Path-Sensitive Backward Slicing

Joxan JAFFAR[1] Vijayaraghavan MURALI[1]
Jorge A. NAVAS[2] Andrew SANTOSA[3]

[1]Department of Computer Science, National University of Singapore

[2]Department of Computer Science and Software Engineering
University of Melbourne

[3]School of Information Technology, University of Sydney

joxan@comp.nus.edu.sg
jorge.navas@unimelb.edu.au
santosa@it.usyd.edu.au

November 2012

Introduction

- We introduce *path-sensitivity* to *slicing*
  - Slicing: Given a *criterion* a control point *l* and variable x, find subset of statements that affects the value of x at *l*
  - Applications in parallelization, software testing, program comprehension, reverse engineering, etc.
- Path-sensitivity: Considers *infeasible* paths for precision
  - Typically considered expensive
- Our approach:
  - *Craig interpolation* to mitigate complexity
  - *Witness paths* to maintain accuracy

(Backward) Program Slicing

```
⟨0⟩ x=0;
⟨1⟩ if (a>=0) c=-a;
⟨2⟩ if (b>=0) x=y; else c=b;
⟨3⟩ if (c>0) x++; else x=1; ⟨4⟩
```

Criterion: $\langle\langle 4\rangle, x\rangle$

We compute a smallest set *INF*:

1. Include statements that affect the value of criterion variable
2. *Data dependency:* Include statements that affect the value of variables in *INF*
3. *Control dependency:* Include the if/while condition in *INF* if body contains statement in *INF*

(Backward) Program Slicing

```
⟨0⟩ x=0;
⟨1⟩ if (a>=0) c=-a;
⟨2⟩ if (b>=0) x=y; else c=b;
⟨3⟩ if (c>0) x++; else x=1; ⟨4⟩
```

Criterion: $\langle\langle 4\rangle, x\rangle$

We compute a smallest set *INF*:

1. Include statements that affect the value of criterion variable
2. *Data dependency:* Include statements that affect the value of variables in *INF*
3. *Control dependency:* Include the if/while condition in *INF* if body contains statement in *INF*

(Backward) Program Slicing

```
⟨0⟩ x=0;
⟨1⟩ if (a>=0) c=-a;
⟨2⟩ if (b>=0) x=y; else c=b;
⟨3⟩ if (c>0) x++; else x=1; ⟨4⟩
```

Criterion: $\langle\langle 4\rangle, x\rangle$

We compute a smallest set *INF*:

1. Include statements that affect the value of criterion variable
2. *Data dependency:* Include statements that affect the value of variables in *INF*
3. *Control dependency:* Include the if/while condition in *INF* if body contains statement in *INF*

(Backward) Program Slicing

```
⟨0⟩ x=0;
⟨1⟩ if (a>=0) c=-a;
⟨2⟩ if (b>=0) x=y; else c=b;
⟨3⟩ if (c>0) x++; else x=1; ⟨4⟩
```

Criterion: $\langle\langle 4\rangle, x\rangle$

We compute a smallest set *INF*:

1. Include statements that affect the value of criterion variable
2. *Data dependency:* Include statements that affect the value of variables in *INF*
3. *Control dependency:* Include the if/while condition in *INF* if body contains statement in *INF*

(Backward) Program Slicing

```
⟨0⟩ x=0;
⟨1⟩ if (a>=0) c=-a;
⟨2⟩ if (b>=0) x=y; else c=b;
⟨3⟩ if (c>0) x++; else x=1; ⟨4⟩
```

Criterion: $\langle\langle 4\rangle, x\rangle$

We compute a smallest set *INF*:

1. Include statements that affect the value of criterion variable
2. *Data dependency:* Include statements that affect the value of variables in *INF*
3. *Control dependency:* Include the if/while condition in *INF* if body contains statement in *INF*

(Backward) Program Slicing

```
⟨0⟩ x=0;
⟨1⟩ if (a>=0) c=-a;
⟨2⟩ if (b>=0) x=y; else c=b;
⟨3⟩ if (c>0) x++; else x=1; ⟨4⟩
```

Criterion: $\langle\langle 4\rangle, x\rangle$

We compute a smallest set *INF*:

1. Include statements that affect the value of criterion variable
2. *Data dependency:* Include statements that affect the value of variables in *INF*
3. *Control dependency:* Include the if/while condition in *INF* if body contains statement in *INF*

(Backward) Program Slicing

```
⟨0⟩ x=0;
⟨1⟩ if (a>=0) c=-a;
⟨2⟩ if (b>=0) x=y; else c=b;
⟨3⟩ if (c>0) x++; else x=1; ⟨4⟩
```

Criterion: $\langle\langle 4\rangle, x\rangle$

We compute a smallest set *INF*:

1. Include statements that affect the value of criterion variable
2. *Data dependency:* Include statements that affect the value of variables in *INF*
3. *Control dependency:* Include the if/while condition in *INF* if body contains statement in *INF*

(Backward) Program Slicing

```
⟨0⟩ x=0;
⟨1⟩ if (a>=0) c=-a;
⟨2⟩ if (b>=0) x=y; else c=b;
⟨3⟩ if (c>0) x++; else x=1; ⟨4⟩
```

Criterion: $\langle\langle 4\rangle, x\rangle$

We compute a smallest set *INF*:

1. Include statements that affect the value of criterion variable
2. *Data dependency:* Include statements that affect the value of variables in *INF*
3. *Control dependency:* Include the if/while condition in *INF* if body contains statement in *INF*

(Backward) Program Slicing

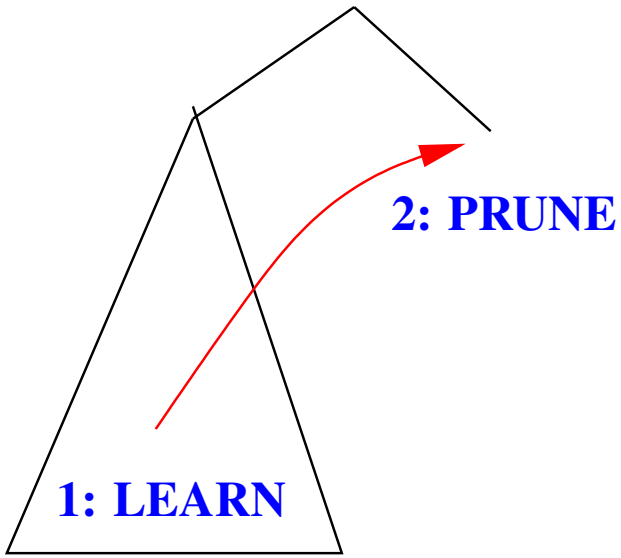
```
⟨0⟩ x=0;
⟨1⟩ if (a>=0) c=-a;
⟨2⟩ if (b>=0) x=y; else c=b;
⟨3⟩ if (c>0) x++; else x=1; ⟨4⟩
```

Criterion: $\langle\langle 4\rangle, x\rangle$

- The less statements included the better
- But we included all statements!
- `x=0` should not be in *INF* as no *feasible* path from `x=0` to `x++` s.t. all conditions *satisfied*

Our Solution

- Naive path-sensitive analysis
  $\rightarrow$ huge search space:
  exponential in the size of the program
- To mitigate the problem we employ *learning*



We use information from already traversed (symbolic execution) subtree to prune other subtrees

Symbolic Execution

- We consider program statements as *constraints*
- Analysis by traversal on symbolic execution tree

Example

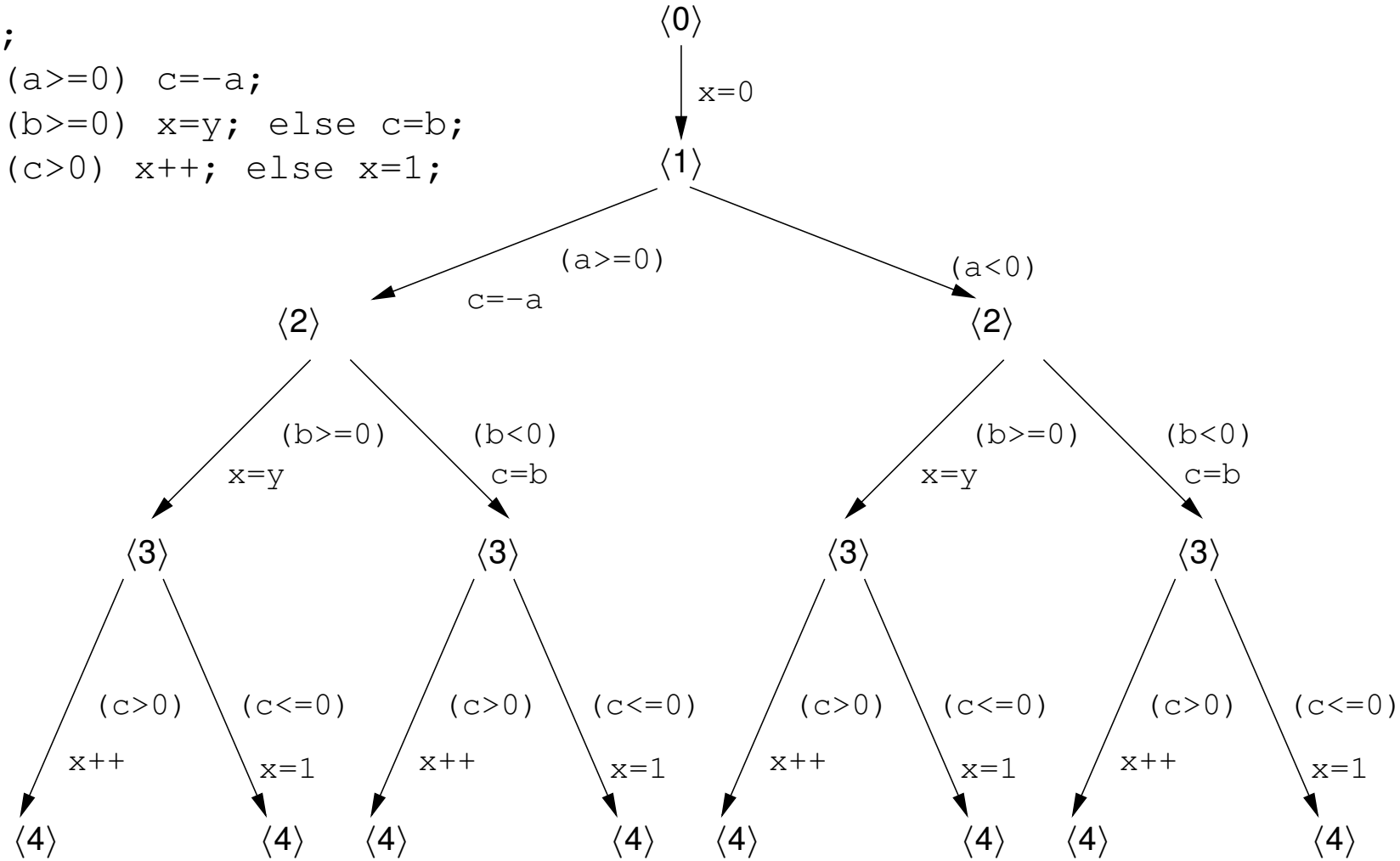
```
⟨0⟩ x=0;
⟨1⟩ if (a>=0) c=-a;
⟨2⟩ if (b>=0) x=y; else c=b;
⟨3⟩ if (c>0) x++; else x=1; ⟨4⟩
```

