

# ***Linear Constraints in Symbolic Software Model Checking***

Alessandro Armando, Claudio Castellini, **Jacopo Mantovani**

armando@dist.unige.it, drwho@dist.unige.it, jacopo@dist.unige.it

Artificial Intelligence Laboratory

DIST - University of Genova



# Software Model Checking

Who are the main actors?

- ⑥ The SLAM Toolkit (Ball et al., Microsoft Research),
- ⑥ BLAST (Henzinger et al., Berkeley University),
- ⑥ MAGIC (Clarke at CMU),
- ⑥ ...

What is their approach?

- ⑥ Predicate Abstraction ( $\rightarrow$  Boolean Programs),
- ⑥ Iterative Abstraction/Refinement.

# Software Model Checking

Issues in predicate abstraction:

- ⑥ Abstraction and Refinement are 'expensive' (intensive use of theorem provers),
- ⑥ Refinement may not terminate.



Need to refine as less as possible

Since Boolean Programs are rather coarse models, we propose

**Linear Programs**

as a finer grained abstraction.

# Linear Programs

We propose Linear Programs as a more suitable abstraction for Software Model Checking.

- ⑥ They are similar to Boolean Programs, **BUT**
- ⑥ Variables range over a numerical domain  $\mathcal{D}$
- ⑥ Expressions are 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  $\mathcal{D}$ .

## Example

Checks if the value of  $x$  is even (even=1) or not.

```
int x, even;
```

```
main() {  
    x = 6 ;  
    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);  
    }  
}
```

# The Control Flow Graph

The **Control flow Graph** of a program  $P$  is a directed graph  $G_P = (V_P, Succ_P)$ , where

- ⑥  $V_P$  is the set of vertices,
- ⑥  $Succ_P$  is the set of successors.

Let  $i \in V_P$ .

A **valuation** for  $i$  is a function  $\omega : InScope_P(i) \rightarrow \mathcal{D}$ .

A **state** is a pair  $\langle i, \omega \rangle$ .

# Trace Semantics

- ⑥ State transitions are of the form

$$\langle i_k, \omega_1 \rangle \rightarrow_P^\alpha \langle i_{k+1}, \omega_2 \rangle,$$

- ⑥ A **path** is a sequence

$$\langle i_0, \omega_0 \rangle \rightarrow_P^{\alpha_1} \langle i_1, \omega_1 \rangle \rightarrow_P^{\alpha_2} \dots \rightarrow_P^{\alpha_n} \langle i_n, \omega_n \rangle$$

subject to restrictions of the call semantics on  
 $\alpha_1, \dots, \alpha_n$ .

A state  $\langle i, \omega \rangle$  is **reachable** iff there exists a 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.

