Software Model Checking Using Linear Constraints

A.Armando, C.Castellini, J.Mantovani

Artificial Intelligence Laboratory
University of Genova, Italy

Seattle, ICFEM 2004

Model Checking

Model Checking is a powerful technique for the Formal Verification of systems. It is used to verify

- hardware

Model Checking

Model Checking is a powerful technique for the Formal Verification of systems. It is used to verify

- hardware (read Finite State Systems),

Model Checking

Model Checking is a powerful technique for the Formal Verification of systems. It is used to verify

- hardware (read Finite State Systems),
- software

Model Checking

Model Checking is a powerful technique for the Formal Verification of systems. It is used to verify

- hardware (read Finite State Systems),
- software (read Infinite State Systems).

Model Checking

Model Checking is a powerful technique for the Formal Verification of systems. It is used to verify

- hardware (read Finite State Systems),
- software (read Infinite State Systems).

Main actors in Software Model Checking are:

- The SLAM Toolkit (Ball et al., Microsoft Research),

Model Checking

Model Checking is a powerful technique for the Formal Verification of systems. It is used to verify

- hardware (read Finite State Systems),
- software (read Infinite State Systems).

Main actors in Software Model Checking are:

- The SLAM Toolkit (Ball et al., Microsoft Research),
- BLAST (Henzinger et al., Berkeley University),

Model Checking

Model Checking is a powerful technique for the Formal Verification of systems. It is used to verify

- hardware (read Finite State Systems),
- software (read Infinite State Systems).

Main actors in Software Model Checking are:

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

Model Checking

Model Checking is a powerful technique for the Formal Verification of systems. It is used to verify

- hardware (read Finite State Systems),
- software (read Infinite State Systems).

Main actors in Software Model Checking are:

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

Model Checking

Model Checking is a powerful technique for the Formal Verification of systems. It is used to verify

- hardware (read Finite State Systems),
- software (read Infinite State Systems).

Main actors in Software Model Checking are:

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

What is the common approach?

Abstract, Check, Refine: the CEGAR approach

Given a program P and a property ϕ to check,

1. Build an abstract model $\hat{P}$ of P

Abstract, Check, Refine: the CEGAR approach

Given a program P and a property ϕ to check,

1. Build an abstract model $\hat{P}$ of P
2. Model check $\hat{P}$

Abstract, Check, Refine: the CEGAR approach

Given a program P and a property ϕ to check,

1. Build an abstract model $\hat{P}$ of P
2. Model check $\hat{P}$
3. Two possible outcomes:
   - If $\hat{P} \models \phi$, then $P \models \phi$, and return TRUE.

Abstract, Check, Refine: the CEGAR approach

Given a program P and a property ϕ to check,

1. Build an abstract model $\hat{P}$ of P
2. Model check $\hat{P}$
3. Two possible outcomes:
   - If $\hat{P} \models \phi$, then $P \models \phi$, and return TRUE.

Note that each abstraction is conservative, that is, it preserves the property to check: $\hat{\phi} = \phi$

Abstract, Check, Refine: the CEGAR approach

Given a program P and a property ϕ to check,

1. Build an abstract model $\hat{P}$ of P
2. Model check $\hat{P}$
3. Two possible outcomes:
   - If $\hat{P} \models \phi$, then $P \models \phi$, and return `TRUE`.
   - If $\hat{P} \not\models \phi$ with a counterexample $\hat{C}$, then check $\hat{C}$:

Abstract, Check, Refine: the CEGAR approach

Given a program P and a property ϕ to check,

1. Build an abstract model $\hat{P}$ of P
2. Model check $\hat{P}$
3. Two possible outcomes:
   - If $\hat{P} \models \phi$, then $P \models \phi$, and return `TRUE`.
   - If $\hat{P} \not\models \phi$ with a counterexample $\hat{C}$, then check $\hat{C}$:
     - If $\hat{C}$ is also a concrete counterexample, then $P \not\models \phi$ and return `FALSE`.