Criterion: $\langle\langle 4\rangle, \mathtt{x}\rangle$

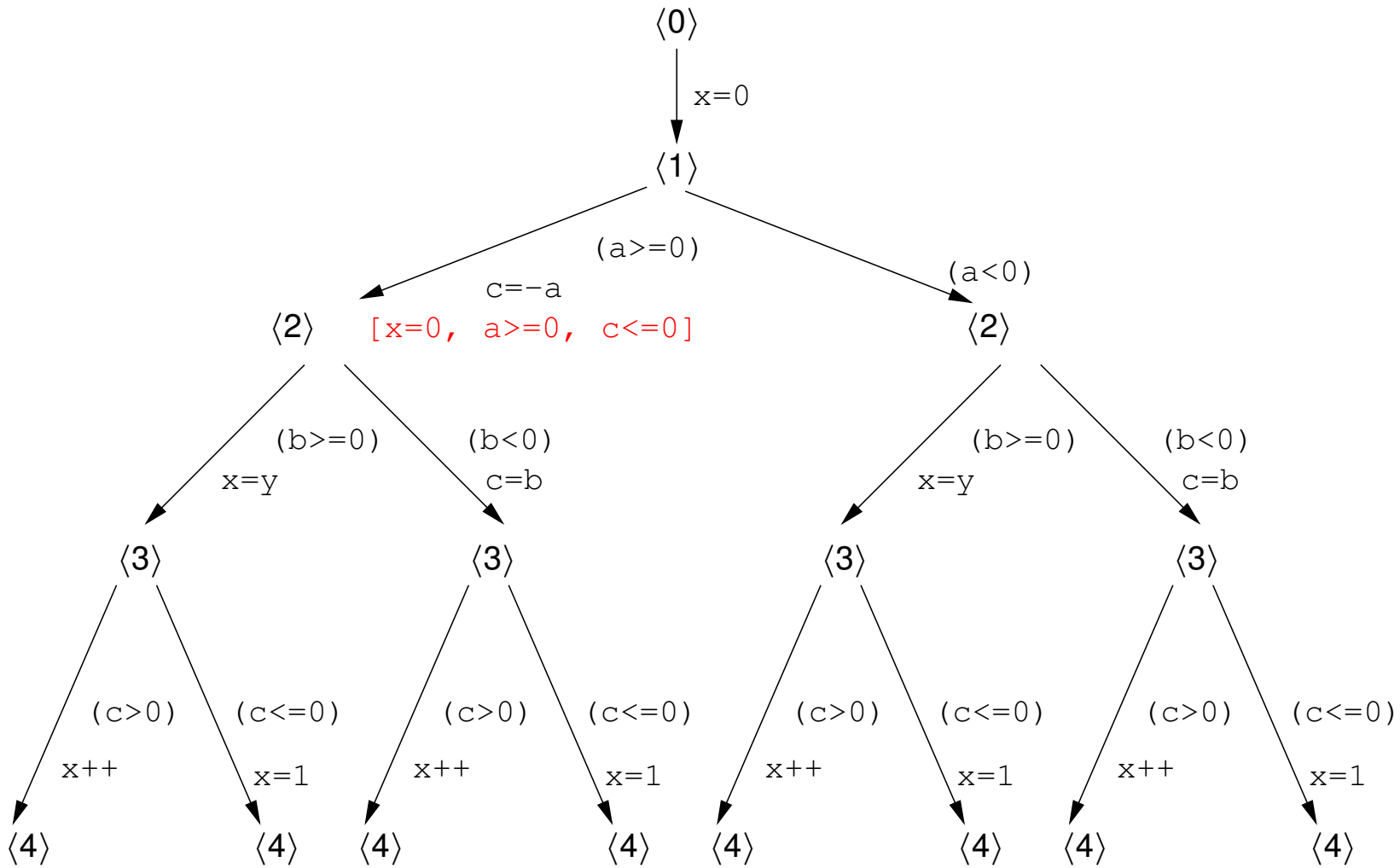
Next: The Tree

Symbolic Execution Tree

```
⟨0⟩  x=0;
⟨1⟩  if (a>=0) c=-a;
⟨2⟩  if (b>=0) x=y; else c=b;
⟨3⟩  if (c>0) x++; else x=1;
⟨4⟩
```



Symbolic Execution Tree



Symbolic State: constraints on program variables

Interpolation

To reduce search tree size:

Core technology: *Craig Interpolation*

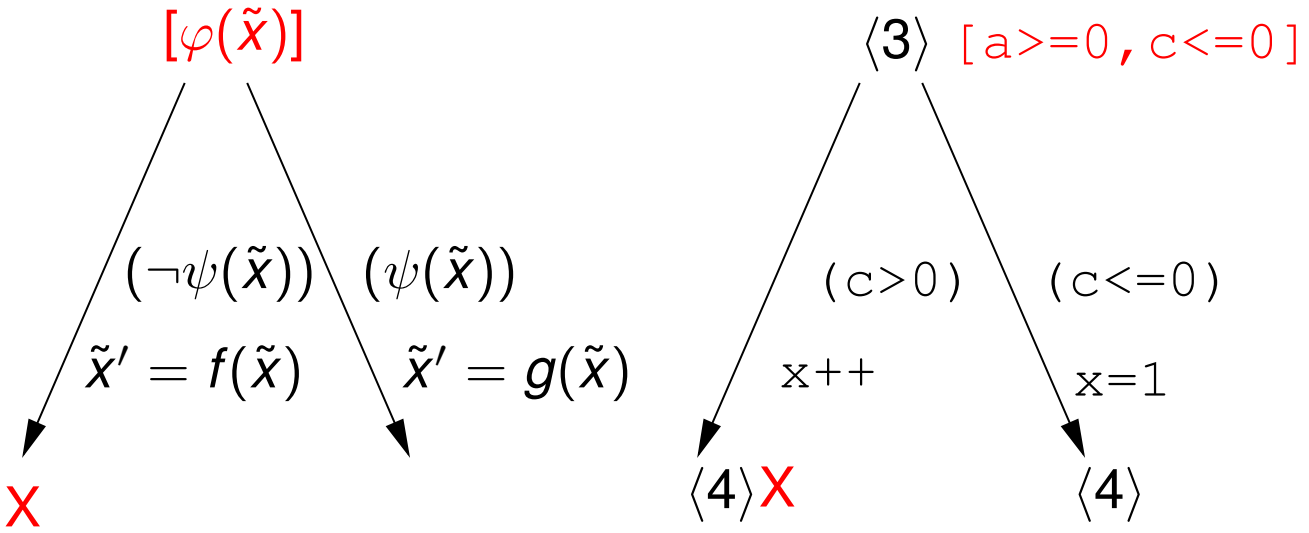
Example:

$$\langle 0\rangle\ \mathtt{x} = 0;\ \langle 1\rangle\ (\mathtt{a} > 0);\ \mathtt{c} = -\mathtt{a};\langle 2\rangle$$
$$(\mathtt{b} >= 0);\ \mathtt{x} = \mathtt{y};\ \langle 3\rangle\ (\mathtt{c} > 0);\ \mathtt{x}++;\ \langle 4\rangle$$

The last guard is *unsatisfiable* (hence, path is *infeasible*): remove statements not needed to ensure *unsatisfiability*

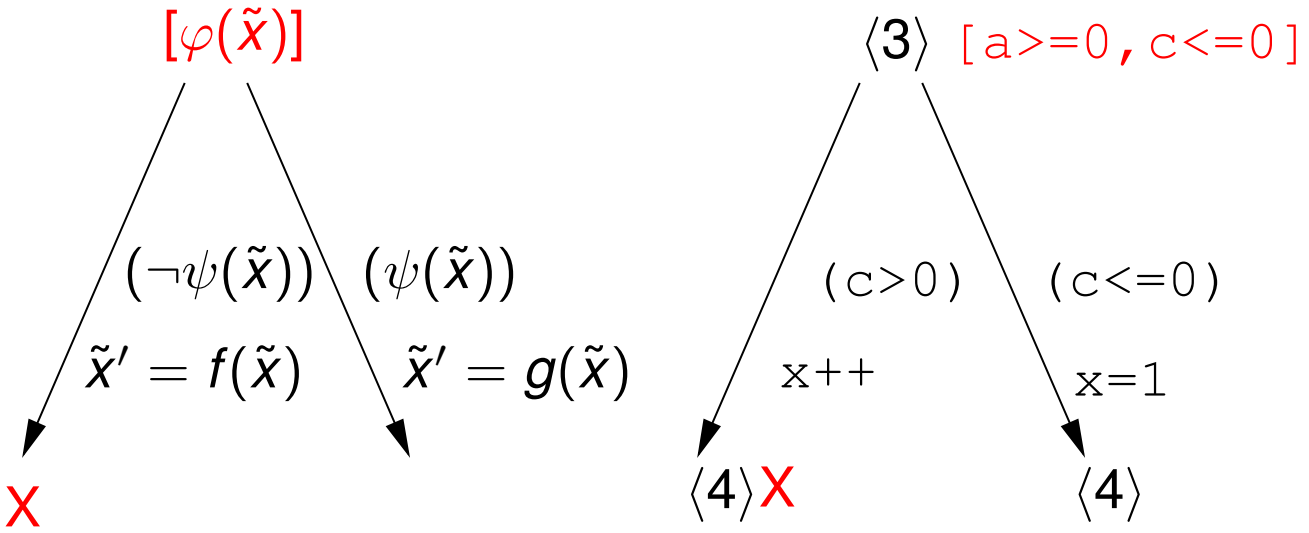
$$\langle 0\rangle\ \langle 1\rangle\ (\mathtt{a} > 0);\ \mathtt{c} = -\mathtt{a};\langle 2\rangle\ \langle 3\rangle\ (\mathtt{c} > 0);\ \langle 4\rangle$$