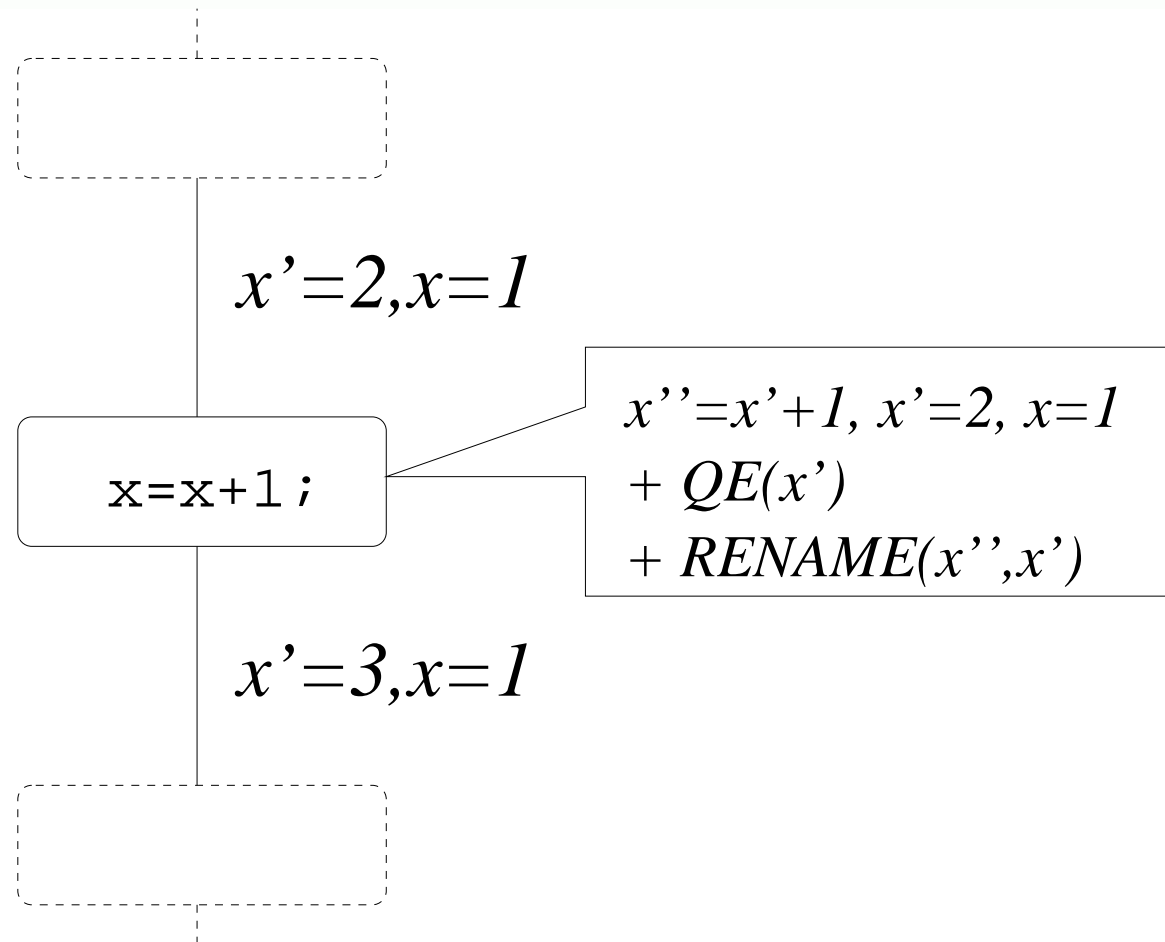
# *The procedure*

Similarly to the SLAM's Bebop approach (based on the Interprocedural Dataflow Analysis procedure of Reps, Horvitz, Sagiv), it computes

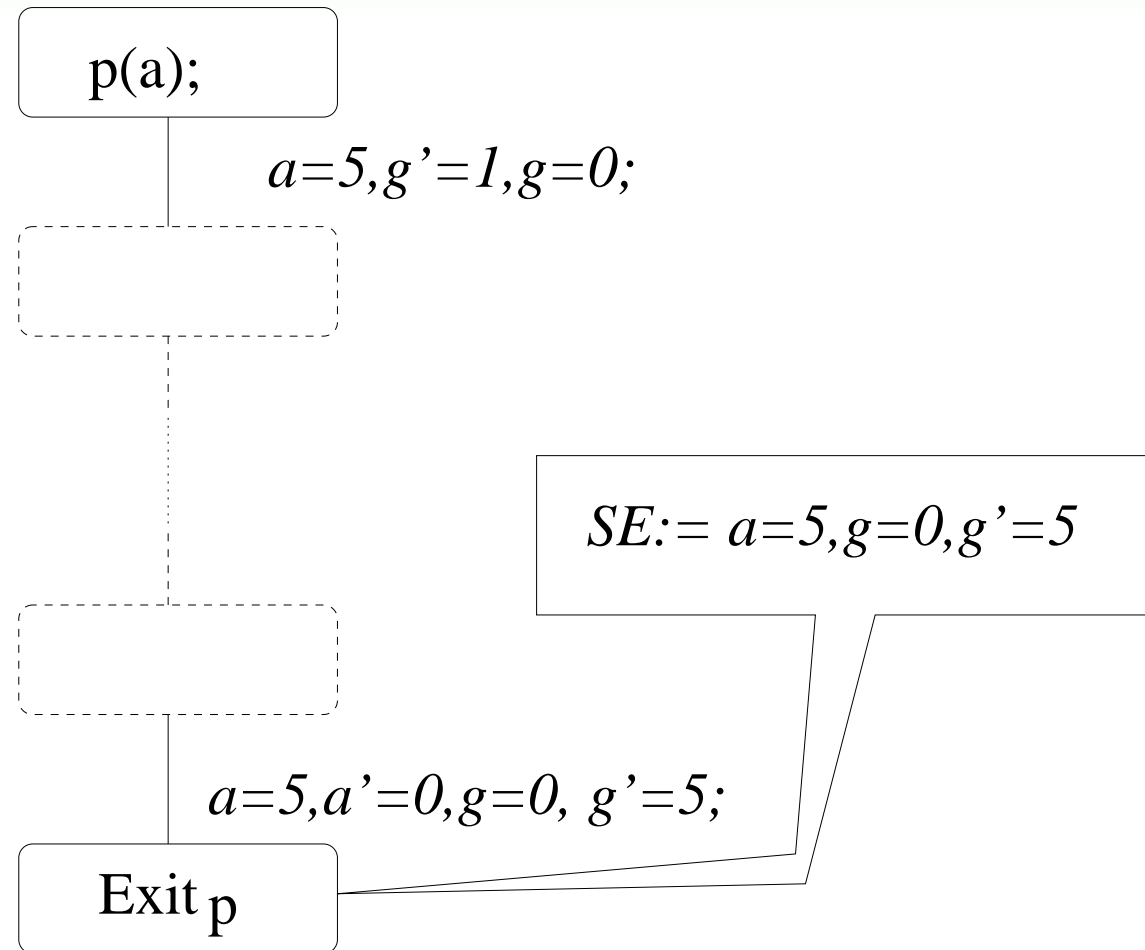
- ⑥ **path edges** to represent the reachability status of vertices
- ⑥ **summary edges** to record the input/output behaviour of procedures.

