

# Introducing Full Linear Arithmetic to Symbolic Software Model Checking\*

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 counter 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 facts; 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 can 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  $c_0 + c_1x_1 + \dots + c_nx_n$ , where  $c_0, \dots, c_n$  are numeric constants and  $x_1, \dots, x_n$  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 Reps, Horwitz, and Sagiv [24]. 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. Experimental results obtained by running our model checker against the library provide evidence of the effectiveness of the approach.

*Structure of the paper.* Section 2 introduces some basic definitions concerning Linear Programs; Section 3 is devoted to our model checking procedure; Section 4 describes the eureka model checker, the benchmark problems, the experimental results, and outlines our future work; finally, Section 5 draws some conclusions.

## 2 Linear Programs

Most of the notation and concepts introduced in this Section are inspired by, or are extensions of, those presented in [1].

### 2.1 Syntax, Variables, Scope

The syntax of Linear Programs is shown in Table 1. As is customary, a superscript  $*$  symbol denotes zero or more occurrences of the preceding item, whereas  $+$  denotes *one* or more.

**Table 1.** Syntax of Linear Programs. A program is denoted by the identifier *prog*.

<i>prog</i> ::= <i>decl</i> * <i>proc</i> *	Global variables declaration and procedure definition
<i>decl</i> ::= <b>int</b> <i>id</i> <sup>+</sup> ;	Variable declaration
<i>id</i> ::= [a-zA-Z_][a-zA-Z0-9_]*	Identifier
<i>proc</i> ::= <i>id</i> ( <i>id</i> * ) <i>decl</i> * <i>sseq</i>	Procedure declaration
<i>sseq</i> ::= <i>lstmt</i> <sup>+</sup>	Sequence of statements
<i>lstmt</i> ::= <i>stmt</i>   <i>id</i> : <i>stmt</i>	Non-labelled statement Labelled statement
<i>stmt</i> ::= ;   <b>goto</b> <i>id</i> ;   <i>id</i> = <i>expr</i> ;   <b>if</b> ( <i>pred</i> ) { <i>sseq</i> } <b>else</b> { <i>sseq</i> }   <b>while</b> ( <i>pred</i> ) { <i>sseq</i> }   <b>assert</b> ( <i>pred</i> );   <i>id</i> ( <i>id</i> * );	Skip Go to Assignment Conditional Iteration Assertion Procedure call
<i>pred</i> ::= <i>expr op expr</i>	Predicate
<i>op</i> ::= <   >   =<   >=   =   !=	Operator
<i>expr</i> ::= <i>mon</i>   <i>mon</i> + <i>expr</i>   <i>mon</i> - <i>expr</i>	Linear expression
<i>mon</i> ::= <i>u_mon</i>   - <i>u_mon</i>	Monomial
<i>u_mon</i> ::= <i>const</i>   <i>id</i>   <i>const</i> * <i>id</i>	Unsigned monomial
<i>const</i> ::= [0-9]*	Numeric constant

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 =, ≠, <, >, ≤, ≥. A program is denoted by the top-level identifier *prog*.

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  $s_i$ .

## 2.2 The Control flow Graph

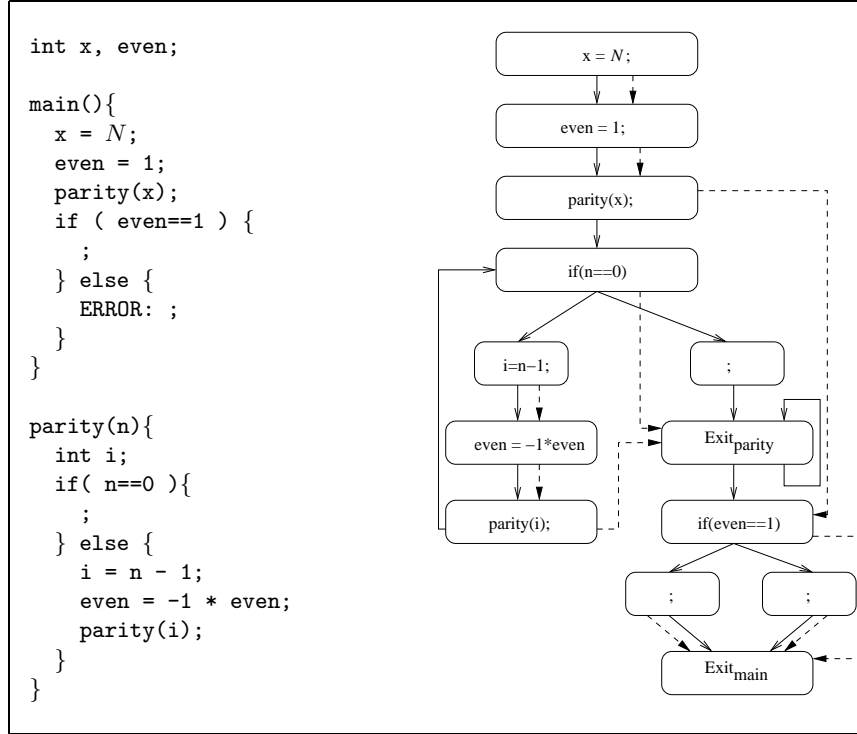
The *Control flow Graph* of a Linear Program  $P$  is a directed graph  $G_P = (V_P, Succ_P)$ , where  $V_P = \{0, 1, \dots, n, n + 1, \dots, n + p\}$  is the set of vertices, one for each statement (from 1 to  $n$ ) and one  $Exit_{pr}$  vertex for each procedure  $pr$  (from  $n + 1$  to  $n + p$ ). Vertex 0 is used to model the failure of an `assert` statement. Vertex 1 is the first statement of procedure `main`:  $1 = First_P(\text{main})$ . 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  $n$  times 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. Given the grammar of Table 1, the set  $Succ_P$  of possible successors of a statement is defined in terms of the function  $Next_P$ . The sequence of statements that have an *sseq* node as the parent of a statement  $s_i$  is called the *containing sequence* of  $s_i$ . Let  $a(s_i)$  the closest ancestor of a statement  $s_i$  that is a *stmt* node. If  $s_i$  is not the last statement in its containing sequence, then  $Next_P(i)$  is the statement immediately following  $s_i$  in that sequence. Otherwise,  $Next_P(i)$  is the statement immediately following  $a(s_i)$  in its containing sequence, if any, or the  $Exit_{pr}$  node, where  $pr = ProcOf_P(i)$ .

