\mathbf{x}))$ or `assert`$(d(\mathbf{x}))$, then $\tau_{i,true} = \lambda \mathbf{x}\mathbf{x}'.(d(\mathbf{x}') \sqcap \mathbf{x}' = \mathbf{x})$, and $\tau_{i,false} = \lambda \mathbf{x}\mathbf{x}'.((^{\sim}d(\mathbf{x}')) \sqcap \mathbf{x}' = \mathbf{x})$, where $\mathbf{x} = InScope_P(i)$.

[2] For sake of simplicity, in the rest of the paper we will write $\lambda \mathbf{x}\mathbf{x}'.D$ instead of $\lambda \mathbf{x} \lambda \mathbf{x}'.D$.

The Join Functions. We now define the *Join* and the *SEJoin* functions. The first one applies to a path edge the given transition relation for a vertex, while the second one is used only during procedure calls, adopting the summary edge as a (partial) transition relation.

Let δ be an ADLC representing the path edges for a given vertex i and let τ be an ADLC representing the transition relation associated with i; then $\mathit{Join}(\delta, \tau)$ computes and returns the ADLC representing the path edges obtained by extending the path edges represented by δ with the transitions represented by τ. Formally, $\mathit{Join}(\delta, \tau) = \lambda \mathbf{x}\mathbf{x}'.\exists \mathbf{x}''.(\delta(\mathbf{x}, \mathbf{x}'') \sqcap \tau(\mathbf{x}'', \mathbf{x}'))$.

Let δ be an ADLC representing the path edges for $s_i = pr(\mathbf{a})$ and let σ be an ADLC representing the summary edges for pr. The *Join* operation between δ and σ is defined as follows. Let $\mathbf{g} = \mathit{Globals(P)}$, $\mathbf{y} = \mathit{Formals}_P(\mathit{First}_P(pr))$, $\mathbf{z} = \mathit{Locals}_P(i)$; then $\mathit{SEJoin}(\delta, \sigma) = \lambda \mathbf{g}\mathbf{g}'\mathbf{z}\mathbf{z}'.\exists \mathbf{g}''.(\exists \mathbf{y}'.(\sigma(\mathbf{g}', \mathbf{g}'', \mathbf{y}') \sqcap \mathbf{a}' = \mathbf{y}')) \sqcap \delta(\mathbf{g}, \mathbf{g}'', \mathbf{z}, \mathbf{z}')$.

Self Loops. $\mathit{SelfLoop}(\delta)$ is used to model the semantics of procedure calls, by making self loops with the target of the path edges. Formally, given an ADLC δ, $\mathit{SelfLoop}(\delta) = \lambda \mathbf{x}\mathbf{x}'.\exists \mathbf{x}''.(\delta(\mathbf{x}, \mathbf{x}'') \sqcap \mathbf{x}'' = \mathbf{x}')$.

3.2 The Model Checking Procedure

The Model Checking procedure works by incrementally storing in a worklist the next statement to be analyzed, by computing path edges and summary edges accordingly, and by removing the processed statement from the worklist. Let W be the worklist, P be an array of $n + p + 1$ ADLCs, and let S be an array of p ADLCs[3]. P collects the path edges of each vertex of the control flow graph, and S collects the summary edges of each procedure.

We define the initial state of our procedure as a triple $\left(1.\epsilon, P^1, S^1\right)$, where $P_1^1 = \lambda \mathbf{x}\mathbf{x}'.(\mathbf{x} = \mathbf{x}')$ with $\mathbf{x} = \mathit{InScope}_P(1)$ and $P_i^1 = \lambda \mathbf{y}\mathbf{y}'.\bot$ for each $0 \leq i \leq (n + p)$ such that $i \neq 1$, and $S_j^1 = \lambda \mathbf{y}\mathbf{y}'.\bot$ for each $1 \leq j \leq p$.