No need to compute twice the effects of procedures for the same input!

# Path Edges: example



# Summary Edges: example



# Symbolic Representation

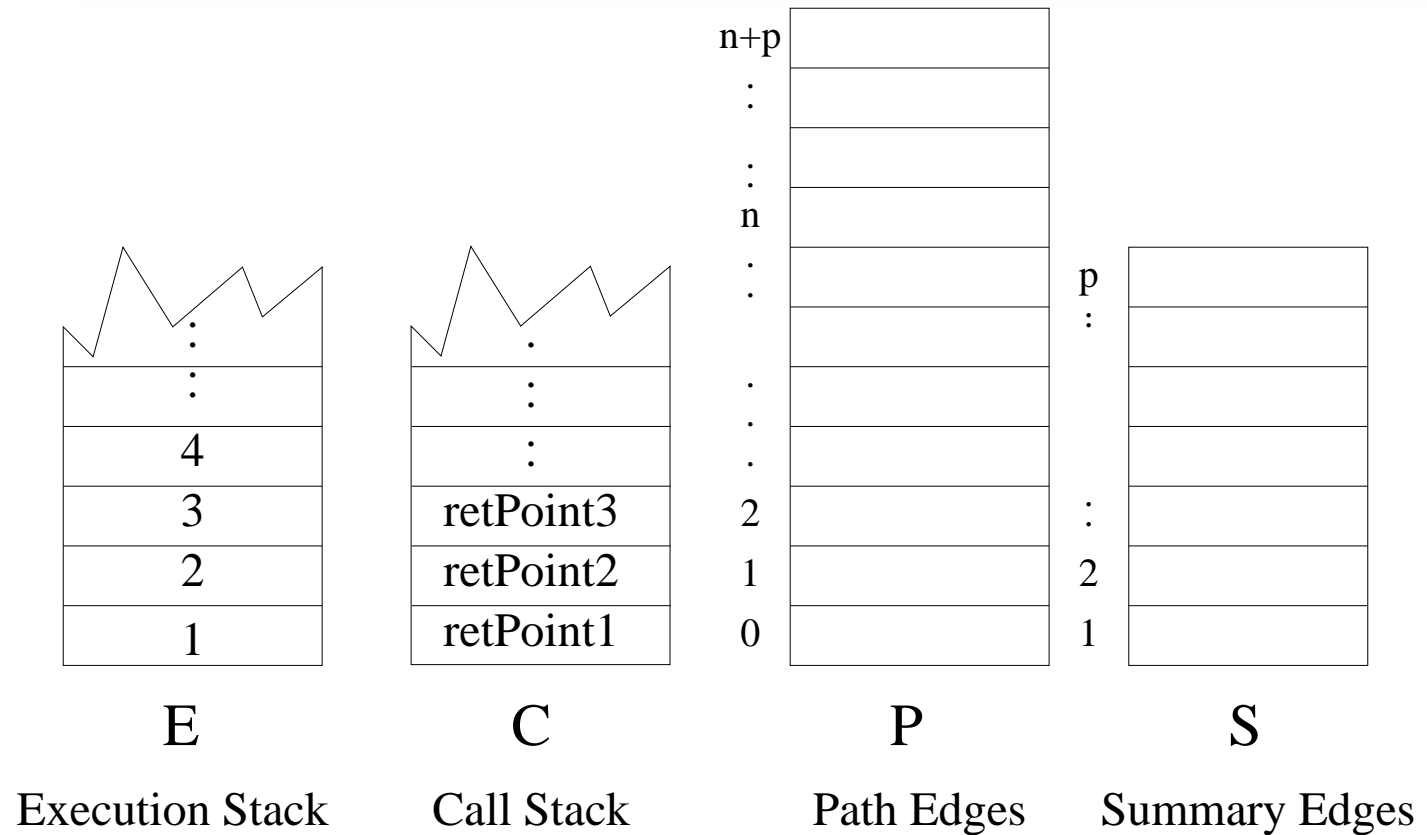
We represent path edges and summary edges by means of **Abstract Disjunctive Linear Constraints** (ADLCs), that is, expressions of the form