Logical View



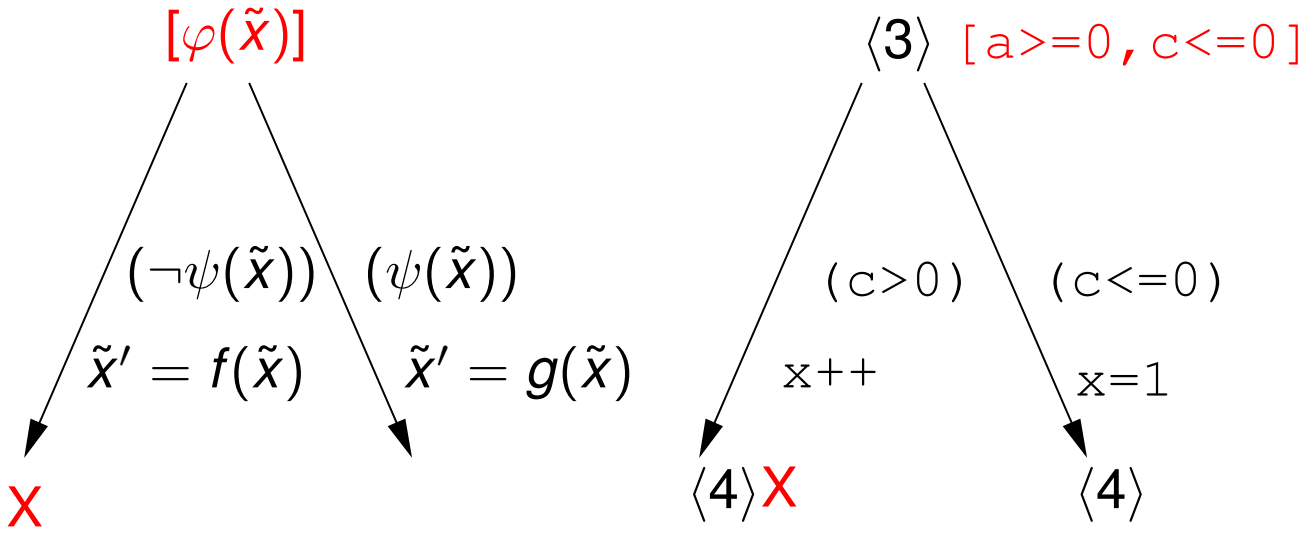
- *Craig interpolant* (e.g., on left path): A condition $\iota(\tilde{x})$ s.t. $\varphi(\tilde{x}) \Rightarrow \iota(\tilde{x})$ and $\iota(\tilde{x}) \Rightarrow (\neg\psi(\tilde{x}) \wedge \tilde{x}' = f(\tilde{x}) \Rightarrow \square)$
- Most general: path's *weakest liberal precondition:*
  Left: $\forall\tilde{x}' : \neg\psi(\tilde{x}) \wedge \tilde{x}' = f(\tilde{x}) \Rightarrow \square$ in example: $c \leq 0$
  Right: $\forall\tilde{x}' : \psi(\tilde{x}) \wedge \tilde{x}' = g(\tilde{x}) \Rightarrow \neg\square$ in example: $\neg\square$
  Conjunction is a *weakest precondition* of *feasible* paths:
  $c \leq 0 \equiv \exists x' : c \leq 0 \wedge x' = 1$
- Its *approximation* efficiently computed by *removal* of a>=0

Logical View



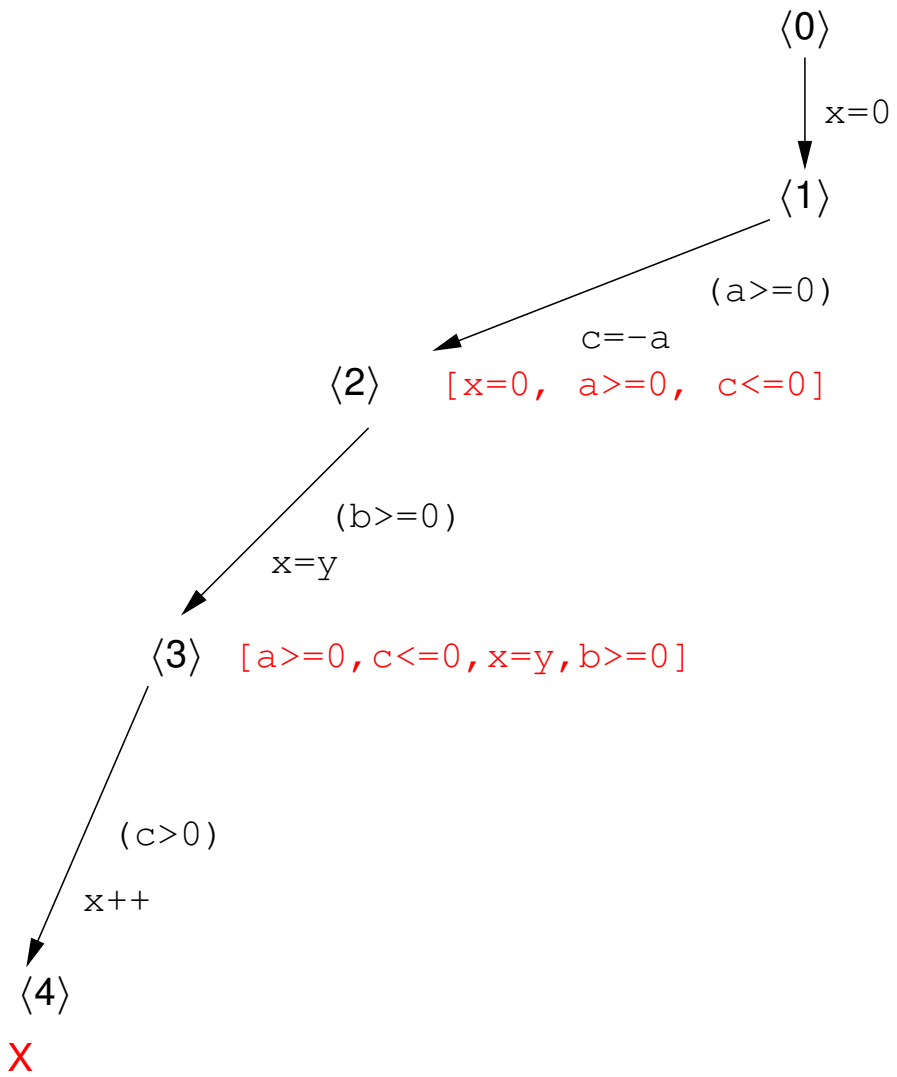
- *Craig interpolant* (e.g., on left path): A condition $\iota(\tilde{x})$ s.t. $\varphi(\tilde{x}) \Rightarrow \iota(\tilde{x})$ and $\iota(\tilde{x}) \Rightarrow (\neg\psi(\tilde{x}) \wedge \tilde{x}' = f(\tilde{x}) \Rightarrow \square)$
- Most general: path's *weakest liberal precondition:*
  Left: $\forall\tilde{x}' : \neg\psi(\tilde{x}) \wedge \tilde{x}' = f(\tilde{x}) \Rightarrow \square$ in example: $c \leq 0$
  Right: $\forall\tilde{x}' : \psi(\tilde{x}) \wedge \tilde{x}' = g(\tilde{x}) \Rightarrow \neg\square$ in example: $\neg\square$
  Conjunction is a *weakest precondition* of *feasible* paths:
  $c \leq 0 \equiv \exists x' : c \leq 0 \wedge x' = 1$
- Its *approximation* efficiently computed by *removal* of a>=0

Logical View

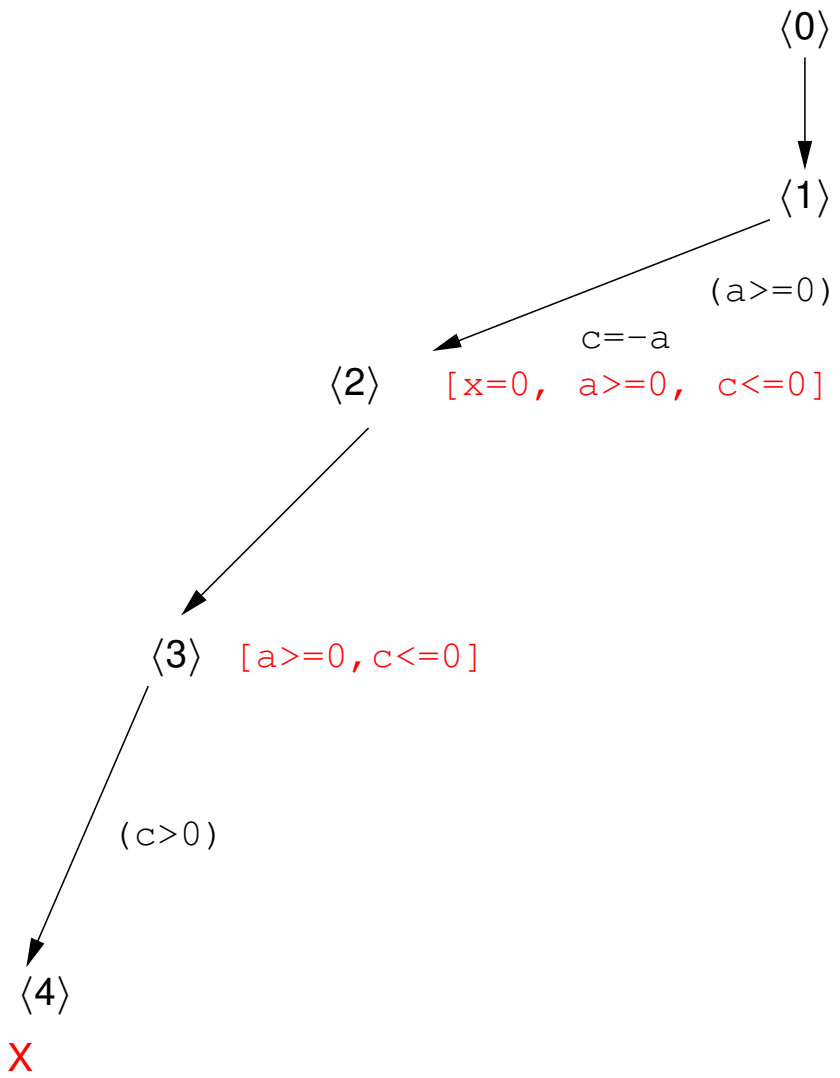


- *Craig interpolant* (e.g., on left path): A condition $\iota(\tilde{x})$ s.t. $\varphi(\tilde{x}) \Rightarrow \iota(\tilde{x})$ and $\iota(\tilde{x}) \Rightarrow (\neg\psi(\tilde{x}) \wedge \tilde{x}' = f(\tilde{x}) \Rightarrow \square)$
- Most general: path's *weakest liberal precondition:*
  Left: $\forall\tilde{x}' : \neg\psi(\tilde{x}) \wedge \tilde{x}' = f(\tilde{x}) \Rightarrow \square$ in example: $c \leq 0$
  Right: $\forall\tilde{x}' : \psi(\tilde{x}) \wedge \tilde{x}' = g(\tilde{x}) \Rightarrow \neg\square$ in example: $\neg\square$
  Conjunction is a *weakest precondition* of *feasible* paths:
  $c \leq 0 \equiv \exists x' : c \leq 0 \wedge x' = 1$
- Its *approximation* efficiently computed by *removal* of a>=0