The set  $Succ_P(i)$  is defined in Table 2.  $Tsucc_P(i)$  and  $Fsucc_P(i)$  denote the successors of a conditional statement, in the *true* and *false* branch, respectively, and  $sSucc_P(i)$  is the single successor of a statement, when  $|Succ_P(i)| = 1$ .

## 2.3 Valuations and transitions

We now define valuations and transitions for a Linear Program. Let  $\mathcal{D}$  be a numerical domain, called the domain of computation. Given a vertex  $i \in V_P$ , a *valuation* is a function  $\omega : InScope_P(i) \rightarrow \mathcal{D}$  (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  $P$



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

**Table 2.** The set  $Succ_P(i)$ .

$i$	$Succ_P(i)$
$goto(\ell)$	$\{j\}$ , where $s_j$ is the statement labelled $\ell$
$x = e;$	$\{Next_P(i)\}$
$;$	$\{Next_P(i)\}$
$if$	$\{Tsucc_P(i), Fsucc_P(i)\}$ , where $Tsucc_P(i)$ is the first statement in the <b>then</b> branch, and $Fsucc_P(i)$ the first statement in the <b>else</b> branch.
$while$	$\{Tsucc_P(i), Fsucc_P(i)\}$ , where $Tsucc_P(i)$ is the first statement in the body of the <b>while</b> loop, and $Fsucc_P(i) = Next_P(i)$ .
$assert$	$\{Tsucc_P(i), Fsucc_P(i)\}$ , where $Tsucc_P(i) = Next_P(i)$ and $Fsucc_P(i) = n + p + 1$ .
$pr(e)$	$\{First_P(pr)\}$
$Exit_{pr}$	$\{Next_P(j)   s_j \text{ is a call to } pr\}$

are denoted by  $\langle i_k, \omega_1 \rangle \rightarrow_P^\alpha \langle i_{k+1}, \omega_2 \rangle$  where  $i_{k+1} \in Succ_P(i_k)$  and  $\alpha$  is a label ranging over the set of terminals:

$$\Sigma(P) = \{\sigma\} \cup \{\langle \mathbf{call}, i, \delta \rangle, \langle \mathbf{ret}, i, \delta \rangle : \exists j \in V_P \text{ s.t.} \\ s_j = \mathbf{call}, i = Next_P(j), \delta : Locals_P(j) \rightarrow \mathcal{D}\}.$$

The transition relation is defined in Table 3; terminals of the form  $\langle \mathbf{call}, i, \delta \rangle$  and  $\langle \mathbf{ret}, i, \delta \rangle$  represent, respectively, entry and exit points of the procedure invoked by  $s_j$ . A *path* is a sequence  $\langle i_0, \omega_0 \rangle \rightarrow_P^{\alpha_1} \langle i_1, \omega_1 \rangle \rightarrow_P^{\alpha_2} \dots \rightarrow_P^{\alpha_n} \langle i_n, \omega_n \rangle$  such

**Table 3.** The transition relation  $\langle i_k, \omega_1 \rangle \rightarrow_P^\alpha \langle i_{k+1}, \omega_2 \rangle$ ,  $i_{k+1} \in Succ_P(i_k)$ .

$i_k$	$\alpha$	$i_{k+1}$	$\omega_2$
<b>;</b>	$\sigma$	$sSucc_P(i_k)$	$\omega_2 = \omega_1$
<b>goto</b> ( $\ell$ )	$\sigma$	$sSucc_P(i_k)$	$\omega_2 = \omega_1$
$x = e$	$\sigma$	$sSucc_P(i_k)$	$\omega_2 = \omega_1[x/\omega_1(e)]$
<b>if</b> ( $d$ ) <b>while</b> ( $d$ ) <b>assert</b> ( $d$ )	$\sigma$	if $d = ?$ then $i_{k+1} \in Succ_P(i)$ else if $\omega_1(d) = true$ then $i_{k+1} = Tsucc_P(i_k)$ else $i_{k+1} = Fsucc_P(i_k)$	$\omega_2 = \omega_1$
$pr(e)$	$\langle \mathbf{call}, i, \delta \rangle$ where $\delta(x) = \omega_1(x)$ $\forall x \in Locals_P(i_k)$	$sSucc_P(pr)$	$\omega_2(x_l) = \omega_1(e_l)$ $\forall x_l \in Formals_P(i_{k+1})$ $\omega_2(g) = \omega_1(g)$ $\forall g \in Globals(L)$
<b>Exitpr</b>	$\langle \mathbf{ret}, i_{k+1}, \delta \rangle$	$i_{k+1} \in Succ_P(i_k)$	$\omega_2(g) = \omega_1(g)$ $\forall g \in Globals(L)$ $\omega_2(x_l) = \delta(x_l)$ $\forall x_l \in Locals_P(i_{k+1})$