We also need to define a function that updates the array P of the path edges and the worklist W, when needed. Given a vertex j (the next to be verified), the current worklist $W = i.is$, an ADLC D (the one just computed for vertex i), and the array P of path edges, the function returns the pair containing the updated worklist W' and the updated array P' of path edges, according to the result of the entailment check $D \sqsubseteq P_j$. We refer to this function as the "propagation" function *prop*.[4]

$$\mathit{prop}(j, i.is, D, P) = \begin{cases} (\mathit{Insert}(j, is), P\,[(D \sqcup P_j)/j]) & \text{if } D \not\sqsubseteq P_j \\ (is, P) & \text{otherwise.} \end{cases}$$

[3] We recall that n is the number of statements, and p is the number of procedures. With P_i and S_i we denote the i-th element of the arrays.

[4] The function $\mathit{Insert}(\mathit{El}, \mathit{List})$ returns a new list in which El is added to List in a non-deterministically chosen position.

A generic transition of the procedure is of the form $(W, P, S) \rightarrow (W', P', S')$, where the values of W', P', S' depend on the vertex being valuated, that is, on the vertex on top of the worklist. Let i be the vertex on top of W, that is, $W = i.is$. The procedure evolves according to the following cases.

Procedure Calls. Let $s_i = pr(\mathbf{a})$, $p = Exit_{pr}$, $l_i = SelfLoop(Join(P_i, \tau_i))$, and $r_i = SEJoin(P_i, S_p)$. In this case the procedure is entered only if no summary edge has been built for the given input. This happens if and only if l_i does not entail $P_{sSucc_P(i)}$; otherwise the procedure is skipped, using r_i as a (partial) transition relation. The call stack is updated only if the procedure is entered. If $l_i \not\sqsubseteq P_{sSucc_P(i)}$, then

$$\begin{aligned} W' &= Insert(sSucc_P(i), is), \\ P' &= P[(P_{sSucc_P(i)} \sqcup l_i)/sSucc_P(i)], \\ S' &= S. \end{aligned}$$

Otherwise, $(W', P') = prop(Next_P(i), W, r_i, P)$ and $S' = S$.

Return from Procedure Calls. Let $i \in Exit_P$. When an exit node is reached, a new summary edge s is built for the procedure. If it entails the old summary edges S_i, then $S' = S$; otherwise s is added to S_i. For each $w \in Succ_P(i)$ let $c \in Call_P$ such that $w = Next_P(c)$, and let $s = Lift_{\mathbf{c}}(P_i)$. If $s \not\sqsubseteq S_i$ then for each w,

$$\begin{aligned} (W', P') &= prop(w, W, SEJoin(P_i, S_i \sqcup s), P), \\ S' &= S\left[(S_i \sqcup s)/i\right]. \end{aligned}$$

Otherwise, $W' = W$, $P' = P$, and $S' = S$.

Conditional Statements. Both of the branches of the conditional statements have to be analyzed. Therefore, the worklist and the array of path edges have to be updated twice, according to the propagation function. Let $i \in Cond_P$, then

$$\begin{aligned} (W'', P'') &= prop(Fsucc_P(i), W, Join(P_i, \tau_{i,false}), P), \\ (W', P') &= prop(Tsucc_P(i), W'', Join(P_i, \tau_{i,true}), P''), \\ S' &= S. \end{aligned}$$

Other Statements. In all other cases the statements do not need any special treatment, and the path edges are simply propagated the single successor. If $i \in V_P \setminus Cond_P \setminus Call_P \setminus Exit_P$, then

$$\begin{aligned} (W', P') &= prop(sSucc_P(i), W, Join(P_i, \tau_i), P), \\ S' &= S. \end{aligned}$$

When no more rules are applicable, P contains the valuations of all the vertices of the control flow graph and S contains the valuations that show the behaviour of every procedure given the values of its formal parameters. As a result, if the ADLC representing a path edge for a vertex is $\bot$, then that vertex is not reachable.