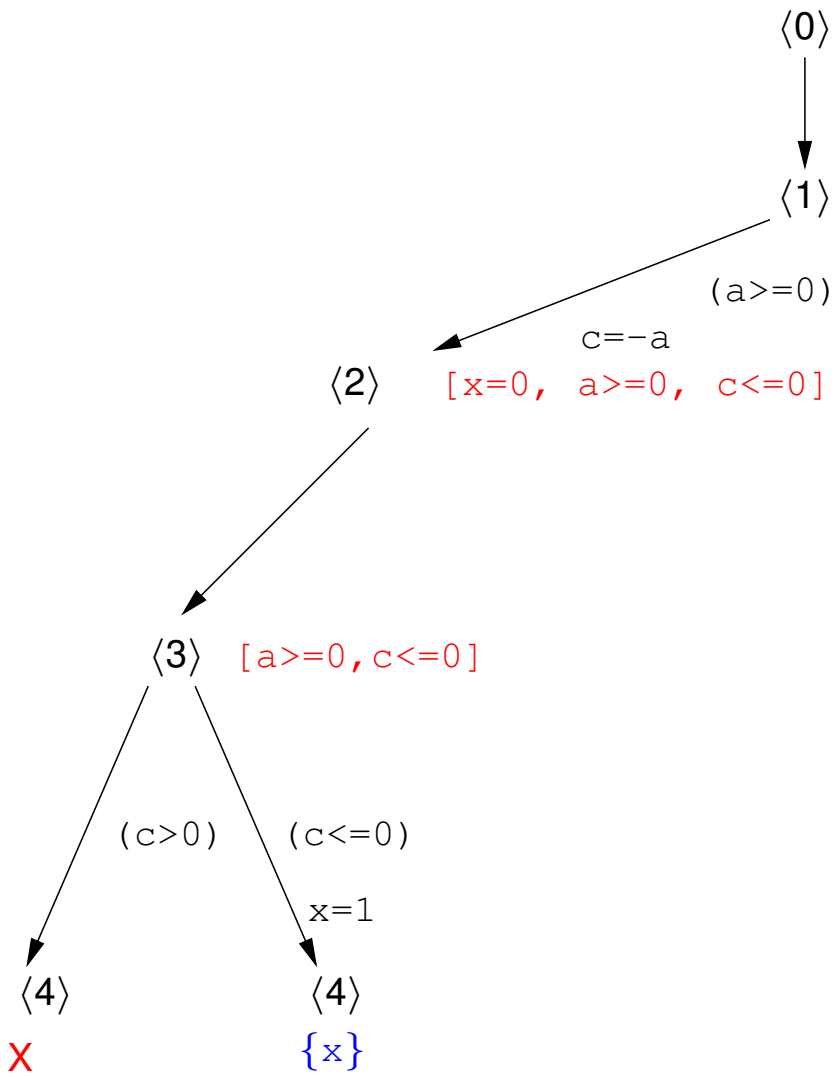
Efficient Path-Sensitive Slicing



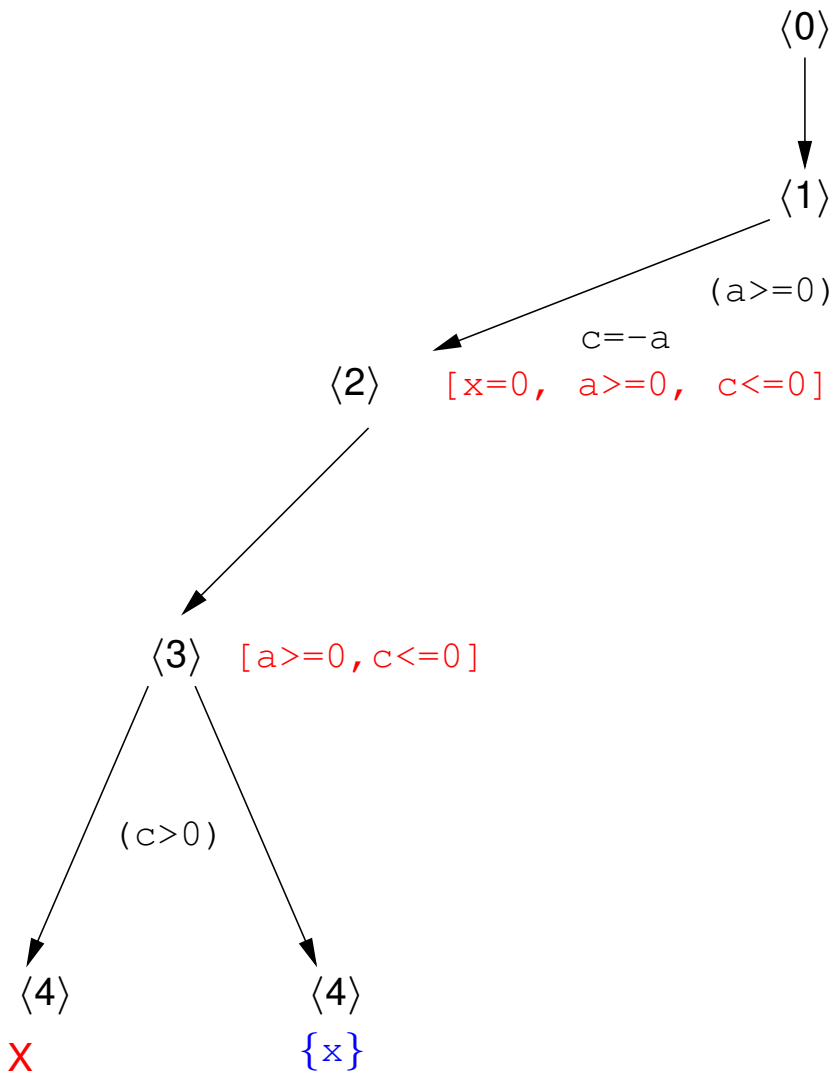
Efficient Path-Sensitive Slicing



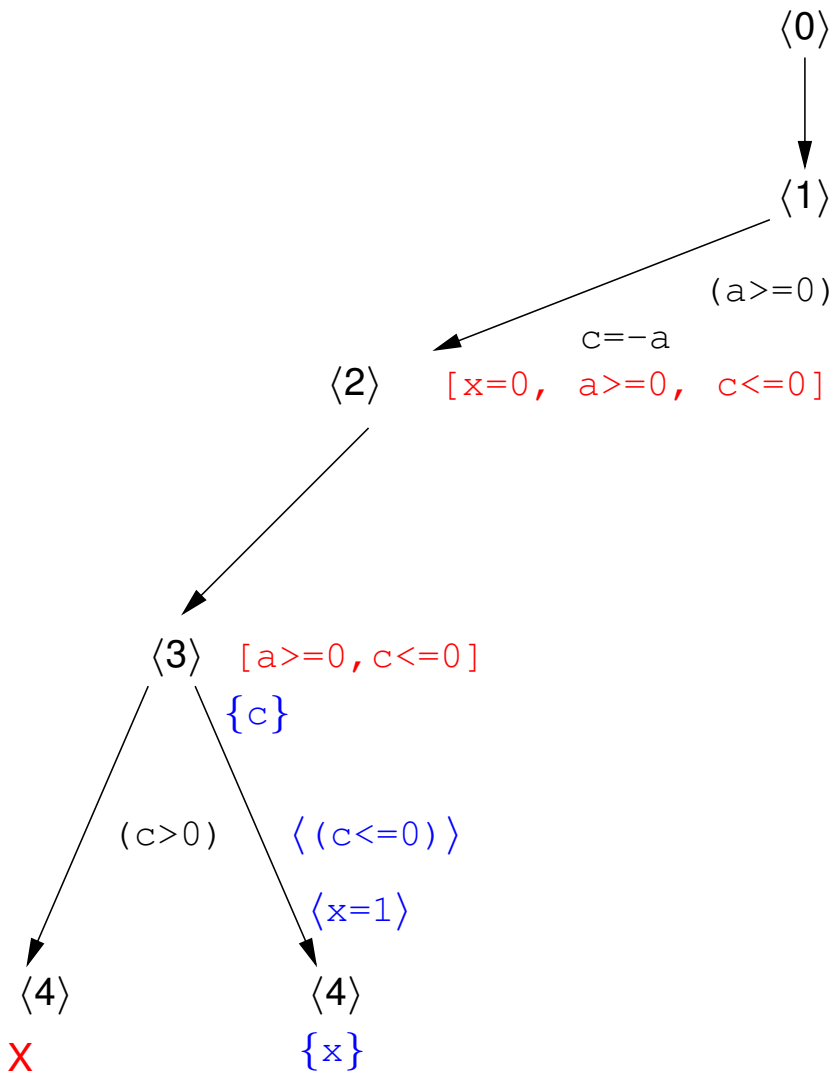
Efficient Path-Sensitive Slicing



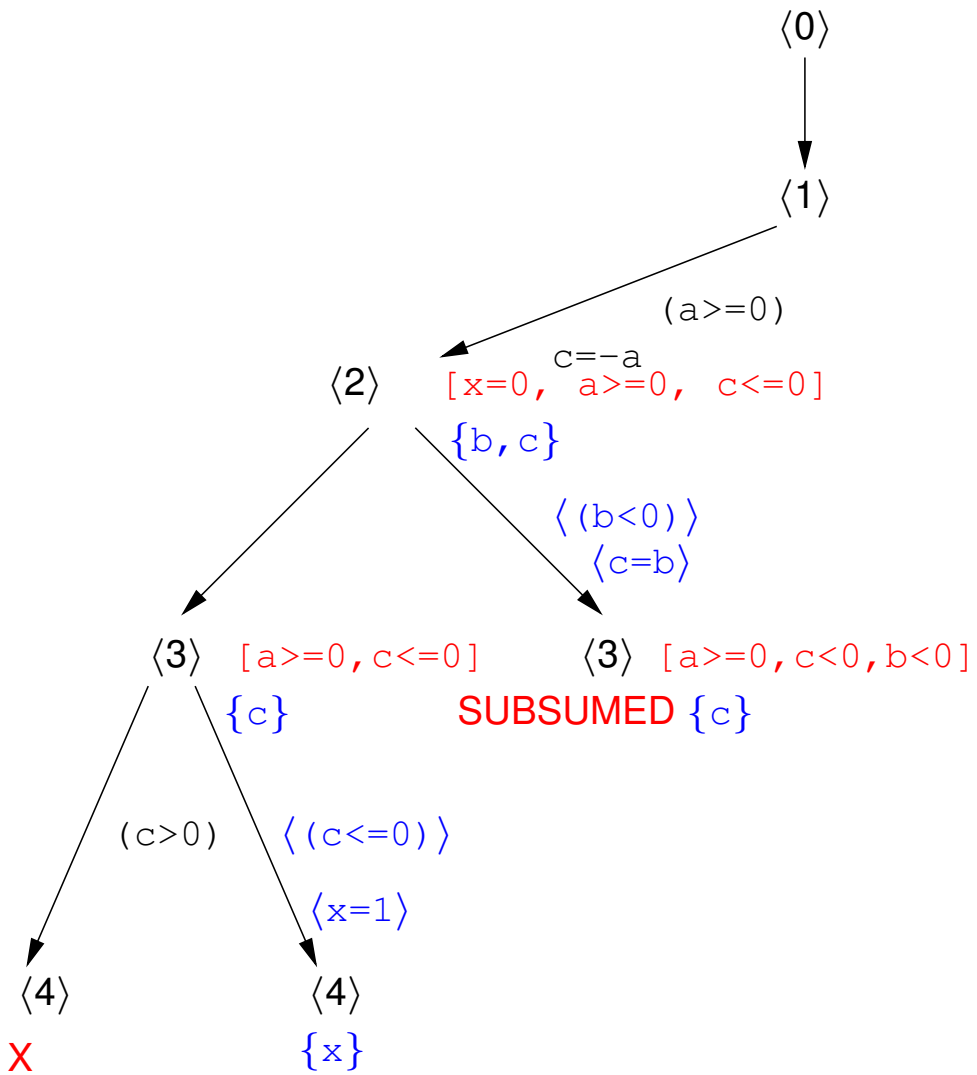
Efficient Path-Sensitive Slicing



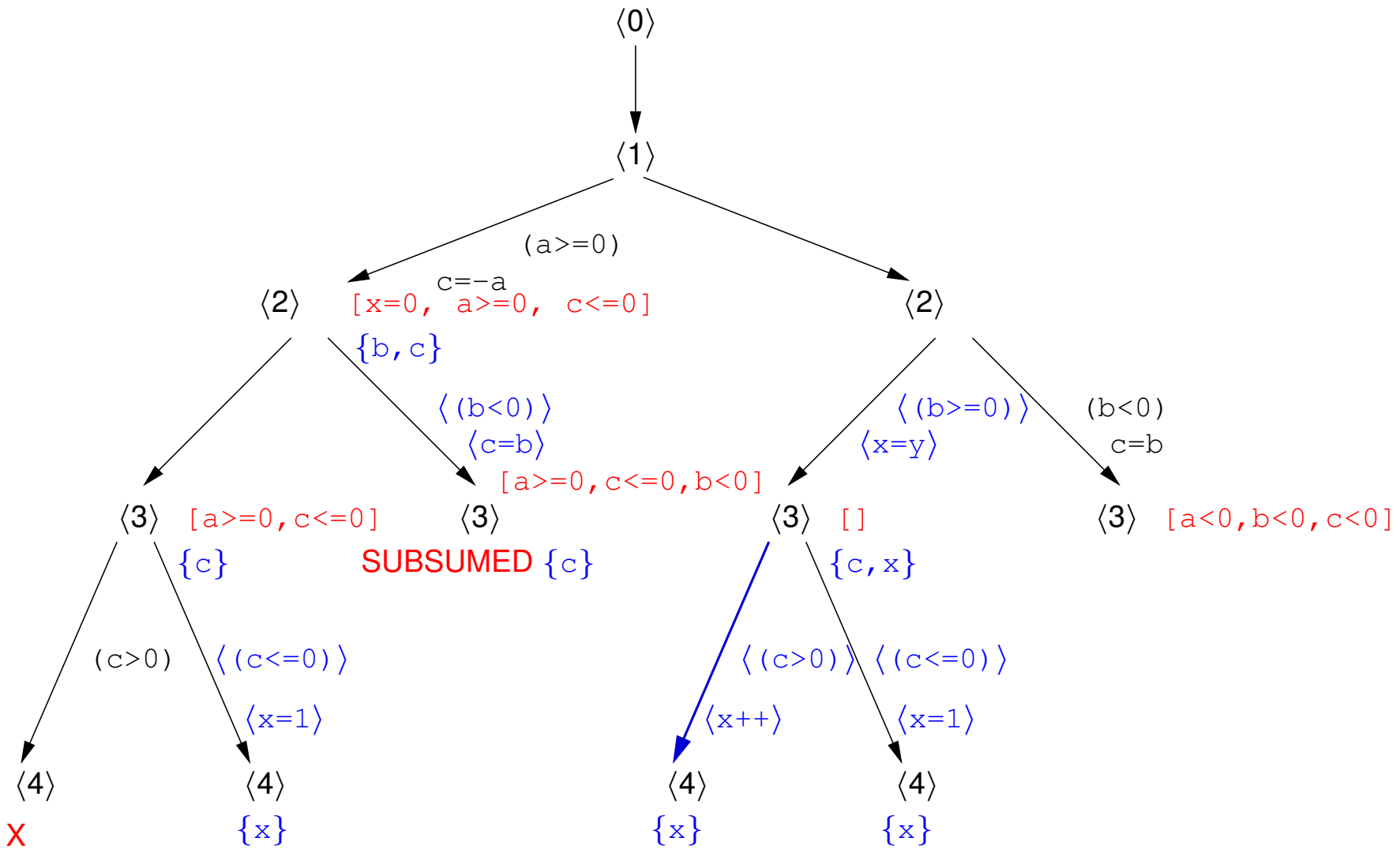
Efficient Path-Sensitive Slicing



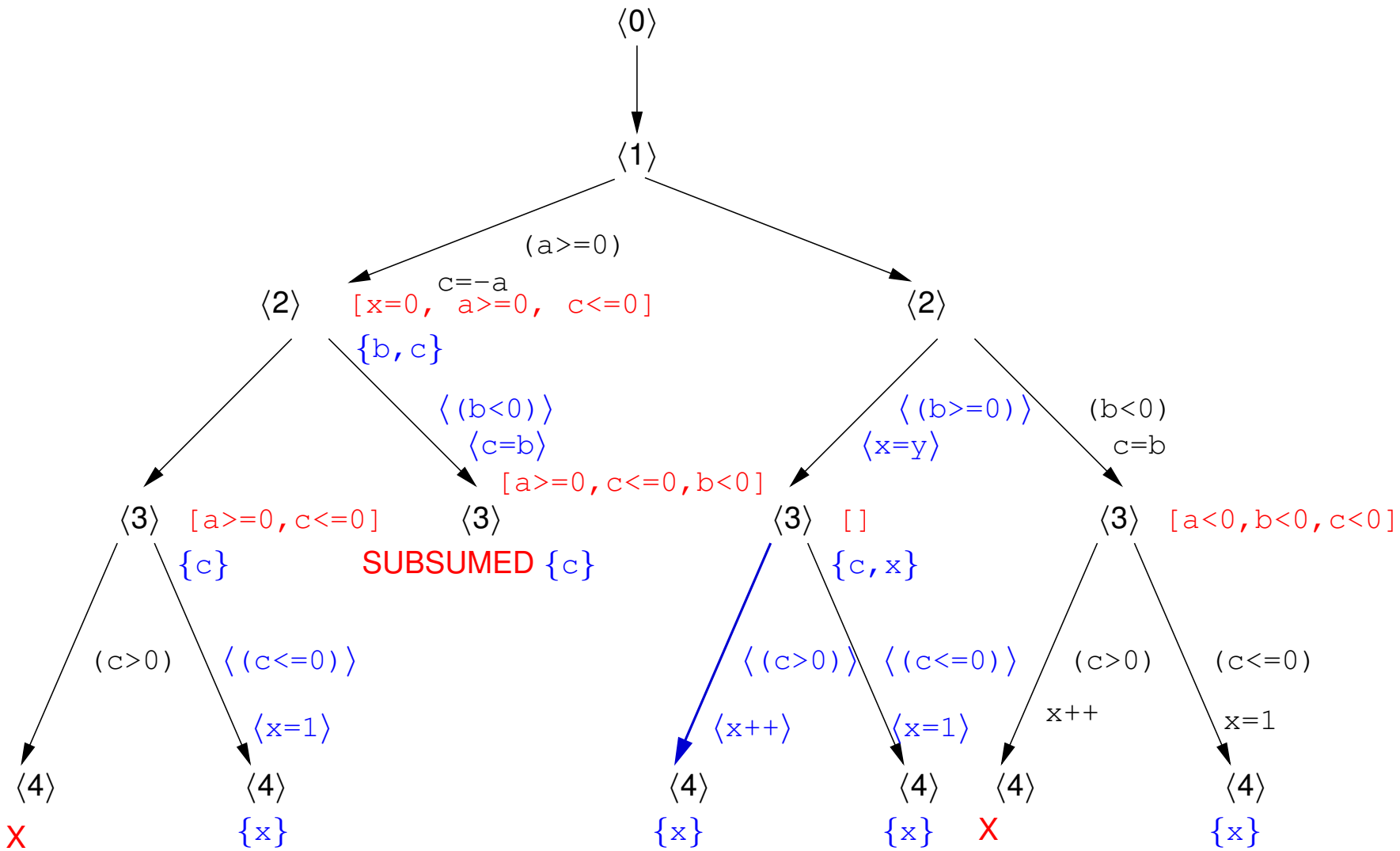
Efficient Path-Sensitive Slicing