Abstract, Check, Refine: the CEGAR approach

Given a program P and a property ϕ to check,

1. Build an abstract model $\hat{P}$ of P
2. Model check $\hat{P}$
3. Two possible outcomes:
    - If $\hat{P} \models \phi$, then $P \models \phi$, and return TRUE.
    - If $\hat{P} \not\models \phi$ with a counterexample $\hat{C}$, then check $\hat{C}$:
        - If $\hat{C}$ is also a concrete counterexample, then $P \not\models \phi$ and return FALSE.
        - If $\hat{C}$ is spurious, then refine $\hat{P}$ and go back to Step 2.

Abstract, Check, Refine: the CEGAR approach

Given a program P and a property ϕ to check,

1. Build an abstract model $\hat{P}$ of P
2. Model check $\hat{P}$
3. Two possible outcomes:
    - If $\hat{P} \models \phi$, then $P \models \phi$, and return `TRUE`.
    - If $\hat{P} \not\models \phi$ with a counterexample $\hat{C}$, then check $\hat{C}$:
        - If $\hat{C}$ is also a concrete counterexample, then $P \not\models \phi$ and return `FALSE`.
        - If $\hat{C}$ is spurious, then refine $\hat{P}$ and go back to Step 2.

How to build the abstract model $\hat{P}$?

Abstraction

Predicate abstraction

Given

Abstraction

Predicate abstraction

Given

- a concrete program P,

Abstraction

Predicate abstraction

Given

- a concrete program P,
- a set of predicates E,

Abstraction

Predicate abstraction

Given

- a concrete program P,
- a set of predicates E,

let α be the abstraction function such that

$$\alpha(P, E) \rightarrow B,$$

where B is a Boolean Program.

Abstraction

Boolean Programs

Imperative programs whose variables are of type boolean only.

Abstraction

Boolean Programs

Imperative programs whose variables are of type boolean only.

From concrete to Boolean Programs

The abstraction function is such that

- it preserves the control-flow graph of the concrete program.

Abstraction

Boolean Programs

Imperative programs whose variables are of type boolean only.

From concrete to Boolean Programs

The abstraction function is such that

- it preserves the control-flow graph of the concrete program. Therefore, Boolean Programs support:
  - iteration constructs,
  - conditional constructs,
  - procedure calls (also with recursion) with call-by-value parameter passing;

Abstraction

Boolean Programs

Imperative programs whose variables are of type boolean only.

From concrete to Boolean Programs

The abstraction function is such that

- it preserves the control-flow graph of the concrete program. Therefore, Boolean Programs support:
  - iteration constructs,
  - conditional constructs,
  - procedure calls (also with recursion) with call-by-value parameter passing;
- each boolean variable b_i is in a one-to-one relation with a concrete predicate $e_i \in E$.

Example Boolean Program

```
decl g;
main()
begin
 decl h;
 h:= !g;
 A(g,h);
 skip;
 A(g,h);
 skip;
 if (g) then skip;
 else skip;
 fi
end
```

```
A(a1,a2)
begin
 if (a1) then
  A(a2,a1);
  skip;
 else
  g:= a2;
 fi
end
```

Issues in Software Model Checking
The CEGAR approach

- Abstraction and Refinement are 'expensive', as they make intensive use of theorem provers.

Issues in Software Model Checking

The CEGAR approach

- Abstraction and Refinement are 'expensive', as they make intensive use of theorem provers.
- Boolean abstraction is very coarse, and requires many Abstract-Check-Refine loops in order to output a solution.

Issues in Software Model Checking

The CEGAR approach

- Abstraction and Refinement are 'expensive', as they make intensive use of theorem provers.
- Boolean abstraction is very coarse, and requires many Abstract-Check-Refine loops in order to output a solution.
- The whole Abstract/Check/Refine loop may not terminate, even by choosing an abstraction (like Boolean Programs) for which the model checking procedure is terminating.