4 The eureka Toolkit

We have implemented the procedure described in Section 3 in a fully automatic prototype system called eureka which, given a Linear Program as input, first generates the control flow graph and then model-checks the input program. The most important operations involved in the procedure, namely quantifier elimination and entailment check, are taken care of, respectively, by a module called QE, employing the Fourier-Motzkin method (see, e.g.,[24])[5], and ICS 2.0 [15], a system able to decide satisfiability of ground first-order formulae belonging to several theories, among which Linear Arithmetic.

In order to qualitatively test the effectiveness and scalability properties of our approach, we have devised a library of Linear Programs called *eureka Library*. As a general statement, the library consists of Linear Programs reasonably close to real C language, and easily synthesizable in different degrees of hardness; in this initial version, it is organised in six parametrized families; for each family, a non-negative number N is used to generate problems which stress one or more aspects of the computation of Linear Programs:

Data The use of arithmetic reasoning, e.g., summing, subtracting, multiplying by a constant and comparing;

Control The use of conditionals, e.g., `if`, `assert` and the condition test of `while`;

Iteration The number of iterations done during the execution;

Recursion The use of recursion.

This classification is suggested by general considerations about what kind of data- and control-intensiveness is usually found in C programs in verification. For example, the corpus analysed by SLAM, consisting of device drivers [2, 20], is highly control-intensive but not data-intensive; arithmetic reasoning is usually considered hard to reason about; and recursion is usually considered harder than iteration.

A problem is generated by instantiating the family template according to N, and a line of the resulting program, tagged with an error label, is then tested for reachability. For each family, increasing N increases the hardness of the problem by making one or more of the above aspects heavier and heavier. Table 1 gives an account of which aspects are stressed in each family. A full account of the structure of the Library is available in [22], as well as at the URL http://www.ai.dist.unige.it/eureka.

For example, `swap_seq.c`(N) consists of a simple procedure `swap()` which swaps the values of two global variables, and is called upon $2N$ times. In this case, increasing N stresses the capacity of the system to deal with arithmetic reasoning. On the other side, `swap_iter.c`(N) does basically the same, but using an iterative cycle; therefore increasing N also forces more and more iterations.

[5] the Fourier-Motzkin method works on the real numbers, whereas Linear Programs variables range over the integers; this may lead to false negatives, but the problem has not arisen in practice so far.

Table 1. The eureka Library of Linear Programs. Each family stresses on or more aspects of the computation of Linear Programs, as N grows.

Family	data	control	iteration	recursion
`swap_seq.c`(N)	$\surd$			
`swap_iter.c`(N)	$\surd$		$\surd$	
`delay_iter.c`(N)	$\surd$	$\surd$	$\surd$	
`delay_recur.c`(N)	$\surd$	$\surd$	$\surd$	$\surd$
`parity.c`(N)	$\surd$			$\surd$
`sum.c`(N)	$\surd$		$\surd$	

As another example, `parity.c`(N) is the template version of the program in figure 1; here N is the number whose parity we want to evaluate. Making N bigger involves more and more arithmetic reasoning, and it also forces the system to follow more recursive calls.

5 Experimental Results

In our experimental analysis we have tried to identify how our system scales on each family of Linear Programs in the eureka Library, as N grows, and why. An initial set of tests has revealed that, unsurprisingly, the hardest operations the system performs are *(i)* the *Join* operation, involving quantifier elimination, and *(ii)* the entailment check. Moreover, some of the programs in the library have proved to be particularly hard because of the number of redundant constraints generated during the conjunction of two ADLCs into a new ADLC, required again by *Join*. This can potentially make path edges and summary edges unmanageable.

Figure 2 shows the experimental results. The x-axis shows N for each family of programs, while the y-axis shows the total CPU time required, on a linear scale. Additionally, we show the time spent by ICS and by the QE module. All tests have been performed on Pentium IV 1.8GHz machines equipped with Linux RedHat 7.2; the timeout was set to 3600 seconds and the memory limit was set to 128MB.