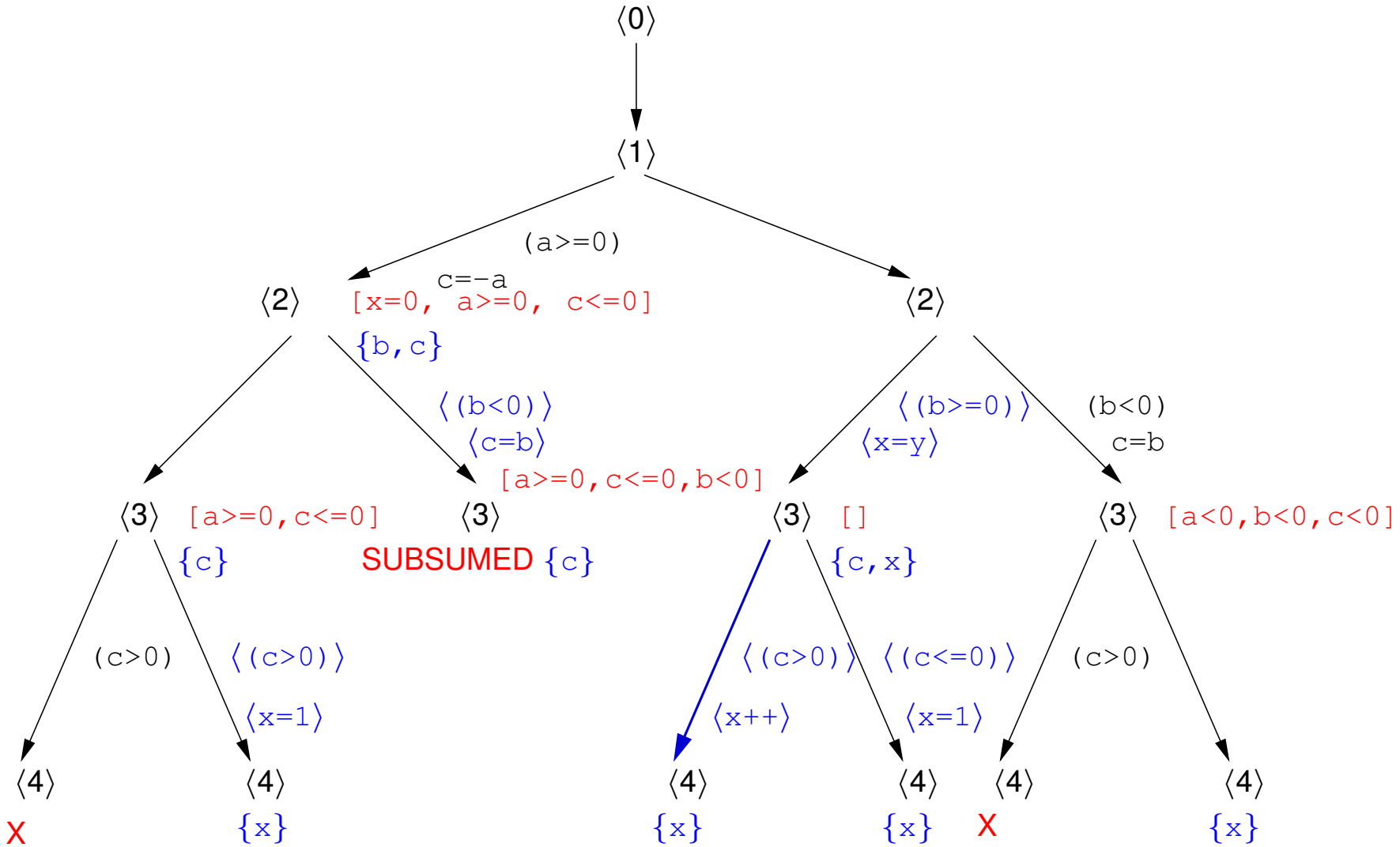
Efficient Path-Sensitive Slicing



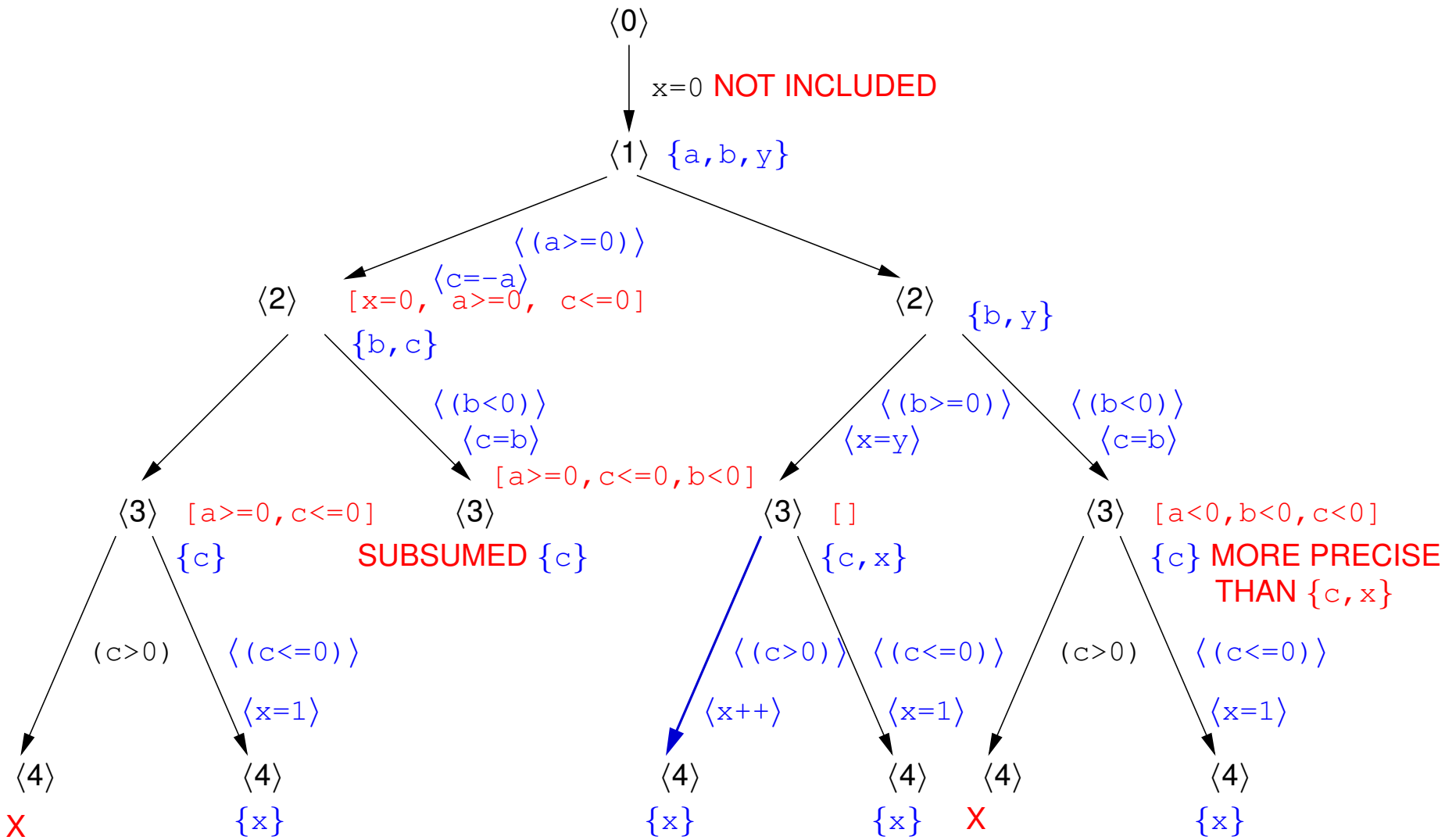
Efficient Path-Sensitive Slicing



Efficient Path-Sensitive Slicing



Efficient Path-Sensitive Slicing

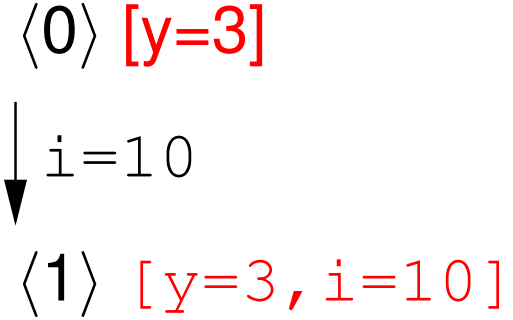


Loops

1. Initial pass to generate loop invariant via *abstract interpretation*.
   - Implementation uses **InterProc**
   - Example (next) uses removal of constraints of updated variables
2. The loop invariant used in symbolic execution