that  $\langle i_k, \omega_k \rangle \rightarrow_P^{\alpha_{k+1}} \langle i_{k+1}, \omega_{k+1} \rangle$  for  $k = 0, \dots, n-1$ . Notice that not all paths represent potential execution paths: in a transition like  $\langle Exit_{pr}, \omega_1 \rangle \rightarrow_P^{\langle \mathbf{ret}, i_2, \delta \rangle} \langle i_2, \omega_2 \rangle$ , the valuation  $\delta$  can be chosen arbitrarily and therefore  $\omega_2$  is not guaranteed to coincide with  $\omega_1$  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* is a path  $\langle i_0, \omega_0 \rangle \rightarrow_P^{\alpha_1} \langle i_1, \omega_1 \rangle \rightarrow_P^{\alpha_2} \dots \rightarrow_P^{\alpha_n} \langle i_n, \omega_n \rangle$  such that  $\alpha_1 \dots \alpha_n$  is a string of the language generated from nonterminal  $M$  by the following context-free grammar:

$$V \rightarrow \epsilon | MV \\ V \rightarrow \langle \mathbf{call}, i, \delta \rangle V \quad \text{for all } \langle \mathbf{call}, i, \delta \rangle \in \Sigma(P)$$

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  $\langle \mathbf{ret}, i, \delta \rangle$  in the string associated to the path.

A *same-level valid path* is a path  $\langle i_0, \omega_0 \rangle \rightarrow_P^{\alpha_1} \langle i_1, \omega_1 \rangle \rightarrow_P^{\alpha_2} \dots \rightarrow_P^{\alpha_n} \langle i_n, \omega_n \rangle$  such that  $\alpha_1 \dots \alpha_n$  is a string of the language generated from nonterminal  $M$  by the following context-free grammar:

$$\begin{aligned} M &\rightarrow \epsilon \mid \sigma \mid MM \\ M &\rightarrow \langle \mathbf{call}, i, \delta \rangle M \langle \mathbf{ret}, i, \delta \rangle \quad \text{for all } \langle \mathbf{call}, i, \delta \rangle, \langle \mathbf{ret}, i, \delta \rangle \in \Sigma(P) \end{aligned}$$

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  $\omega$  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  $\Omega_i$  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  $\Omega_i$  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  $\langle 1, \omega_0 \rangle \rightarrow_P^{\alpha_1} \dots \rightarrow_P^{\alpha_k} \langle e, \omega_e \rangle$  and a same-level valid path  $\langle e, \omega_e \rangle \rightarrow_P^{\alpha_{k+1}} \dots \rightarrow_P^{\alpha_n} \langle i, \omega_i \rangle$  for some valuation  $\omega_0$ . 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  $s_i = \mathit{pr}(e_1, \dots, e_n)$ , let  $y_1, \dots, y_n$  be the formal parameters of  $\mathit{pr}$  associated to the actuals  $e_1, \dots, e_n$  respectively, and let  $\pi = \langle \omega_i, \omega_o \rangle$  be a path edge of a vertex  $\mathit{Exit}_{\mathit{pr}}$ . A *summary edge*  $\sigma = \langle \omega_1, \omega_2 \rangle$  of  $\pi$  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  $j = 1, \dots, n$ .

The computation of the summary edges is one of the most important parts of the algorithm. Summary edges record the output valuation  $\omega_2$  of a procedure for a given input valuation  $\omega_1$ . 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  $c_0 + c_1x_1 + \dots + c_nx_n$ , where  $c_0, c_1, \dots, c_n$  are constants and  $x_1, x_2, \dots, x_n$  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 linear formula that is always true, while  $\perp$  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  $D$ .<sup>1</sup> The following operations over DLCs are defined:

