Witness Generation for Data-flow Analysis

Master's Thesis

Simmo Saan

University of Tartu, Institute of Computer Science

2 June 2020

Overview

Introduction

Static program analyzer:

- Input program
- Input specification — e.g., does not reach error

Output satisfies/violates specification

Introduction

Static program analyzer:

- Input program
- Input specification — e.g., does not reach error

Output satisfies/violates specification

Problems with boolean output

- Unhelpful does not say where and why violates
- Untrustworthy analyzer itself may contain bugs

Witnesses

- Proof objects — key information for proof:

  Violation witness error path

  Correctness witness invariants

- Formalized as automata

Witnesses

- Proof objects — key information for proof:

  Violation witness error path

  Correctness witness invariants

- Formalized as automata

Benefits of witnesses

Helpful understandable by programmers

Trustworthy automatically checkable by independent tools

Comparable Competition on Software Verification (SV-COMP)

Data-flow analysis

- Static analysis technique
  - Solve constraint system over complete lattice
- Overapproximation of all program paths
  - Efficient
  - Well-suited for certain problems
- Used by Goblint
  - Multi-threaded C programs
  - University of Tartu, Technical University of Munich

Data-flow analysis

- Static analysis technique
  - Solve constraint system over complete lattice
- Overapproximation of all program paths
  - Efficient
  - Well-suited for certain problems
- Used by Goblint
  - Multi-threaded C programs
  - University of Tartu, Technical University of Munich

Problem with data-flow analysis

- Computation does not correspond to required witness

Goal and contribution

- Design methods for generating witnesses with data-flow analysis
- Implement and evaluate them in Goblint

Abstract reachability graphs (ARGs)

- Directed graph (automaton) of reachable abstract program states
- Internal representation of many analyzers
- Easily convertible to witnesses

Abstract reachability graphs (ARGs)

- Directed graph (automaton) of reachable abstract program states
- Internal representation of many analyzers
- Easily convertible to witnesses

Problems with ARGs

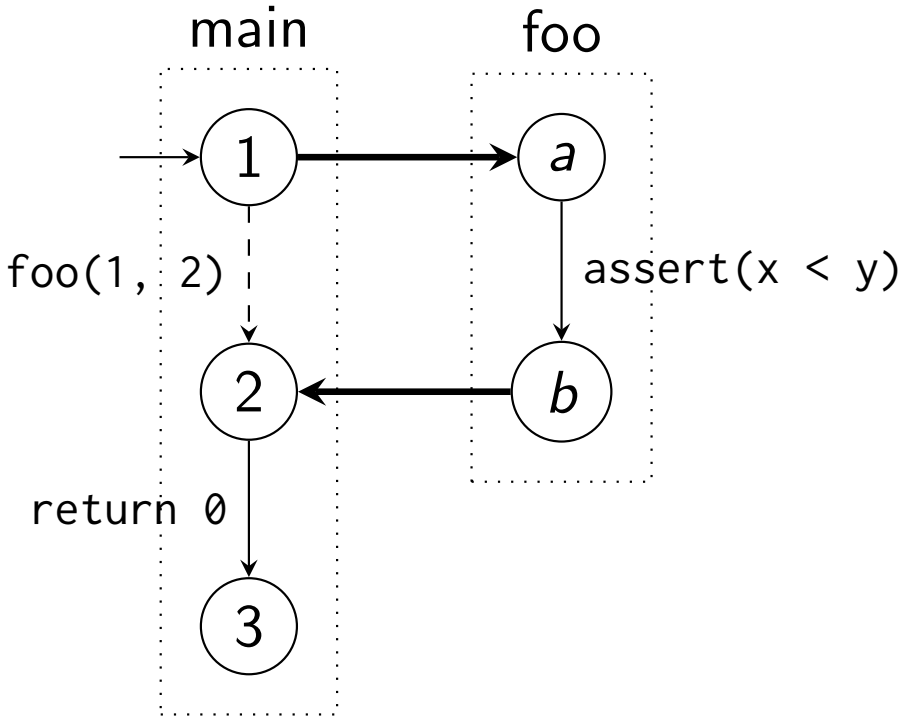
- Not used in data-flow analysis
- Function calls must be inlined

Interprocedural ARGs

Basic inlining

```
void foo(int x, int y)
{
    assert(x < y);
}

int main()
{
    foo(1, 2);
    return 0;
}
```