3. Fixpoint computation to generate backward dependencies

Loops

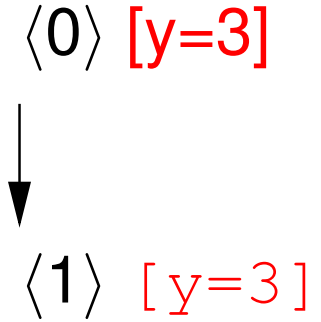
```
⟨0⟩ i=10;
⟨1⟩ while (i>0) {
⟨2⟩     if (x>y) x--;
        i--;
    } ⟨3⟩
```



Criterion:
⟨⟨3⟩,x⟩

Loops

```
⟨0⟩ i=10;
⟨1⟩ while (i>0) {
⟨2⟩     if (x>y) x--;
        i--;
    } ⟨3⟩
```

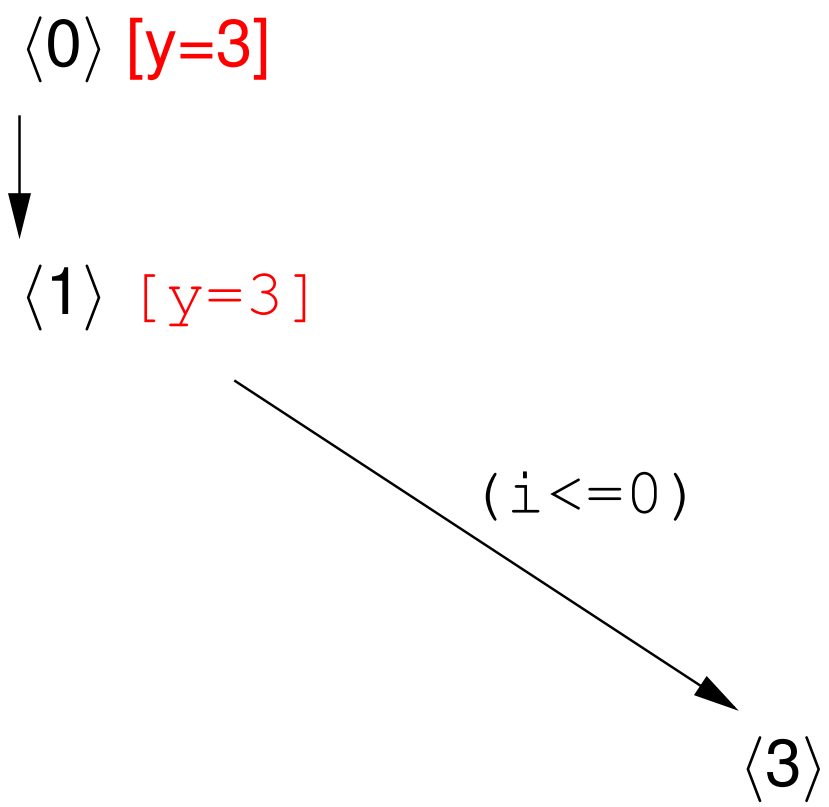


Criterion:
⟨⟨3⟩,x⟩

Loops

```
⟨0⟩ i=10;
⟨1⟩ while (i>0) {
⟨2⟩     if (x>y) x--;
        i--;
    } ⟨3⟩
```