- *Application.* Let  $\lambda\mathbf{x}\lambda\mathbf{x}'.D$  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  $\lambda\mathbf{x}\lambda\mathbf{x}'.D$  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  $\delta_1$  and  $\delta_2$  be two ADLCs. Then,  $\delta_1 \sqcap \delta_2 = \lambda\mathbf{x}\lambda\mathbf{x}'.(\delta_1(\mathbf{x}, \mathbf{x}') \sqcap \delta_2(\mathbf{x}, \mathbf{x}'))$ .
- *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  $\delta_1$  and  $\delta_2$  be two ADLCs. Then,  $\delta_1 \sqcup \delta_2 = \lambda\mathbf{x}\lambda\mathbf{x}'.(\delta_1(\mathbf{x}, \mathbf{x}') \sqcup \delta_2(\mathbf{x}, \mathbf{x}')) = \lambda\mathbf{x}\lambda\mathbf{x}'.(\delta_1(\mathbf{x}, \mathbf{x}') \vee \delta_2(\mathbf{x}, \mathbf{x}'))$ .
- *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:  $\sim\delta = \lambda\mathbf{x}\lambda\mathbf{x}'.(\sim\delta(\mathbf{x}, \mathbf{x}'))$ .
- *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  $\delta_1$  and  $\delta_2$  be two ADLCs,  $\delta_1 \sqsubseteq \delta_2$  iff all the pairs of valuations satisfying  $\delta_1$  satisfy also  $\delta_2$ :  $\delta_1 \sqsubseteq \delta_2$  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 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  $\mathbf{y} = \text{Formals}_P(i)$ ,  $\mathbf{x} = \text{InScope}_P(i)$ ,  $\mathbf{z} = \text{sLocals}_P(i)$ ,  $\mathbf{g} = \text{Globals}(P)$ ; then  $\text{Lift}_{pr}(\delta) = \lambda\mathbf{g}\mathbf{g}'\mathbf{y}.\exists\mathbf{z}\mathbf{z}'\mathbf{y}'.\delta(\mathbf{x}, \mathbf{x}')$ . 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  $pr$ .<sup>2</sup>

<sup>1</sup> For sake of simplicity, in the rest of the paper we will write  $\lambda\mathbf{x}\lambda\mathbf{x}'.D$  instead of  $\lambda\mathbf{x}\lambda\mathbf{x}'.D$ .

<sup>2</sup> Note that our concept of Summary Edge is slightly different from the one described in [1].

**Transition Relations.** From here on, we will use the expression  $\mathbf{x}' = \mathbf{x}$  as a shorthand for  $x'_1 = x_1 \wedge \dots \wedge x'_n = x_n$ . 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  $\mathbf{y} = \mathbf{e}$ , 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  $pr(\mathbf{a})$ , 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  $\mathbf{x} = InScope_P(i)$ .

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

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

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

### 3.2 The Model Checking Procedure

The Model Checking procedure works by incrementally storing in an execution stack the next statement to be analyzed, by computing path edges and summary edges accordingly, and by removing the processed statement from the execution stack. Also, a call stack is maintained in order to keep track of the return points of each procedure called.

Let  $E$  be the execution stack and  $C$  be the call stack. Moreover, let  $P$  be an array of  $n + p + 1$  ADLCs and  $S$  be an array of  $p$  ADLCs<sup>3</sup>.  $P$  collects the path

<sup>3</sup> 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.

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 4-uple  $(1, \epsilon, P^1, \epsilon, S^1)$ , where  $P_1^1 = \lambda \mathbf{x} \mathbf{x}' . (\mathbf{x} = \mathbf{x}')$  with  $\mathbf{x} = InScope_P(1)$  and  $P_i^1 = \lambda \mathbf{y} \mathbf{y}' . \perp$  for each  $0 \leq i \leq (n + p)$  such that  $i \neq 1$ , and  $S_j^1 = \lambda \mathbf{y} \mathbf{y}' . \perp$  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 execution stack  $E$ , when needed. Given a vertex  $j$  (the next to be verified), the current execution stack  $E = i.is$ , 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 execution stack  $E'$  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  $prop$ .

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

A generic transition of the procedure is of the form  $(E, P, C, S) \rightarrow (E', P', C', S')$ , where the values of  $E', P', C', S'$  depend on the vertex being valuated, that is, on the vertex on top of the execution stack. Let  $i$  be the vertex on top of  $E$ , that is,  $E = i.is$ . 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

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

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

**Return from Procedure Calls.** Let  $i \in Exit_P$ ,  $C = c.cs$ , and  $s = Lift_c(P_i)$ . 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  $S'$ . Note that the call stack may never be empty, while applying this rule. If  $s \not\sqsubseteq S_i$  then

$$\begin{aligned} (E', P') &= prop(Next_P(c), E, SEJoin(P_i, S_i \sqcup s), P), \\ C' &= cs, \text{ and } S' = S[(S_i \sqcup s)/i]. \end{aligned}$$

Otherwise,  $E' = E$ ,  $P' = P$ ,  $C' = cs$ , and  $S' = S$ .

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

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

**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

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

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  $\perp$ , then that vertex is not reachable.

## 4 The eureka Toolkit

### 4.1 The Model Checker

We have implemented the approach described in the previous Section in a prototype system called *eureka*. *eureka* is written in about 2300 lines of (Sicstus) Prolog and employs, among the others, two main modules: (i) each time a *Join* operation is performed, a Prolog module *QE* is used, which performs quantifier elimination on an ADLC; in the current version of *eureka*, *QE* employs the Fourier-Motzkin method (see, e.g., [22]); (ii) each time an entailment check is performed, an external ground decision procedure for Linear Arithmetic is called upon. While the previous version of *eureka* was able to perform only propositional entailment checks, making the procedure highly incomplete, the current version makes use of ICS 2.0 [17] to this end. In doing so the implementation of our procedure is now sound and complete with respect to the entailment checks<sup>4</sup>. Moreover, for each vertex  $i$  of the control flow graph the maximal number of variables in the ADLC for  $i$  is  $O(|inScope_P(i)|)$ , and this helps the *QE* module not increasing its execution time.