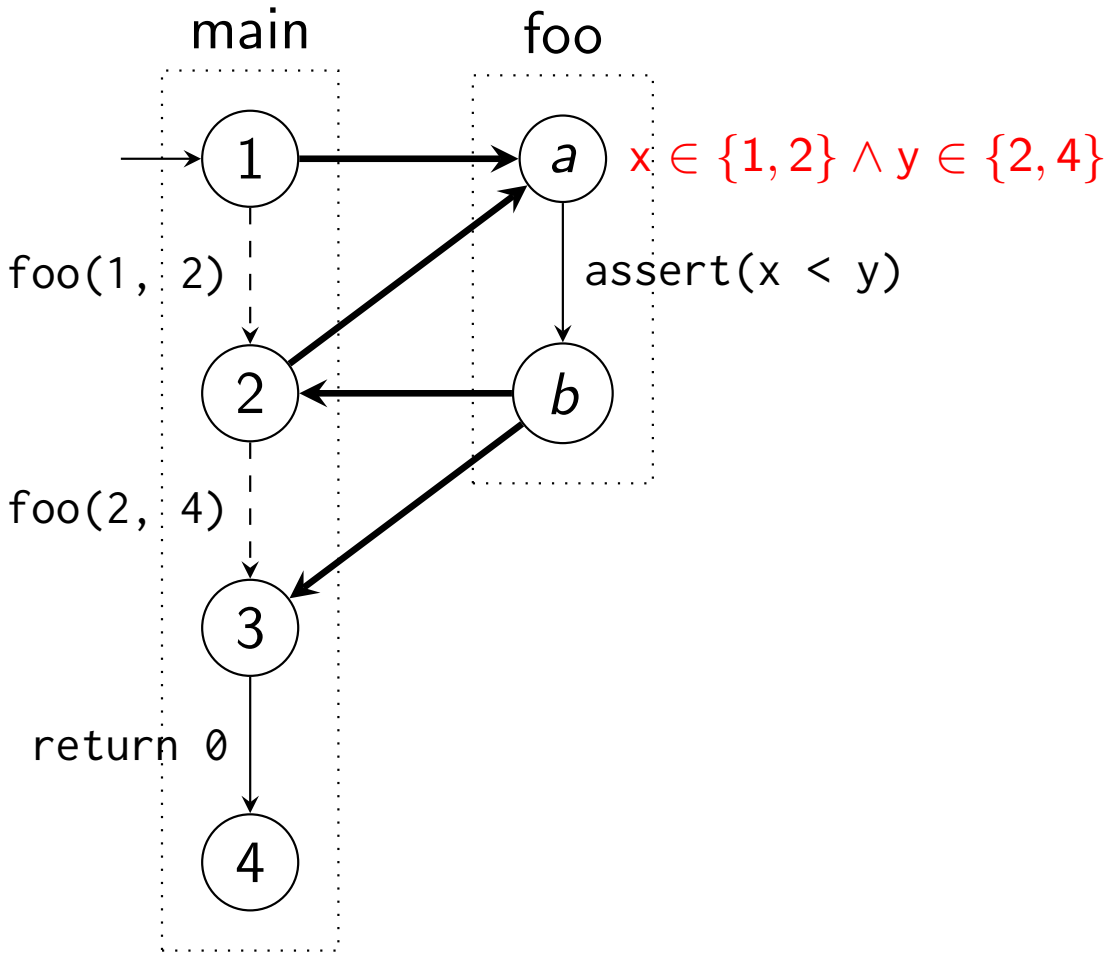


Interprocedural ARGs

Context-sensitive inlining

```
void foo(int x, int y)
{
    assert(x < y);
}

int main()
{
    foo(1, 2);
    foo(2, 4); // added
    return 0;
}
```