$$\lambda \mathbf{x} \lambda \mathbf{x}' . D$$

where  $D = \bigvee_i \bigwedge_j c_{ij}$ . Main operators:

- ⑥  $Join(P, \tau)$ : applies the transition relation  $\tau$  to a path edge  $P$ ,
- ⑥  $\sqsubseteq$ : entailment between ADLCs,
- ⑥  $\exists \mathbf{x}$ : quantifier elimination,
- ⑥  $\sqcap, \sqcup, \sim$ : and, or, not.

# The Procedure: data structure



# The Procedure: transitions

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  $E = i.is$ , with  $i \in V_P$ .

**Initial state:**  $(First_P(\text{main}).\epsilon, P^1, \epsilon, S^1)$ , where

- ⑥  $P_1^1 = \lambda \mathbf{x} \mathbf{x}' . (\mathbf{x} = \mathbf{x}')$
- ⑥  $P_i^1 = \lambda \mathbf{y} \mathbf{y}' . \perp$  for each  $0 \leq i \leq (n + p)$ ,  $i \neq First_P(\text{main})$
- ⑥  $S_j^1 = \lambda \mathbf{y} \mathbf{y}' . \perp$  for each  $1 \leq j \leq p$

## Rule example: Conditionals

Let  $i$  be a vertex associated with a conditional statement (`while`, `if`, `assert`), then

$$(E'', P'') = \text{prop}(F\text{succ}_P(i), E, \text{Join}(P_i, \tau_{i,\text{false}}), P),$$

$$(E', P') = \text{prop}(T\text{succ}_P(i), E'', \text{Join}(P_i, \tau_{i,\text{true}}), P''),$$

$$C' = C, \text{ and}$$

$$S' = S.$$

The function *prop*:

if 'new' information is provided (checked with the entailment operator) it updates the arrays  $E$  and  $P$ .

We have implemented **eureka**, a symbolic model checker for Linear Programs. It employs

- ⑥ symbolic representation of constraints,
- ⑥ quantifier elimination on ADLCs each time a *Join* operation is performed,
- ⑥ an external decision procedure (ICS 2.0) for the boolean combination of linear arithmetic constraints, each time an entailment check is performed.

# *Experiments*

Hard to find competitors, since

- ⑥ the SLAM Toolkit is not publicly available,
- ⑥ Flanagan's CLP tool is in a too preliminary stage.

**BLAST** (UCB) is the only tool (comparable to ours) that has been possible to obtain.

Problem: few benchmarks in literature.

To carry out quantitative analysis between BLAST and Eureka we've built a benchmark library of Linear Programs.

# Benchmarks

- ⑥ Aim: test the scalability of Eureka and allow comparison with rival tools
- ⑥ Programs are parametric in  $N$  (increasing difficulty for our tool)

Program name	data	control	iteration	recursion
<code>swap_seq.c(N)</code>	✓			
<code>swap_iter.c(N)</code>	✓	✓	✓	
<code>parity.c(N)</code>	✓	✓		✓
<code>delay_iter.c(N)</code>	✓	✓	✓	
<code>delay_recur.c(N)</code>	✓	✓	✓	✓
<code>sum.c(N)</code>	✓	✓	✓	

# Comparison

Numbers in table: the hardest instance ( $N$ ) of the programs that could be verified by the tools.