<sup>4</sup> We translate  $<, >$  to  $\leq, \geq$  adding 1 or  $-1$  to the constants involved; the Fourier-Motzkin method works on the real numbers, whereas Linear Program variables range over the integers; this may lead to false negatives, but the problem has not arisen in practice so far.

## 4.2 The Benchmarks

In order to test the effectiveness and efficiency of our approach, we have devised a library of Linear Programs, that we call the **eureka** library, consisting of example programs grouped by their shape and characteristics. The library is realized in order to be (a) sufficiently close to real programming, given the intrinsic limitations of Linear Programs, (b) flexible enough to stress different aspects of the computation, (c) easily synthesizable from a set of templates and parameters. We have first identified four aspects of Linear Programs computation we are interested in (i) **data**: how much does the program stress data structures, i.e., to what extent does it sum, subtract and compare numerical values? Moreover, how many global and local variables are involved in arithmetic computations? (ii) **control**: how many conditionals are there in the program, that is, how complex is the control flow graph, and how complex are the conditions to be tested in the conditionals? (iii) **iteration**: how many iteration statements are there in the program, how complex are the iteration conditions and statements enclosed in the iteration blocks, and how many times is the block iterated? (iv) **recursion**: does the program make use of recursion? And, if so, how deep is it and how complex are the data manipulated in recursive procedures?

We have then devised six families of Linear Programs, of increasing expressivity (and, supposedly, difficulty), by building six template programs, each one stressing one or more of the above computation aspects. Each family is generated by instantiating the template according to a non-negative number  $N$ , somehow representing the hardness of a single instance. Here goes a description of the six families: `swap_seq.c(N)` defines two global variables and calls  $2N$  times a procedure `swap()` which swaps their values; `swap_iter.c(N)` does the same, but `swap()` is called  $N$  times from within a `while()` statement; `parity.c(N)` evaluates the parity of a natural number  $N$  using a recursive procedure; `delay_iter.c(N)` defines a procedure `delay()` which implements a delay via an empty, fixed-length loop; the procedure is then, in turn, called upon  $N$  times from within a `while()` statement; `delay_recur.c(N)` defines a *recursive* procedure `delay(n)` which implements a delay by recurring  $n$  times; like in `delay_iter.c(N)`, the procedure is then called upon  $N$  times from within a `while()` statement; `sum.c(N)` evaluates the sum of the first  $N$  natural numbers using a `while()` statement.

Lastly, willing to have an estimate of the hardness of each family, we have ranked them according to the number of aspects stressed; we have also noted how many global/local variables are defined in the program (local variables also include procedure formals), and how many conditions and/or iteration blocks involve them. The idea is that, because of the way our approach is structured, we expect long iteration loops, involving several variables, to be hard, since they potentially generate complex path/summary edges. Table 4 gives an account of this classification. All programs are roughly 30 lines long, except `swap_seq.c(N)`, whose length grows linearly with  $N$ . The **eureka** library is publicly available at the URL <http://www.mrg.dist.unige.it/eureka>.

**Table 4.** A library of benchmark Linear Programs. For each family, parametrised by  $N$ , the computation aspects stressed are listed, plus the number of (G)lobal and (L)ocal variables, of (C)onditions and (I)teration blocks.

Program name	data	control	iteration	recursion	G/L	C/I
<code>swap_seq.c(N)</code>	✓				2/1	1/0
<code>swap_iter.c(N)</code>	✓	✓	✓		2/2	2/1
<code>parity.c(N)</code>	✓	✓		✓	2/2	2/0
<code>delay_iter.c(N)</code>	✓	✓	✓		0/2	3/2
<code>delay_recur.c(N)</code>	✓	✓	✓	✓	0/3	3/1
<code>sum.c(N)</code>	✓	✓	✓		2/1	4/2