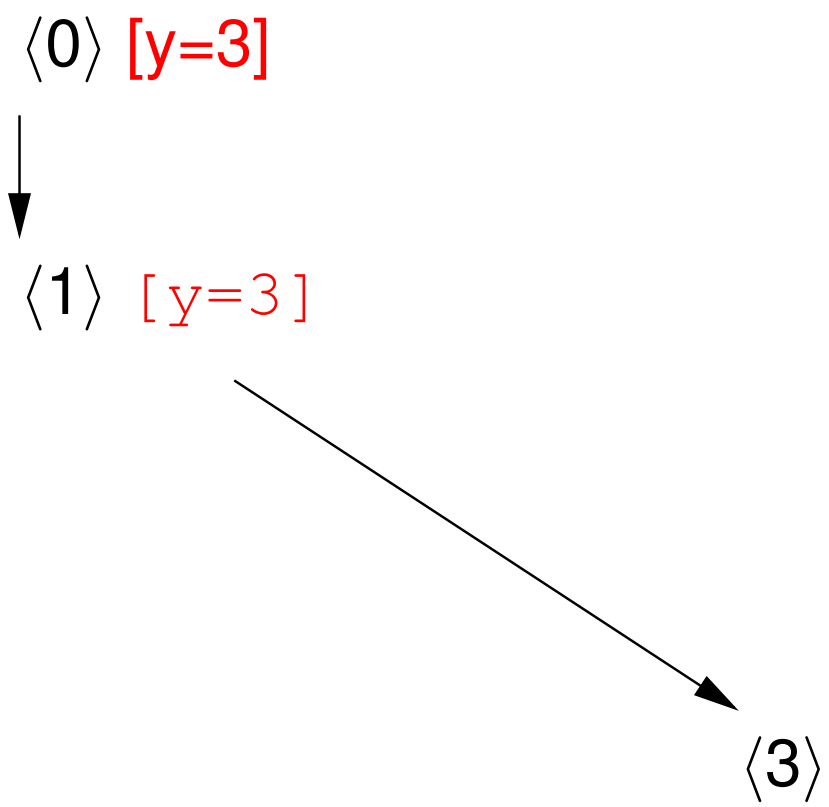
Criterion:
⟨⟨3⟩,x⟩



Loops

```
⟨0⟩ i=10;
⟨1⟩ while (i>0) {
⟨2⟩     if (x>y) x--;
        i--;
    } ⟨3⟩
```

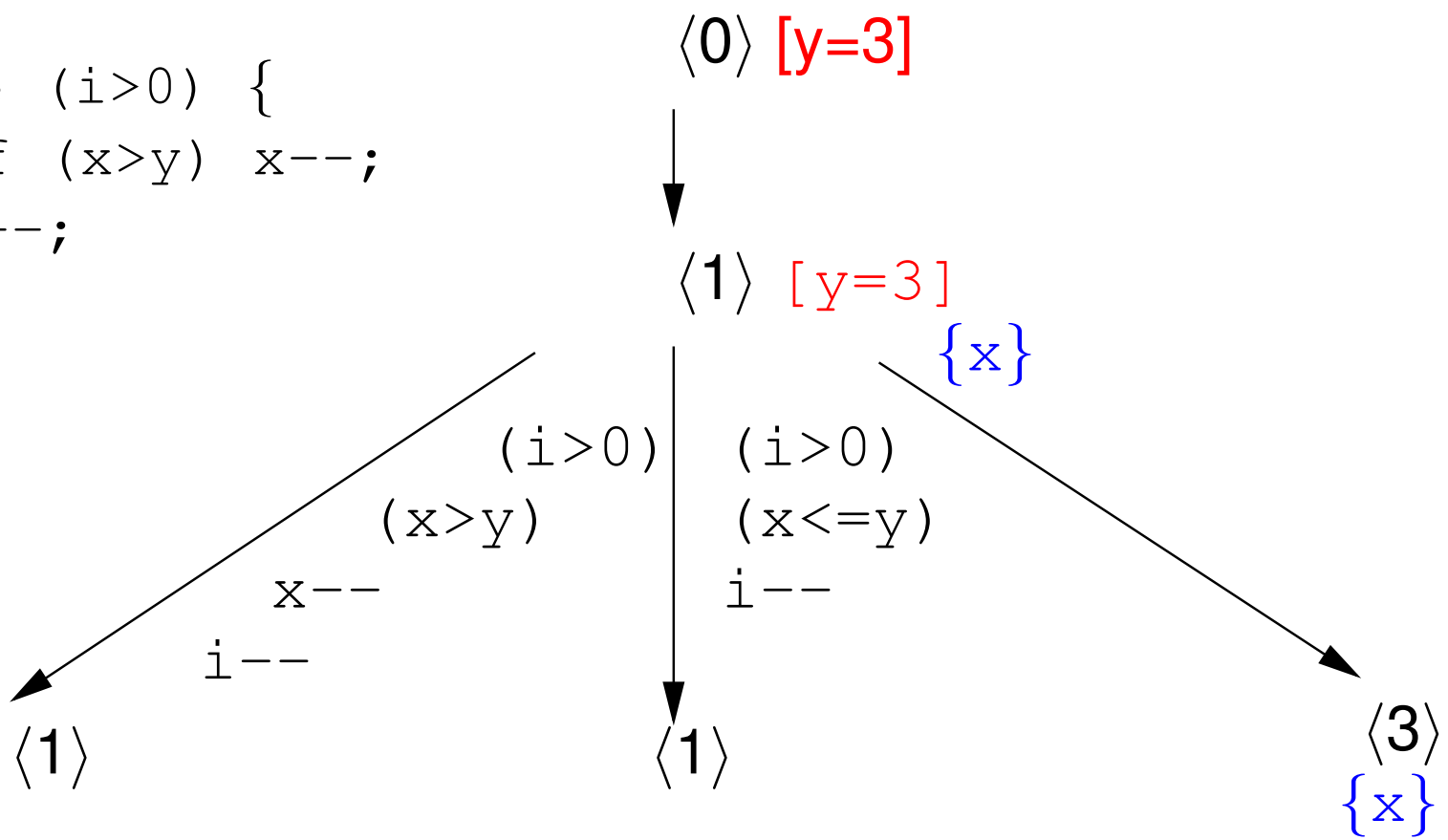
Criterion:
⟨⟨3⟩,x⟩



Loops

```
⟨0⟩ i=10;
⟨1⟩ while (i>0) {
⟨2⟩     if (x>y) x--;
        i--;
    } ⟨3⟩
```

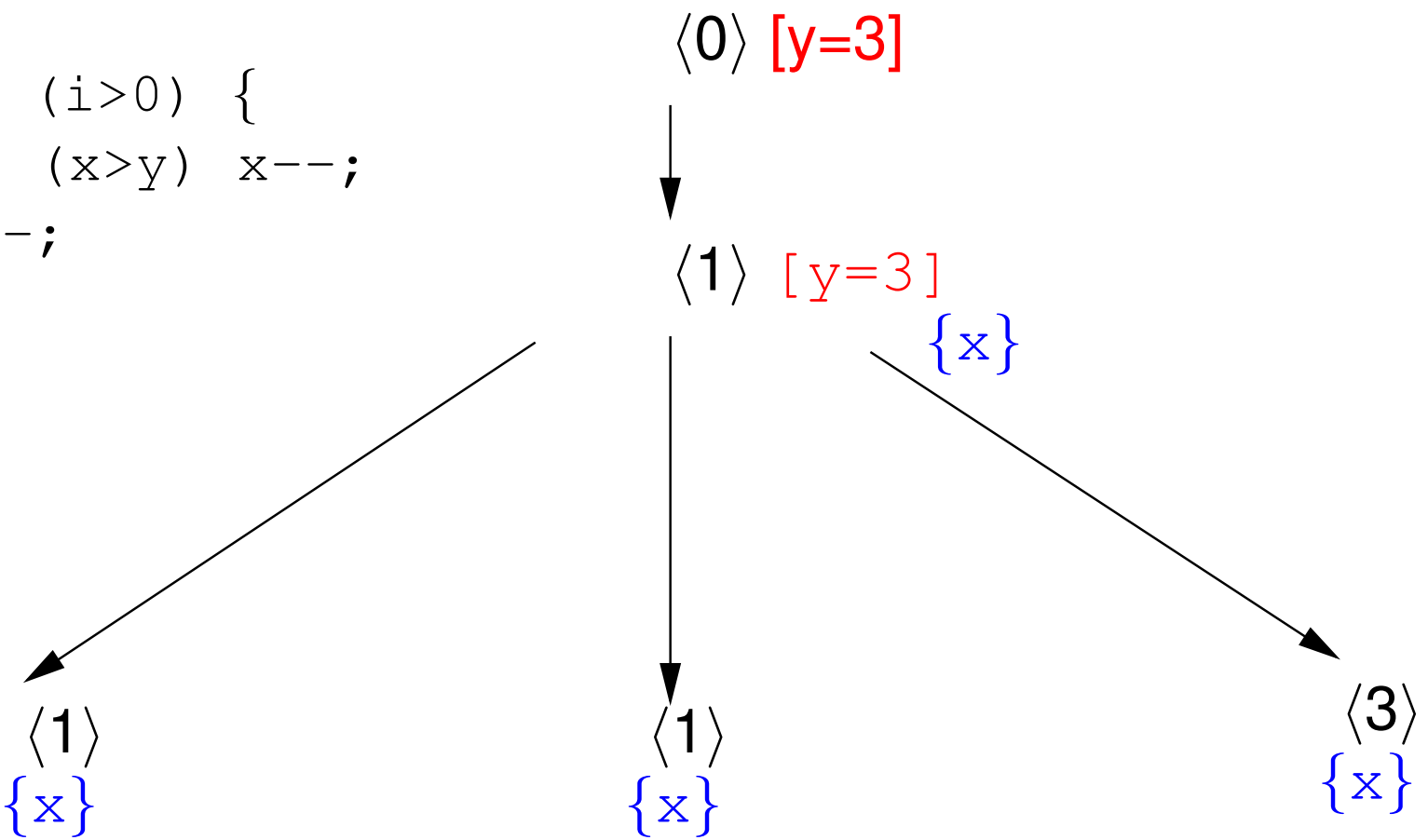
Criterion:
⟨⟨3⟩,x⟩



Loops

```
⟨0⟩ i=10;
⟨1⟩ while (i>0) {
⟨2⟩     if (x>y) x--;
        i--;
    } ⟨3⟩
```