Linear Programs

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

Linear Programs

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

Linear Programs

Imperative programs such that:

- variables range over a numerical domain $\mathcal{D}$

Linear Programs

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

Linear Programs

Imperative programs such that:

- variables range over a numerical domain $\mathcal{D}$
- expressions are of the form

$$c_0 + c_1 x_1 + \ldots + c_n x_n,$$

where $c_0, \ldots, c_n$ are numeric constants and $x_1, \ldots, x_n$ are program variables ranging over $\mathcal{D}$.

Example Linear Program

```
int x,y;
main(){
 int k;
 k=0; x=10;   y=20;
 while(k<50){
  swap();
  k=k+1;
 }
 if(x==10){
  ERROR:  ;
 } else {
   ;
 }
}
```

```
swap(){
 int c;
 c=x;
 x=y;
 y=c;
}
```

Overview of the procedure

Similarly to the SLAM's Bebop approach (based on the Interprocedural Dataflow Analysis procedure of Reps, Horwitz, Sagiv [POPL, 1995]), the procedure computes reachability of vertices of the control-flow graph, using:

- Path Edges to represent the reachability status of vertices

Overview of the procedure

Similarly to the SLAM's Bebop approach (based on the Interprocedural Dataflow Analysis procedure of Reps, Horwitz, Sagiv [POPL, 1995]), the procedure computes reachability of vertices of the control-flow graph, using:

- Path Edges to represent the reachability status of vertices
- Summary Edges to record the input/output behaviour of procedures.

Overview of the procedure

Similarly to the SLAM's Bebop approach (based on the Interprocedural Dataflow Analysis procedure of Reps, Horwitz, Sagiv [POPL, 1995]), the procedure computes reachability of vertices of the control-flow graph, using:

- Path Edges to represent the reachability status of vertices
- Summary Edges to record the input/output behaviour of procedures.

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

Overview of the procedure

Similarly to the SLAM's Bebop approach (based on the Interprocedural Dataflow Analysis procedure of Reps, Horwitz, Sagiv [POPL, 1995]), the procedure computes reachability of vertices of the control-flow graph, using:

- Path Edges to represent the reachability status of vertices
- Summary Edges to record the input/output behaviour of procedures.

The 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.

Overview of the procedure

Example

```
int x,y;
main(){
 x=0;
 y=0;
 while(x<1){
  x=x+1;
  p(x,y);
 }
 ;
}
p(a,b){
 b=b+a;
}
```

$(x' = x) \wedge (y' = y)$

Overview of the procedure

Example

```
int x,y;
main(){
 x=0;
 y=0;
 while(x<1){
  x=x+1;
  p(x,y);
 }
 ;
}
p(a,b){
 b=b+a;
}
```

$(x' = x) \land (y' = y)$

$(x' = 0) \land (y' = y)$

Overview of the procedure

Example

```
int x,y;
main(){
 x=0;
 y=0;
 while(x<1){
  x=x+1;
  p(x,y);
 }
 ;
}
p(a,b){
 b=b+a;
}
```

$(x' = x) \wedge (y' = y)$

$(x' = 0) \wedge (y' = y)$

$(x' = 0) \wedge (y' = 0)$

Overview of the procedure

Example

```
int x,y;
main(){
  x=0;
  y=0;
  while(x<1){
    x=x+1;
    p(x,y);
  }
  ;
}
p(a,b){
  b=b+a;
}
```

Annotations beside the code:

- `main(){`: $(x' = x) \wedge (y' = y)$
- `x=0;`: $(x' = 0) \wedge (y' = y)$
- `y=0;`: $(x' = 0) \wedge (y' = 0)$
- `while(x<1){`: $(x' = 0) \wedge (y' = 0)$

Overview of the procedure

Example

```
int x,y;
main(){
  x=0;
  y=0;
  while(x<1){
    x=x+1;
    p(x,y);
  }
  ;
}
p(a,b){
  b=b+a;
}
```