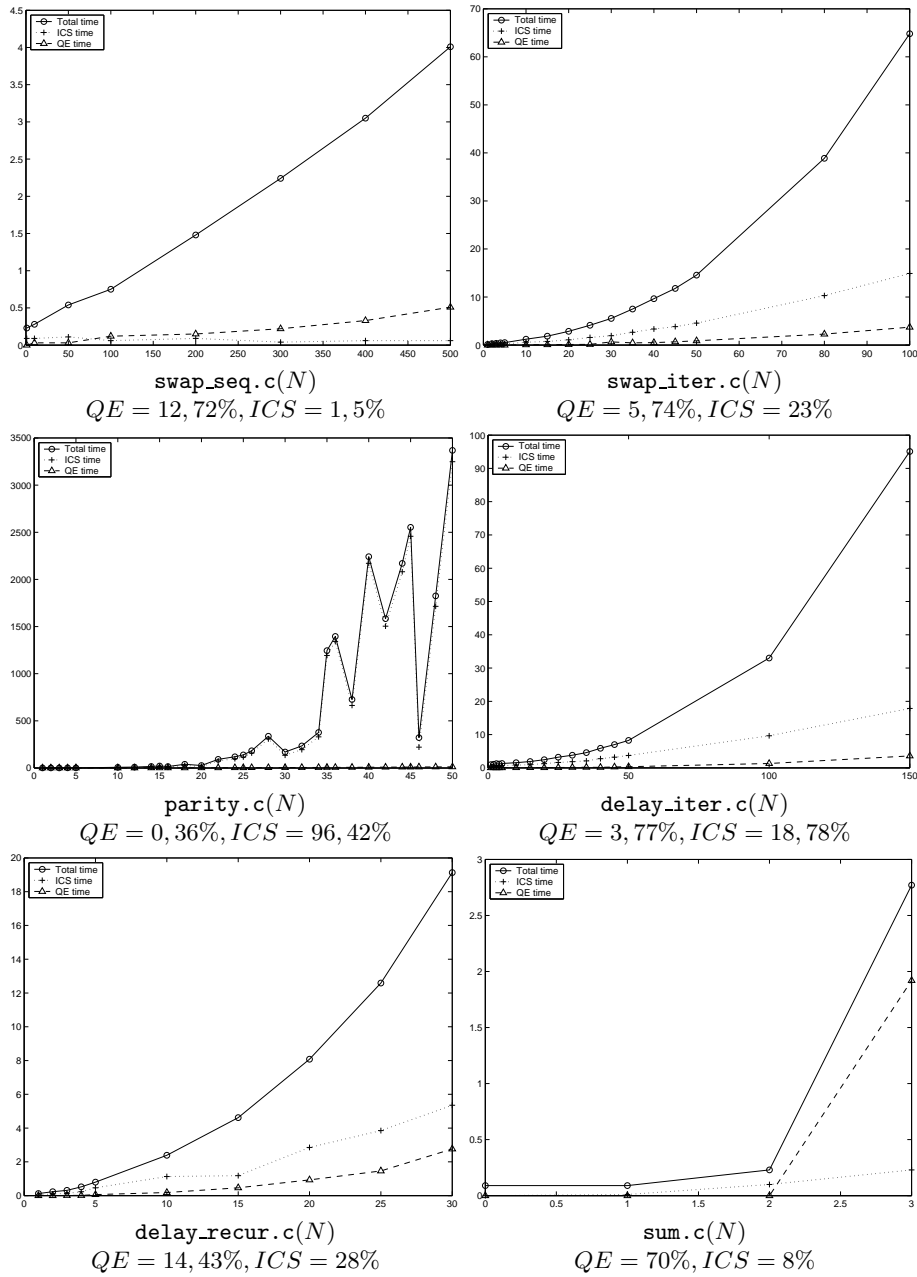
### 4.3 Experimental results

We have run `eureka` on the `eureka` library. All tests were performed on Pentium IV 1.8GHz machines, equipped with Linux RedHat 7.2. The memory limit was set automatically by Sicstus Prolog to 128 MB. The results obtained are visible in Figure 2. Graphs have  $N$  on the  $x$ -axis and CPU time on the  $y$ -axis; the curves plotted on each graph denote the total CPU time required to verify each program, the time spent by ICS and the time spent by the  $QE$  module, as  $N$  grows. Moreover, for each graph, we report the percentage of total CPU time spent in the  $QE$  module and in ICS for the hardest instance. A more detailed analysis follows, for each graph.

`swap_seq.c(N)`: as expected, this program presents a simple linear pattern, given by its sequential nature. As  $N$  grows, `swap()` is called upon repeatedly, but the summary edges do not change, so that once they have been calculated, they need not be re-calculated any longer. As a consequence, the model checking time (that is, total time minus  $QE$  and ICS time) dominates;  $QE$  time grows much more slowly, and ICS time is constant (entailment check needs be done just once).

`swap_iter.c(N)`: times grow quite fast, with model checking time dominating overall, and ICS time dominating quantifier elimination. This is due to the fact that the number of entailment checks is  $O(N)$ , one for each iteration, whereas the iteration block only involves one variable, an easy task for  $QE$ . Interestingly, the summary edge for the procedure `swap()` just needs be calculated twice (the two initial values of the global variables, and their swapped values) and does not grow any longer.

`parity.c(N)`: the curve presents a strong oscillating component, which is entirely due to ICS. In fact, ICS time dominates completely, and the number of *calls* to ICS grows linearly. If we ignore the oscillating component, we notice that the time starts rising for  $N > 30$ . Quantifier elimination is basically irrelevant. Considering again the structure of `parity.c(N)` (see Figure 1), it is clear that `parity()` is called  $x$  times, each time with a different actual argument and a different value of the variable `even`. This implies that `parity()`'s summary edge



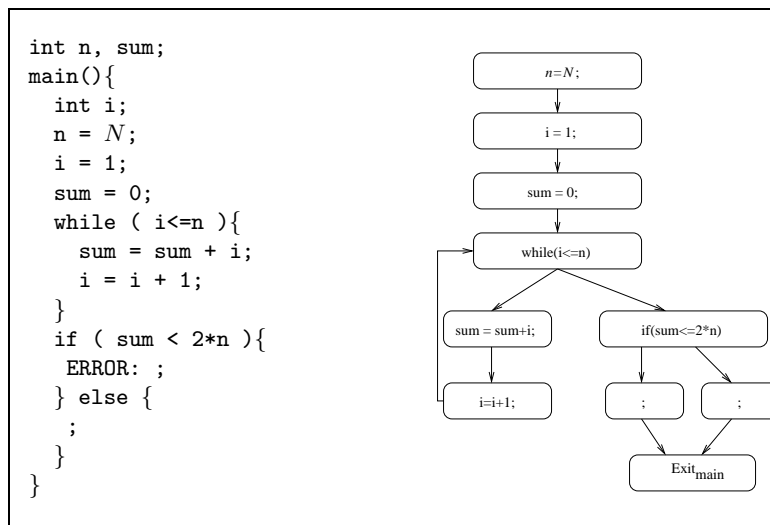
**Fig. 2.** Experimental results. Each graph shows the total CPU time, the time spent by ICS and the time required by the  $QE$  module, as  $N$  grows; each caption indicates the program family and the percentage of total CPU time spent in the  $QE$  module and in ICS for the hardest instance.