A quick analysis of the results follows. First of all, let us note that in all cases except one the hardness of the problems grows monotonically with N; this indicates that, at least for our procedure, stressing the above mentioned aspects of computation actually makes the problems harder and harder. The only remarkable exception, presenting a strong oscillating behaviour, is `parity.c`(N); as one can see by looking at the graph, ICS is entirely responsible for it. Moreover, a more detailed analysis reveals that the *number of calls* to ICS grows monotonically; this means that some instances of the problems given to ICS are actually easier than those for a smaller N. We argue that this is due to the way constraints are stored in the summary edge during the calls to `parity(n)`: in some cases they ease ICS's search, whereas in others they do not.

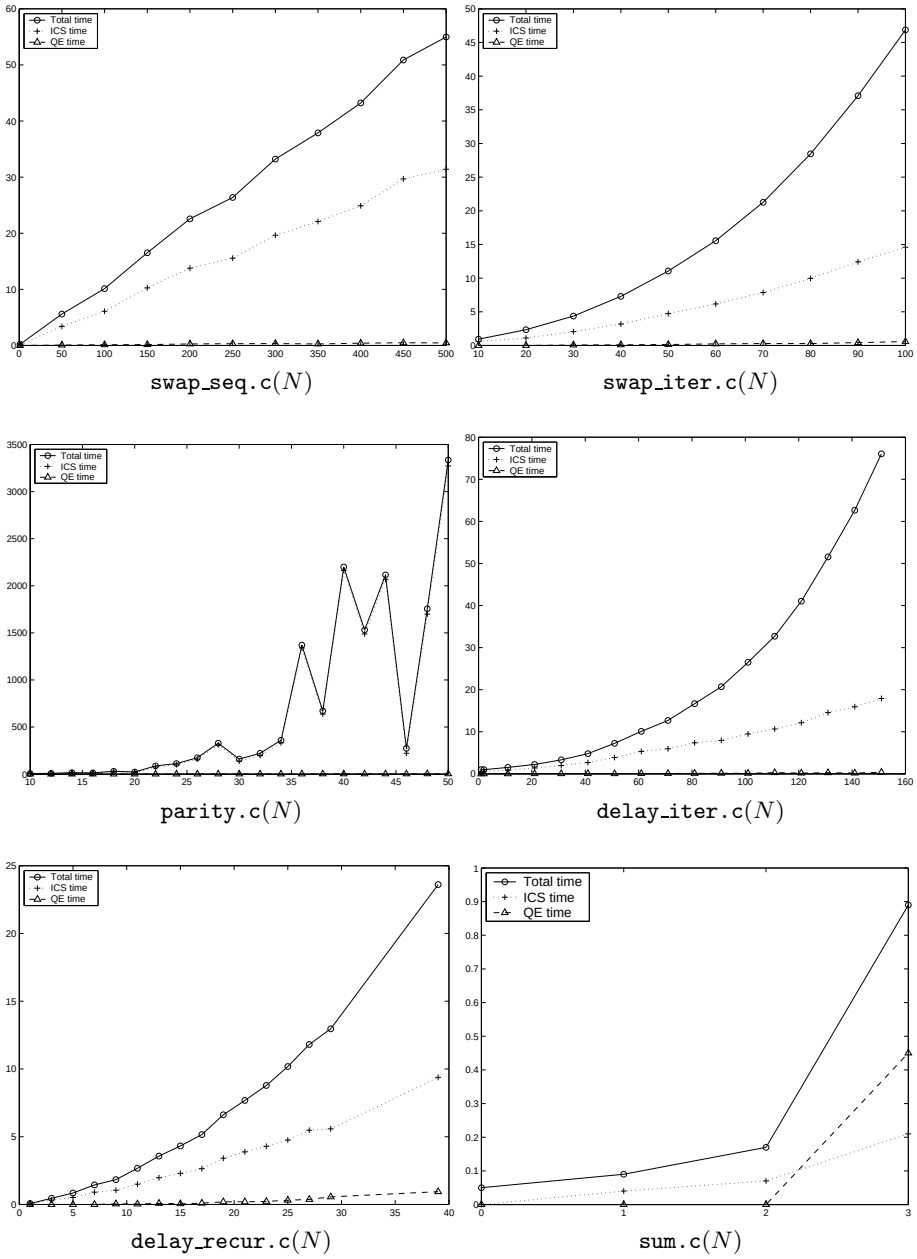

Fig. 2. Experimental results. Each graph shows the total CPU time, the time spent by ICS and the time required by the QE module, as N; each caption indicates the related program family.

Secondly, the only family whose hardness seems to grow linearly with N is `swap_seq.c`(N). This is reasonable, since once the summary edges for the swapping procedure have been calculated twice, they need not be re-calculated any longer. Quantifier elimination is irrelevant.

Thirdly, `swap_iter.c`(N), `delay_iter.c`(N) and `delay_recur.c`(N) grow more than linearly with N, with ICS time dominating quantifier elimination but accounting only for a fraction of the total time. Here the burden of conjoining ADLCs plays a relevant role.