$(x' = x) \wedge (y' = y)$

$(x' = 0) \wedge (y' = y)$

$(x' = 0) \wedge (y' = 0)$

$(x' = 0) \wedge (y' = 0)$

$(x' = 1) \wedge (x = 0) \wedge (y' = 0)$

Overview of the procedure

Example

```
int x,y;
main(){
  x=0;
  y=0;
  while(x<1){
    x=x+1;
    p(x,y);
  }
  ;
}
p(a,b){
  b=b+a;
}
```

- main(){ : $(x' = x) \wedge (y' = y)$
- x=0; : $(x' = 0) \wedge (y' = y)$
- y=0; : $(x' = 0) \wedge (y' = 0)$
- while(x<1){ : $(x' = 0) \wedge (y' = 0)$
- x=x+1; : $(x' = 1) \wedge (x = 0) \wedge (y' = 0)$
- p(a,b){ b=b+a; : $(a = 1) \wedge (a' = a) \wedge (b = 0) \wedge (b' = b)$

Overview of the procedure

Example

```
int x,y;
main(){
  x=0;
  y=0;
  while(x<1){
    x=x+1;
    p(x,y);
  }
  ;
}
p(a,b){
  b=b+a;
}
```

$(x' = x) \wedge (y' = y)$

$(x' = 0) \wedge (y' = y)$

$(x' = 0) \wedge (y' = 0)$

$(x' = 0) \wedge (y' = 0)$

$(x' = 1) \wedge (x = 0) \wedge (y' = 0)$

$(a = 1) \wedge (a' = a) \wedge (b = 0) \wedge (b' = b)$

$(a = 1) \wedge (a' = a) \wedge (b = 0) \wedge (b' = 1)$

Overview of the procedure

Example

```
int x,y;
main(){
 x=0;
 y=0;
 while(x<1){
  x=x+1;
  p(x,y);
 }
 ;
}
p(a,b){
 b=b+a;
}
```

main(){ — $(x' = x) \wedge (y' = y)$

x=0; — $(x' = 0) \wedge (y' = y)$

y=0; — $((x' = 0) \wedge (y' = 0)) \vee ((x' = 1) \wedge (x = 1) \wedge (y' = 1) \wedge (y = 0))$

while(x<1){ — $(x' = 0) \wedge (y' = 0)$

x=x+1; — $(x' = 1) \wedge (x = 0) \wedge (y' = 0)$

b=b+a; — $(x = 1) \wedge (x' = 1) \wedge (y = 0) \wedge (y' = 1)$

Overview of the procedure

Example

```
int x,y;
main(){
 x=0;
 y=0;
 while(x<1){
  x=x+1;
  p(x,y);
 }
 ;
}
p(a,b){
 b=b+a;
}
```

$(x' = x) \wedge (y' = y)$

$(x' = 0) \wedge (y' = y)$

$((x' = 0) \wedge (y' = 0)) \vee ((x' = 1) \wedge (x = 1) \wedge (y' = 1) \wedge (y = 0))$

$(x' = 0) \wedge (y' = 0)$

$(x' = 1) \wedge (x = 0) \wedge (y' = 0)$

$(x' = 1) \wedge (x = 1) \wedge (y' = 1) \wedge (y = 0)$

$(x = 1) \wedge (x' = 1) \wedge (y = 0) \wedge (y' = 1)$

Overview of the procedure

Example

```
int x,y;
main(){
  x=0;
  y=0;
  while(x<1){
    x=x+1;
    p(x,y);
  }
  ;
}
p(a,b){
  b=b+a;
}
```

main(){ : $(x' = x) \wedge (y' = y)$

x=0; : $(x' = 0) \wedge (y' = y)$

y=0; : $((x' = 0) \wedge (y' = 0)) \vee ((x' = 1) \wedge (x = 1) \wedge (y' = 1) \wedge (y = 0))$