Program name	Eureka	BLAST
<code>swap_seq.c(N)</code>	500	no
<code>swap_iter.c(N)</code>	100	1
<code>parity.c(N)</code>	50	no
<code>delay_iter.c(N)</code>	150	1000
<code>delay_recur.c(N)</code>	30	no
<code>sum.c(N)</code>	3	2

# Benchmarks

## delay\_recur.c

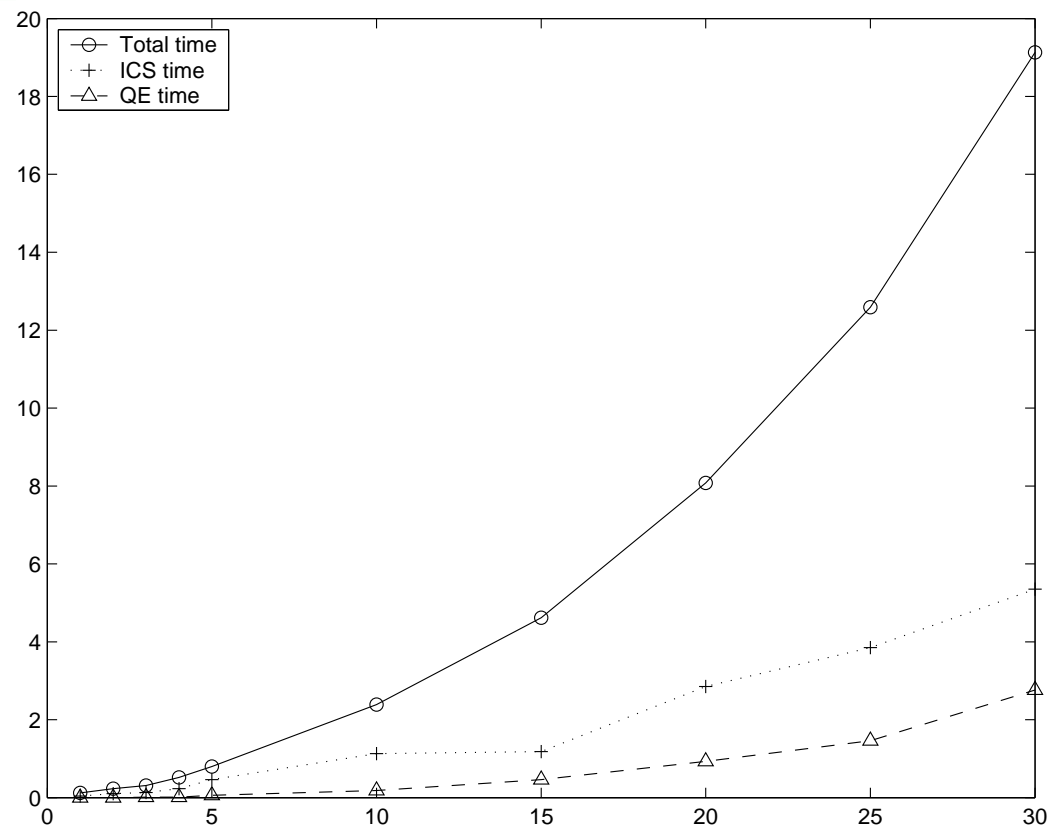
```
main() {
    int x;
    x = 0;

    while(x<10) {
        delay(x);
        x = x + 1;
    }
    if (x > 0) {
        ;
    } else {
        ERROR: ;
    }
}
```

```
delay(n) {
    int i;

    if ( n==0 ) {
        ;
    } else {
        i=n-1;
        delay(i);
    }
}
```

# Results



# Conclusions

So far:

- ⑥ A procedure for Model Checking of Software using Linear Constraints,
- ⑥ A prototype implementation called **eureka**,
- ⑥ A benchmark library,
- ⑥ Experimental Results and papers available at URL  
`http://www.ai.dist.unige.it/eureka`.

# Conclusions

## Future Work:

- ⑥ Complexity Analysis,
- ⑥ Add more examples and programs to the library,
- ⑥ Deal with Abstraction and Refinement,
- ⑥ More comparative analysis,
- ⑥ Use of widening techniques to enforce termination.