must be recalculated (and grows) at each recursion, making ICS's task harder and harder. We have no explanation so far for ICS's oscillating behaviour.

`delay_iter.c(N)`: this program behaves like `swap_iter.c(N)`. This seems surprising at first, since it defines no global variables, whereas `swap_iter.c(N)` has 2. A closer analysis reveals that in this case the summary edge (associated with `delay()`) must be calculated  $O(N)$  times, but turns out to be empty each time, since there are no globals and no formals. Therefore, analogously to `swap_iter.c(N)`, it does not grow as the computation proceeds.

`delay_recur.c(N)`: though it uses both iteration and recursion, this program behaves much like `parity.c(N)` and `swap_iter.c(N)`. The reason lies, once again, in the way the summary edge of `delay(n)`, which in this case *does* depend on a formal parameter, is calculated. It is first calculated for  $n = 0$ ; at the next call to `delay(n)`, with  $n = 1$ , it needs be calculated only for  $n = 1$ , since the procedure has already been called upon with  $n = 0$ ; same goes for  $n = 2, 3, \dots, N$ . As a result, its overall calculation is not too expensive.

`sum.c(N)`: figure 2 shows that this is a particularly hard program. In fact, already for  $N = 4$ , the path edge to be propagated within the `while()` statement that computes the sum becomes so complicated that the *QE* module runs out of memory. In fact, the total CPU time is dominated by the *QE* time.



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

Consider Figure 3, which shows the program and its control flow graph; the reason for its particular hardness is that the iteration condition, and the assignments inside the loop altogether, involve 3 variables (plus their primed and doubly primed counterparts, according to the procedure description of section

3, reaching a total of 9 variables). Subject to four iterations, at each of which no entailment is discovered, the path edge quickly becomes huge and the Fourier-Motzkin method suffers from space explosion.

**Discussion** Not all our expectations were met by the experimental results. In particular, one would expect problems to get harder and harder as more structural characteristics are added: data-intensivity, iteration, recursion; also, an increase in difficulty is usually associated with the number of lines of code. In fact, the real bottlenecks occur in the computation of path edges and summary edges. They can become prohibitively expensive as more and more *variables* and *assignments* are involved in long loops or deep recursions. For example, consider `sum.c` again: during the evaluation of `sum.c(4)`, the quantifier elimination module gets stuck, although the iteration loop does not look particularly hard.

As far as efficiency and effectiveness are concerned, `eureka` behaves quite well on most of the library, gracefully handling both iteration and recursion, and showing some interesting properties of scalability.

**Comparative Results.** Only 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. BLAST is a model checker for C programs that implements the abstraction/refinement paradigm “lazily”, that is, when an error in the model is found, it does not refine the whole abstraction that has been built, but only the fragment of code that generated the spurious trace. It must be remarked that this system (as well as [2]) is not tailored for Linear Programs, but for a superset of them<sup>5</sup>. We have run BLAST against the *iterative* programs of the `eureka` library, that is, against `swap_seq.c`, `swap_iter.c`, `delay_iter.c`, and `sum.c` with the following results. The error it usually reported was about the infeasibility of discovering new predicates during the refinement process. It happened on `swap_seq.c(1)`, `swap_iter.c(2)`, and on `sum(3)`. Only on `delay_iter.c` BLAST demonstrated a very good scalability property. The reader may look at Figure 4.

#### 4.4 Future Work

Still a lot of work has to be done. Firstly, 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 is undecidable in general, we plan to cope with the non termination of our model checking procedure by investigating the use of widening techniques [15, 4]. 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.

---

<sup>5</sup> The BLAST toolkit is able to verify C programs endowed with pointers and structures, for example, with the only constraint that the programs have not to be recursive.

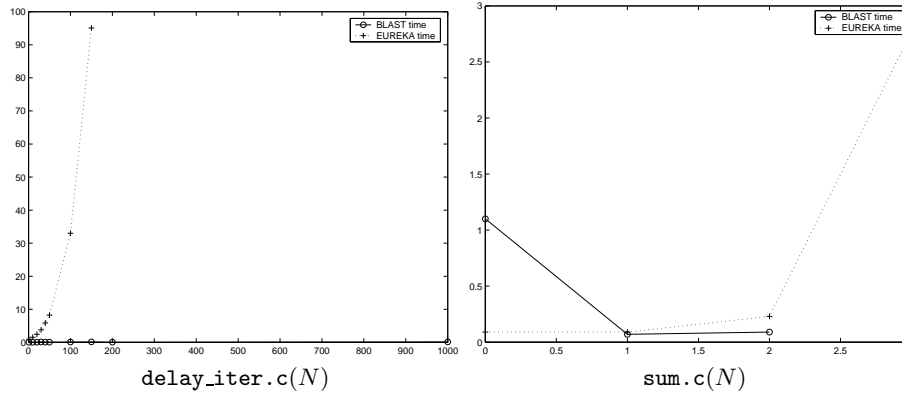


Fig. 4. Some comparative results between BLAST and EUREKA

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.