while(x<1){ : $(x' = 0) \wedge (y' = 0)$

x=x+1; : $(x' = 1) \wedge (x = 0) \wedge (y' = 0)$

} : $(x' = 1) \wedge (x = 1) \wedge (y' = 1) \wedge (y = 0)$

; : $(x' = 1) \wedge (x = 1) \wedge (y' = 1) \wedge (y = 0)$

b=b+a; : $(x = 1) \wedge (x' = 1) \wedge (y = 0) \wedge (y' = 1)$

Overview of the procedure

Example 2

```
int x,y;
main(){
 int k;
 k=0; x=10;   y=20;
 while(k<50){
  swap();
  k=k+1;
 }
  ...
}
```

```
swap(){
 int c;
 c=x;
 x=y;
 y=c;
}
```

$$((x' = 10) \wedge (x = 20) \wedge (y' = 20) \wedge (y = 10))$$
$$\vee$$
$$((x' = 20) \wedge (x = 10) \wedge (y' = 10) \wedge (y = 20))$$

Overview of the procedure

Symbolic Representation

We represent Path Edges and Summary Edges by means of Abstract Disjunctive Linear Constraints (ADLCs for short).

Overview of the procedure

Symbolic Representation

ADLCs

An Abstract Disjunctive Linear Constraint is an expression of the form

$$\lambda \mathbf{x} \lambda \mathbf{x}'. \bigvee_i \bigwedge_j c_{ij}.$$

Overview of the procedure

Symbolic Representation

ADLCs

An Abstract Disjunctive Linear Constraint is an expression of the form

$$\lambda \mathbf{x} \lambda \mathbf{x}'. \bigvee_i \bigwedge_j c_{ij}.$$

Main operators over ADLCs:

- $Join(P, \tau)$: computes the ADLC resulting from the application of the transition relation τ to an ADLC P,
- $\sqsubseteq$: entailment,
- $\exists \mathbf{x}$: quantifier elimination,
- $\sqcap$, $\sqcup$, $^{\sim}$: and, or, not.

Eureka

We have implemented eureka, a fully automatic symbolic model checker for Linear Programs.

- Path Edges and Summary Edges are represented symbolically by ADLCs;
- ADLCs are manipulated with a Fourier-Motzkin Constraint Solver;
- It applies quantifier elimination on ADLCs each time a $Join$ operation is performed,
- Entailment checks are provided by ICS 2.0.

Comparison

- It was very difficult to compare eureka with other tools since most of them are either not publicly available or need auxiliary specifications.

Comparison

- It was very difficult to compare eureka with other tools since most of them are either not publicly available or need auxiliary specifications.
- We chose BLAST, the Berkeley Lazy Abstraction Software verification Tool, for a comparison.

Comparison

- It was very difficult to compare eureka with other tools since most of them are either not publicly available or need auxiliary specifications.
- We chose BLAST, the Berkeley Lazy Abstraction Software verification Tool, for a comparison.
- We carried out some quantitative analysis between BLAST and Eureka on our benchmark library of Linear Programs.

The Eureka library of Benchmarks

- Aim: assess the scalability of the model checkers;
- Programs are parametric in a positive integer N (the higher N, the more difficult the program)

The Eureka library of Benchmarks

- Aim: assess the scalability of the model checkers;
- Programs are parametric in a positive integer N (the higher N, the more difficult the program)

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

Experimental Results

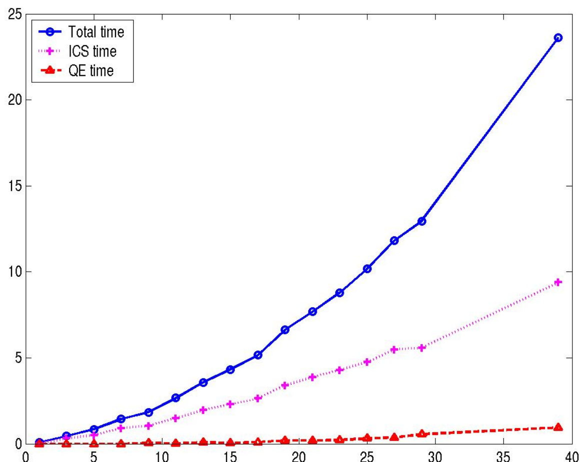
Delay_recur.c(N)