Lastly, the graph for `sum.c`(N) shows that this family represents a particularly hard set of problems for `eureka`. In fact, already for $N = 4$, the path edges to be propagated within the `while()` statement that computes the sum become so complicated that the procedure runs out of time.

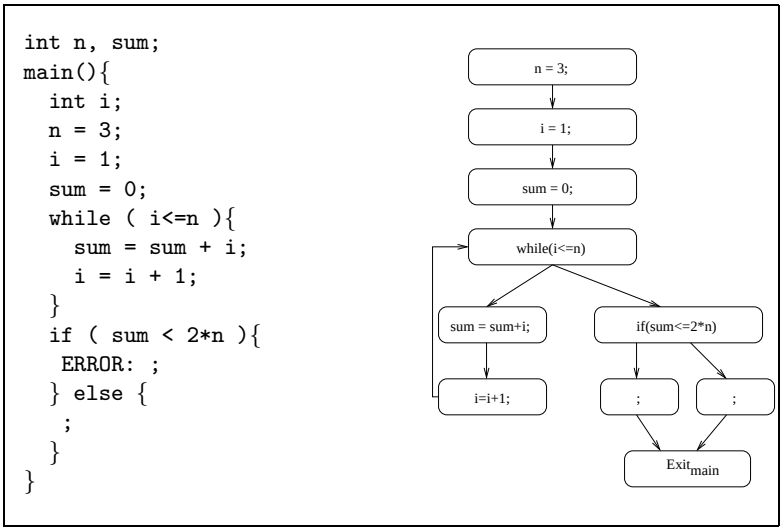

Fig. 3. `sum.c`(3) and its control flow graph.

Consider Figure 3, which shows the program for $N = 3$ and its control flow graph; the iteration condition, and the assignments inside the loop altogether, involve 3 variables (plus their primed and doubly primed counterparts, according to the model checking procedure, reaching a total of 9 variables). Subject to four iterations, at each of which no entailment is discovered, the path edges become bigger and bigger and the conjoining machinery goes timeout.

Notice that here, unlike all other examples, the quantifier elimination time dominates; still, an analysis of the failed attempt at `sum.c`(4) reveals that ICS and QE times account for less than 15% of the total time. As it was the case for, e.g., `swap_iter.c`(N), the operation of conjoining ADLCs is the bottleneck.

Summing up: it seems that quantifier elimination is *not* the hardest point of the procedure; rather, the way path and summary edges are built, that is, how ADLCs are managed, is the first source of inefficiency. In particular, we believe that a better encoding would significantly improve performance; keeping path and summary edges as compact as possible would also have a beneficial impact on ICS, since the problems it must solve could be significantly simpler.