## 5 Conclusions

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 prototype implementation, called the eureka model checker, has been tested against the eureka library, a set of benchmark programs built to stress different aspects of computation. As shown in section 4, the experiments reveal that the approach is not only viable, but also promising.

## References

1. Ball, T., Rajamani, S.K.: Bebop: A symbolic model checker for boolean programs. In: Proc. of the 7th SPIN Workshop on Model checking of Software. (2000) 113–130
2. Ball, T., Rajamani, S.K.: Automatically validating temporal safety properties of interfaces. In: Proc. of the 8th SPIN Workshop on Model Checking of Software, Springer-Verlag New York, Inc. (2001) 103–122
3. Ball, T., Majumdar, R., Millstein, T.D., Rajamani, S.K.: Automatic predicate abstraction of c programs. In: SIGPLAN Conference on Programming Language Design and Implementation. (2001) 203–213

4. 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
5. Bultan, T., Gerber, R., Pugh, W.: Symbolic model checking of infinite state systems using presburger arithmetic. In: *Computer Aided Verification*. (1997) 400–411
6. Chaki, S., Clarke, E., Groce, A., Jha, S., Veith, H.: Modular verification of software components in c. In: *Proc. of the 25th international conference on Software engineering*, IEEE Computer Society (2003) 385–395
7. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: *Computer Aided Verification*. (2000) 154–169
8. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In Jensen, K., Podelski, A., eds.: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*. Volume 2988 of *Lecture Notes in Computer Science.*, Springer (2004) 168–176
9. Clarke, E., Kroening, D., Sharygina, N., Yorav, K.: Predicate abstraction of ANSI-C programs using SAT. *Formal Methods in System Design (FMSD)* (2004) To appear.
10. Flanagan, C., Qadeer, S.: Predicate abstraction for software verification. In: *Proc. of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of programming languages*, ACM Press (2002) 191–202
11. Flanagan, C.: Automatic software model checking using clp. In: *Proc. of ESOP 03: European Symposium on Programming*. Volume 2618 of *Lecture Notes in Computer Science.*, Springer (2003) 189–203
12. Choi, Y., Rayadurgam, S., Heimdahl, M.P.: Automatic abstraction for model checking software systems with interrelated numeric constraints. In: *Proc. of the 8th European Software Engineering Conference, joint with 9th ACM SIGSOFT symposium on Foundations of Software Engineering*, ACM Press (2001) 164–174
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 Symposium on Principles of Programming Languages*, Tucson, Arizona, ACM Press, New York, NY (1978) 84–97
14. Cousot, P., Cousot, R.: Refining model checking by abstract interpretation. *Automated Software Engineering: An International Journal* **6** (1999) 69–95
15. Cousot, P., Cousot, R.: Comparison of the Galois connection and widening/narrowing approaches to abstract interpretation. *JTASPEFL '91*, Bordeaux. *BIGRE* **74** (1991) 107–110
16. Cousot, P., Cousot, R.: Software analysis and model checking. In Brinksmma, E., Larsen, K., eds.: *Proc. of the 14th International Conference on Computer Aided Verification*, 2002. Copenhagen, Denmark, LNCS 2404, Springer (2002) 37–56
17. de Moura, L., Ruess, H., Shankar, N., Rushby, J.: The ICS decision procedure for embedded deduction. To appear at the *International Joint Conference of Automated Reasoning*, Cork, Ireland (2004)
18. Fribourg, L.: Constraint logic programming applied to model checking. Invited tutorial. In: *Proc. 9th Int. Workshop. on Logic-based Program Synthesis and Transformation (LOPSTR'99)*, Venezia, Italy, 1999. Volume 1817., Springer (2000) 31–42
19. S. Graf, H. Saidi: Construction of abstract state graphs with PVS. In Grumberg, O., ed.: *Proc. 9th Conference on Computer Aided Verification*, 1997. Volume 1254., Springer (1997) 72–83

20. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: Symposium on Principles of Programming Languages. (2002) 58–70
21. Kelly, W., Maslov, V., Pugh, W., Rosser, E., Shpeisman, T., Wonnacott, D.: The omega library interface guide. Technical report, University of Maryland at College Park (1995)
22. Lassez, J.L., Maher, M.: On Fourier’s Algorithm’s for Linear Arithmetic Constraints. *Journal of Automated Reasoning* **9** (1992) 373–379
23. Mueller-Olm, M., Seidl, H.: Precise interprocedural analysis through linear algebra. In: Proc. of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM Press (2004) 330–341
24. Reps, T., Horwitz, S., Sagiv, M.: Precise interprocedural dataflow analysis via graph reachability. *POPL* **95** (1995) 49–61
25. Strichman, O.: On solving presburger and linear arithmetic with sat. In: Proc. of the 4th International Conference on Formal Methods in Computer-Aided Design, Springer (2002) 160–170