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

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

Experimental Results

Delay_recur.c(N)



Experimental Results

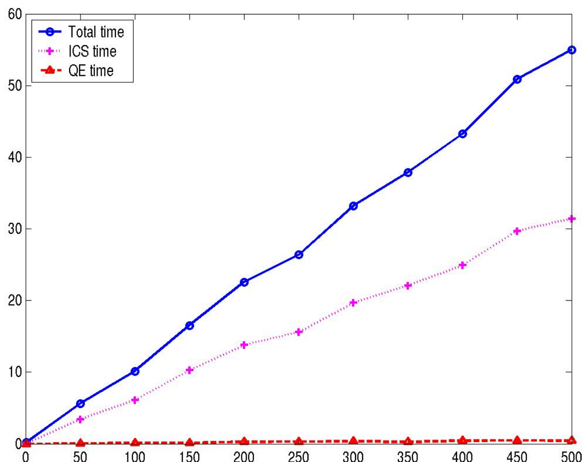
Swap_seq.c(N)

```
int x,y;
main(){
 x=10; y=20;
 swap();
 swap();
 /* 2N times */
 swap();
 swap();
 if (x==10){
   ;
 } else { ERROR: ; }
}
```

```
swap(){
 int c;
 c=x;
 x=y;
 y=c;
}
```

Experimental Results

Swap_seq.c(N)



Experimental Results

Comparison with BLAST

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

Program name	Eureka	BLAST
swap_seq.c(N)	500	no
swap_iter.c(N)	100	1
parity.c(N)	50	no
delay_iter.c(N)	150	1000
delay_recur.c(N)	39	no
sum.c(N)	3	2

Summary

- We developed a model checking procedure for software, that uses Linear Constraints;
- We devised a prototype implementation of the procedure called eureka.
- We proposed a set of benchmarks, called the eureka library, to assess the scalability property of our tool and to allow a comparison with BLAST.
- We had good experimental results!
- All numbers, papers and talks are available at URL

  `http://www.ai.dist.unige.it/eureka`.

Future Work

- Find classes of programs for which the procedure always terminates,
- Complexity Analysis for those classes of programs,

Future Work

- Find classes of programs for which the procedure always terminates,
- Complexity Analysis for those classes of programs,
- Extend the library,

Future Work

- Find classes of programs for which the procedure always terminates,
- Complexity Analysis for those classes of programs,
- Extend the library,
- Embed eureka in the Abstraction and Refinement loop (current),

Future Work

- Find classes of programs for which the procedure always terminates,
- Complexity Analysis for those classes of programs,
- Extend the library,
- Embed eureka in the Abstraction and Refinement loop (current),
- Do more comparative analysis,

Future Work

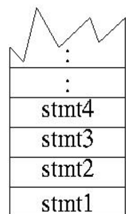
- Find classes of programs for which the procedure always terminates,
- Complexity Analysis for those classes of programs,
- Extend the library,
- Embed eureka in the Abstraction and Refinement loop (current),
- Do more comparative analysis,
- Use of widening techniques to enforce termination (e.g. Parma Polyhedra Library, Omega tool).

The End

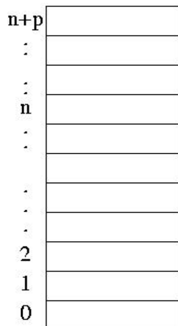
THANK YOU!

Overview of the procedure
Data Structures

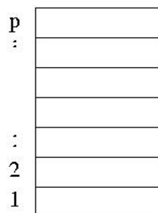
n: # of vertices, p: # of procedures.



$W : \mathbb{N} \rightarrow V_P$
Worklist



$P : \mathbb{N} \rightarrow ADLCs$
Path Edges



$S : \mathbb{N} \rightarrow ADLCs$
Summary Edges

Procedure Transitions

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.