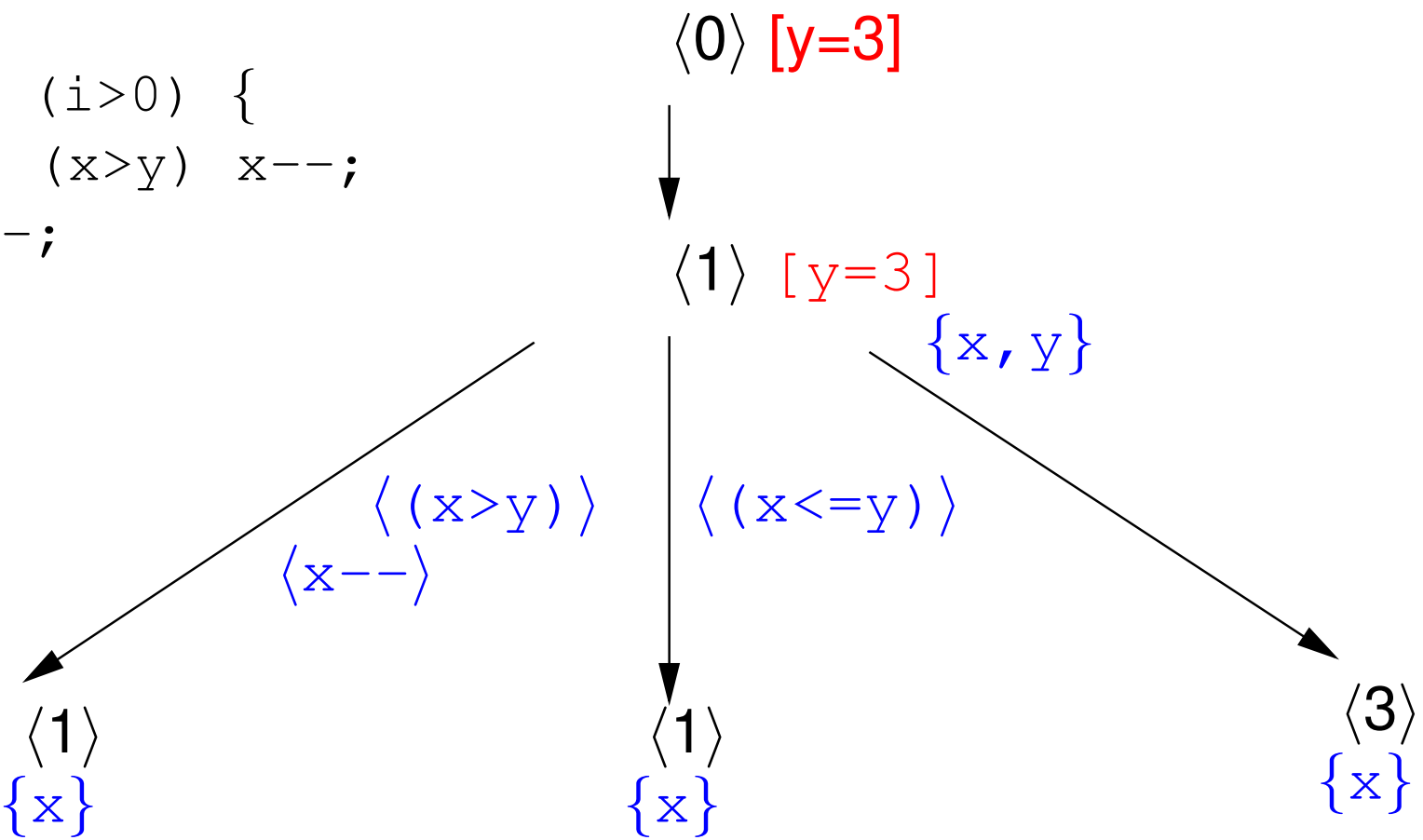
Criterion:
⟨⟨3⟩,x⟩



Loops

```
⟨0⟩ i=10;
⟨1⟩ while (i>0) {
⟨2⟩     if (x>y) x--;
        i--;
    } ⟨3⟩
```

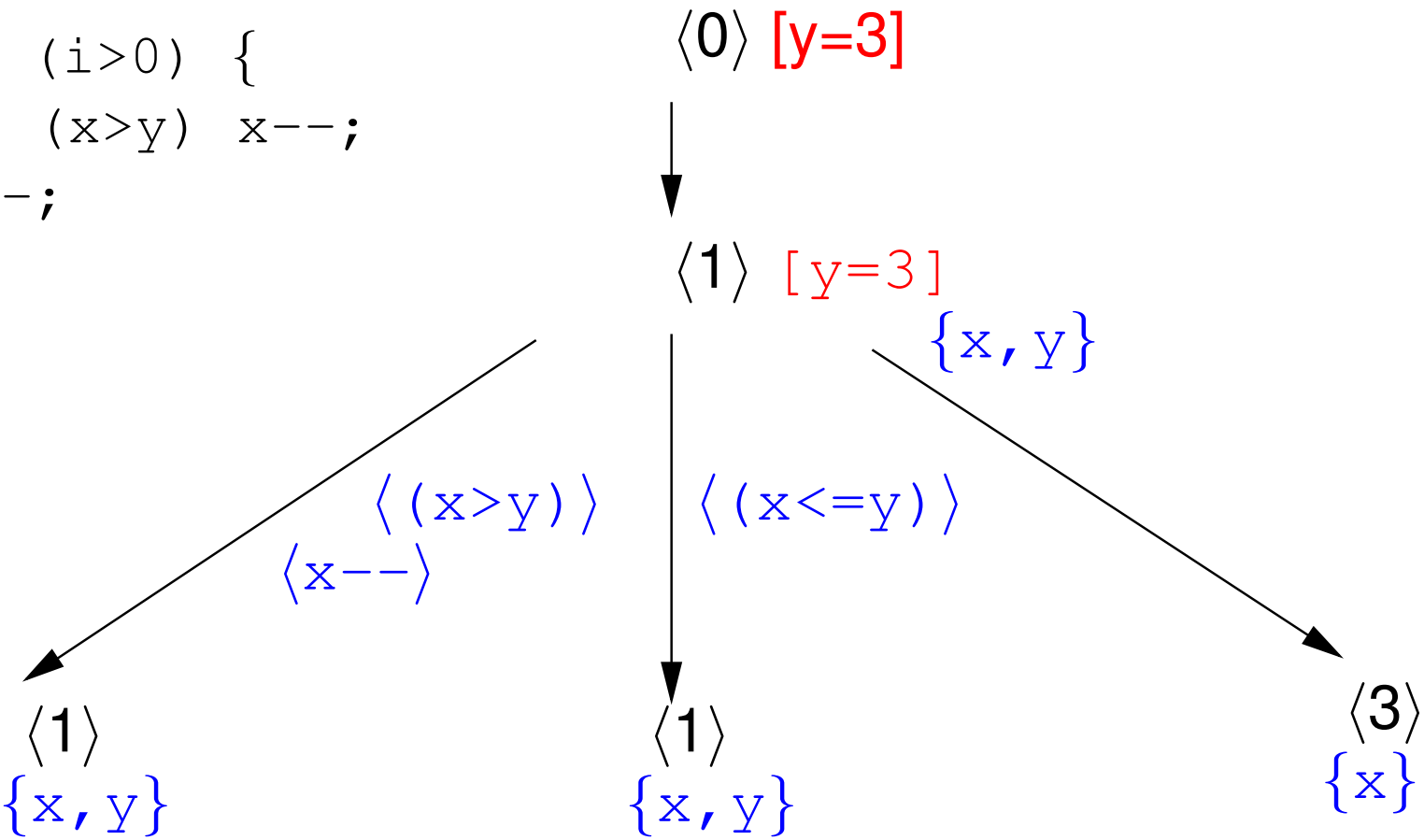
Criterion:
⟨⟨3⟩,x⟩



Loops

```
⟨0⟩ i=10;
⟨1⟩ while (i>0) {
⟨2⟩     if (x>y) x--;
        i--;
    } ⟨3⟩
```

Criterion:
⟨⟨3⟩,x⟩



Experimental Results

Program	LOC	Path-Insens		Path-Sens		
				Reuse		No Reuse
		Size Red	Time	Size Red	Time	Time
mpeg	5K	4%	21s	8%	628s	∞[1]
diskperf	6K	32%	2s	57%	94s	∞
floppy	8K	36%	9s	47%	263s	∞
cdaudio	9K	23%	10s	52%	301s	∞
serial	12K	39%	16s	50%	395s	∞
fcron.2.9.5	12K	42%	32s	61%	832s	∞
Mean		**23%**	**15s**	**38%**	**418s**	–

Results on Intel 3.2GHz 2Gb. [1] timeout after 2 hours or 2.5 Gb of memory consumption

References

- J. Jaffar, V. Murali, J. A. Navas, and A. E. Santosa. *Path-Sensitive Backward Slicing*. SAS '12
- J. Jaffar, V. Murali, J. A. Navas, and A. E. Santosa. *TRACER: A Symbolic Execution Tool for Verification*. CAV '12.
- J. Jaffar, J. A. Navas, and A. E. Santosa. *Unbounded Symbolic Execution for Program Verification*. Proc. RV '11.
- J. Jaffar, A. E. Santosa, and R. Voicu. *An Interpolation Method for CLP Traversal*. In 15th CP, Springer, 2009
- J. Jaffar, A. E. Santosa, and R. Voicu. *Efficient Memoization for Dynamic Programming with Ad-Hoc Constraints*. In 23rd AAAI, pages 297–303, AAAI Press, 2008

Related Work

- M. Weiser. *Program Slicing*. In ICSE '81
- Canfora, Cimitile, de Lucia. *Conditioned Program Slicing*. Information and Software Technology 40, 1998
- Danicic, Fox, Harman. *Consit: A Conditioned Program Slicer*. ICSM '00
- Daoudi et al. *Consus: A Scalable Approach to Conditioned Slicing*. Working Conference on Reverse Engineering, 2002.
  Scalable = infeasible path detection. This we already do
- Snelting. *Combining Slicing and Constraint Solving for Validation of Measurement Software*. SAS '96.
  Less precise: Conservatively includes a statement if there's at least one feasible path to the criterion point

Conclusion

- Mitigating state-space blowup using interpolation
- Application in path-sensitive slicing
- Applicable to other analyses
  - Program verification
  - Worst-case execution time (WCET)
  - Memory usage bound
  - Automatic symmetry reduction
  - Dynamic programming
  - Automatic parallelization (?)

Contact: santosa@it.usyd.edu.au