Comparative Results. Only a few comparative tests have been possible, since one of the main systems able to tackle Linear Programs, namely SLAM [2], is not publicly available; we compared our results with the BLAST toolkit [20] instead. The input language of BLAST is a superset of that of Linear Programs, but the tool lacks support for recursion. We have run BLAST on the non-recursive programs of the eureka Library; but even for small instances (namely `swap_seq.c`(1), `swap_iter.c`(2) and `sum.c`(3)) the system stopped, reporting the infeasibility of discovering new predicates during its refinement process. Only on `delay_iter.c`(N) BLAST scales very well, exhibiting an almost constant behaviour.

6 Related Work

The Boolean *Abstraction/Refinement* paradigm is a common approach to Software Model Checking of C programs. It has been successfully applied to device drivers in the Slam toolkit [2] and in Blast [20]. The latter is a so-called "lazy" model checker, since if it finds an error in the model, it does not refine the whole abstraction that has been built, but only the fragment of code that generated the spurious error. Other model checkers for C that perform abstraction and refinement are MAGIC [7], Boop [9], and Moped [17]. Moreover, Flanagan [11] proposes both a technique and a tool that translate imperative programs to CLP. Each procedure is translated into two CLP relations, an *error* and a *transfer* relation. The program is then verified by performing a depth-first search trying to satisfy the error relation of the `main` procedure. Since the proposed tool is in a too preliminary stage for testing,[6] we plan to compare eureka with it as soon as the first release will be distributed.

FeaVer and Modex [12] provide support for the translation of C programs to input languages of verification tools like SPIN [4]. Basically they abstract a C program to a finite model that can then be verified. Two other tools do the same for Java software: Bandera [5] and Java PathFinder [6].

Various tools and approaches have been applied to programming languages other than C and to different systems. Bultan et al. [3] have developed a Model Checking procedure for concurrent systems that encodes constraints in Presburger Arithmetic and applies widening operators in order to enforce termination. The Kiss [19] toolkit is devoted to analyze concurrent software. It reduces the input program to a sequential program that simulates a large subset of behaviours of the original, concurrent one. Kiss uses the Slam toolkit as back-end.

[6] Personal communication by Cormac Flanagan.

7 Conclusions and Future Work

We have proposed Linear Programs as an alternative model for sequential programs to be used in the context of the abstraction/refinement paradigm. Linear Programs are a less abstract and more expressive model than boolean programs, and can explicitly encode complex correlations between data and control, allowing a great reduction of the inefficiencies due to the iterative refinement process. We have also described and implemented a model checking procedure for Linear Programs. The eureka prototype has been tested against the eureka library, a set of benchmark programs built to stress different aspects of computation. As shown in section 5, the experimental results are promising, since they reveal good scalability properties of our approach.

Future Work. Since the eureka model checker is a prototype tool, it can be heavily optimized, for example by adopting Gaussian elimination for resolving equalities instead of the Fourier-Motzkin method or, more radically, by replacing it with the *Omega library* [21]. Also, since the reachability problem becomes undecidable if applied to Linear Programs, we plan to cope with the non termination of our model checking procedure by investigating the use of widening techniques [14, 3]. In doing so, one has to be aware that the procedure may terminate with some false positive results. Lastly, we are also working on abstraction. Since eureka only works on Linear Programs, we argue it may be possible to build a tool able to abstract from "usual" C programs (for instance including structures, pointers and array) to Linear Programs. The improvements described above, when done, will be tested against the eureka library, that will be growing with new programs of increasing difficulty and new categories.