Procedure Transitions

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.
Initial state: $(\,First_P(\texttt{main}).\epsilon,\, P, S)$

Procedure Transitions

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.
Initial state: $(1.\epsilon, P, S)$, where

Procedure Transitions

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.
Initial state: $(1.\epsilon, P, S)$, where

- $P(1) = \lambda \mathbf{x}\mathbf{x}'.(\mathbf{x} = \mathbf{x}')$
- $P(i) = \lambda \mathbf{y}\mathbf{y}'.\bot$ for each $0 \leq i \leq (n + p)$, $i \neq 1$
- $S(j) = \lambda \mathbf{y}\mathbf{y}'.\bot$ for each $1 \leq j \leq p$

Procedure Transitions

Conditionals

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

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

Procedure Transitions

Conditionals

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

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

The purpose of the function $prop$ is to update W and P if 'new' information is provided (checked with the entailment operator).

Procedure Transitions

Procedure Calls

Let i be a vertex associated with statement $s_i = pr(\mathbf{a})$; let $p = Exit_{pr}$, $l_i = SelfLoop(Join(P(i), \tau_i))$, and $r_i = SEJoin(P(i), S(p))$.

Procedure Transitions

Procedure Calls

Let i be a vertex associated with statement $s_i = pr(\mathbf{a})$; let $p = Exit_{pr}$, $l_i = SelfLoop(Join(P(i), \tau_i))$, and $r_i = SEJoin(P(i), S(p))$. If $l_i \not\sqsubseteq P(sSucc_P(i))$,

Procedure Transitions

Procedure Calls

Let i be a vertex associated with statement $s_i = pr(\mathbf{a})$; let $p = Exit_{pr}$, $l_i = SelfLoop(Join(P(i), \tau_i))$, and $r_i = SEJoin(P(i), S(p))$. If $l_i \not\sqsubseteq P(sSucc_P(i))$, then

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

Procedure Transitions

Procedure Calls

Let i be a vertex associated with statement $s_i = pr(\mathbf{a})$; let $p = Exit_{pr}$, $l_i = SelfLoop(Join(P(i), \tau_i))$, and $r_i = SEJoin(P(i), S(p))$. 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,

$$\begin{aligned} (W', P') &= prop(Next_P(i), W, r_i, P), \\ S' &= S. \end{aligned}$$

Procedure Transitions

Exit from Procedure Calls

Let $i \in Exit_P$, $w \in Succ_P(i)$, $c \in Call_P$ such that $w = ReturnPt_P(c)$, and $s = Lift_{\mathbf{c}}(P_i)$.

Procedure Transitions

Exit from Procedure Calls

Let $i \in Exit_P$, $w \in Succ_P(i)$, $c \in Call_P$ such that $w = ReturnPt_P(c)$, and $s = Lift_{\mathbf{c}}(P_i)$. If $s \not\sqsubseteq S(i)$

Procedure Transitions

Exit from Procedure Calls

Let $i \in Exit_P$, $w \in Succ_P(i)$, $c \in Call_P$ such that $w = ReturnPt_P(c)$, and $s = Lift_{\mathbf{c}}(P_i)$. If $s \not\sqsubseteq S(i)$ then for each w,

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

Procedure Transitions

Exit from Procedure Calls

Let $i \in Exit_P$, $w \in Succ_P(i)$, $c \in Call_P$ such that $w = ReturnPt_P(c)$, and $s = Lift_{\mathbf{c}}(P_i)$. 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,

$$\begin{aligned} W' &= W, \\ P' &= P \\ S' &= S. \end{aligned}$$

Procedure Transitions

All other cases

If $i \in V_P \setminus Cond_P \setminus Call_P \setminus Exit_P$, then

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

Procedure Transitions

All other cases

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,

Procedure Transitions

All other cases

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,
- S contains the valuations that show the behaviour of every procedure.

Procedure Transitions

All other cases

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,
- S contains the valuations that show the behaviour of every procedure.

Result: if the Path Edges for a vertex i are valuated to $\bot$, then s_i is not reachable.