References

1. Ball, T., Rajamani, S.K.: Bebop: A symbolic model checker for boolean programs. In: Proc. of SPIN 2000. (2000) 113–130
2. Ball, T., Rajamani, S.K.: Automatically validating temporal safety properties of interfaces. In: Proc. of SPIN 2001, Springer-Verlag New York, Inc. (2001) 103–122
3. Bultan, T., Gerber, R., Pugh, W.: Model-checking concurrent systems with unbounded integer variables: symbolic representations, approximations, and experimental results. ACM Transactions on Programming Languages and Systems **21** (1999) 747–789
4. Holzmann, G.J.: The model checker spin. IEEE Trans. Softw. Eng. **23** (1997) 279–295
5. Corbett, J.C., Dwyer, M.B., Hatcliff, J., Laubach, S., Pasareanu, C.S., Robby, Zheng, H.: Bandera: extracting finite-state models from java source code. In: Proc. of the 22nd int. conf. on Software engineering, ACM Press (2000) 439–448
6. Visser, W., Havelund, K., Brat, G., Park, S.: Java pathfinder - second generation of a java model checker (2000)
7. Chaki, S., Clarke, E., Groce, A., Jha, S., Veith, H.: Modular verification of software components in c. In: Proc. of the 25th int. conf. on Software engineering, IEEE Computer Society (2003) 385–395

8. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: CAV. (2000) 154–169
9. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In Jensen, K., Podelski, A., eds.: TACAS 2004. Volume 2988 of Lecture Notes in Computer Science., Springer (2004) 168–176
10. Clarke, E., Kroening, D., Sharygina, N., Yorav, K.: Predicate abstraction of ANSI–C programs using SAT. FMSD (2004) To appear.
11. Flanagan, C.: Automatic software model checking using clp. In: Proc. of ESOP 03. Volume 2618 of LNCS., Springer (2003) 189–203
12. Holzmann, G.J., Smith, M.H.: Software model checking: extracting verification models from source code. Software Testing, Verification and Reliability **11** (2001) 65–79
13. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Conference Record of the Fifth Annual ACM SIGPLAN-SIGACT POPL Symposium, ACM Press, New York, NY (1978) 84–97
14. Cousot, P., Cousot, R.: Comparison of the Galois connection and widening/narrowing approaches to abstract interpretation. JTASPEFL '91, Bordeaux. BIGRE **74** (1991) 107–110
15. de Moura, L., Ruess, H., Shankar, N., Rushby, J.: The ICS decision procedure for embedded deduction. To appear at the IJCAR, Cork, Ireland (2004)
16. Chen, H., Wagner, D.: MOPS: an infrastructure for examining security properties of software. In: Proceedings of the 9th ACM Conference on Computer and Communications Security, Washington, DC (2002) 235–244
17. Schwoon, S.: Model-Checking Pushdown Systems. PhD thesis, Technische Universität München (2002)
18. Weissenbacher, G.: An Abstraction/Refinement Scheme for Model Checking C Programs. PhD thesis, Technische Universität Graz (2003)
19. Qadeer, S., Wu, D.: Kiss: keep it simple and sequential. In: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation, ACM Press (2004) 14–24
20. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: POPL 2002. (2002) 58–70
21. Kelly, W., Maslov, V., Pugh, W., Rosser, E., Shpeisman, T., Wonnacott, D.: The omega library interface guide. Technical report, Univ. of Maryland at College Park (1995)
22. Armando, A., Castellini, C., Mantovani, J.: Introducing full linear arithmetic to symbolic software model checking. Technical report, available at URL: http://www.ai.dist.unige.it/eureka. University of Genova (2004)
23. Armando, A., de Lucia, P.: Symbolic model-checking of linear programs. Technical report, Datalogiske Skrifter, Technical Report No. 94, Roskilde University (2002)
24. Lassez, J.L., Maher, M.: On Fourier's Algorithm's for Linear Arithmetic Constraints. JAR **9** (1992) 373–379
25. Reps, T., Horwitz, S., Sagiv, M.: Precise interprocedural dataflow analysis via graph reachability. POPL **95** (1995) 49–61