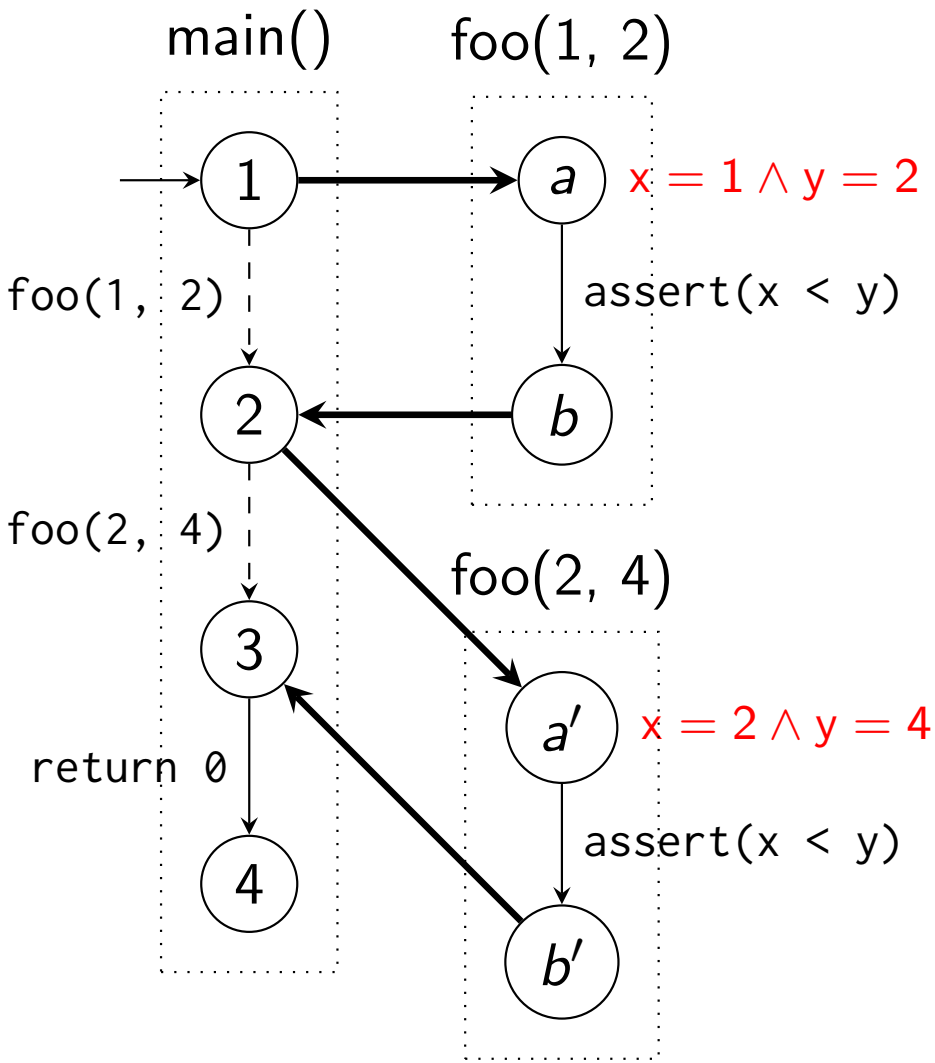


Interprocedural ARGs

Context-sensitive inlining

```
void foo(int x, int y)
{
    assert(x < y);
}

int main()
{
    foo(1, 2);
    foo(2, 4); // added
    return 0;
}
```



ARGs from interprocedural data-flow analysis

1. Context-sensitively inlined ARG
   - Sound extension of constraint system
   - Record context-sensitive predecessor relationship
   - Gives reachability information

ARGs from interprocedural data-flow analysis

1. Context-sensitively inlined ARG
   - Sound extension of constraint system
   - Record context-sensitive predecessor relationship
   - Gives reachability information

2. Call-site–sensitively inlined ARG
   - Postprocess previous ARG
   - Ensures unique entering and returning transitions

Witness analysis

Original constraint system:

$$
\begin{aligned}
[\mathsf{entry}_{\mathsf{main}}, c_{\mathsf{main}}] &\sqsupseteq d_{\mathsf{main}} \\
[v, c] &\sqsupseteq [\![s]\!]^\sharp\,(\mathsf{get}\,[u, c]) && \forall u \xrightarrow{s} v \\
[v, c] &\sqsupseteq \mathbf{let}\,(c', d') = \mathsf{enter}^\sharp\,(\mathsf{get}\,[u, c]) && \forall u \xrightarrow{f()} v \\
&\qquad\quad () = \mathsf{set}\,[\mathsf{entry}_f, c']\ \ d' \\
&\qquad\ \mathbf{in}\ \ \mathsf{combine}^\sharp\,(\mathsf{get}\,[u, c])\ \ (\mathsf{get}\,[\mathsf{return}_f, c'])
\end{aligned}
$$

Witness analysis

Extended constraint system:

$$[\mathsf{entry}_{\mathsf{main}}, c_{\mathsf{main}}] \sqsupseteq (d_{\mathsf{main}}, \varnothing)$$

$$[v, c] \sqsupseteq (\llbracket s \rrbracket^\sharp \langle \mathsf{get}\,[u, c] \rangle_1, \{([u, c], s)\}) \qquad \forall u \xrightarrow{s} v$$

$$[v, c] \sqsupseteq \mathbf{let}\,(c', d') = \mathsf{enter}^\sharp \langle \mathsf{get}\,[u, c] \rangle_1 \qquad \forall u \xrightarrow{f()} v$$
$$() = \mathsf{set}\,[\mathsf{entry}_f, c']\,(d', \{([u, c], \mathsf{enterFunction}_f)\})$$
$$\mathbf{in}\,(\mathsf{combine}^\sharp \langle \mathsf{get}\,[u, c] \rangle_1 \langle \mathsf{get}\,[\mathsf{return}_f, c'] \rangle_1,$$
$$\{([\mathsf{return}_f, c'], \mathsf{returnFromFunction}_f)\})$$

In every reached context c:

- Every reached $u \xrightarrow{s} v$ gives $[u, c] \xrightarrow{s} [v, c]$
- Every reached $u \xrightarrow{f()} v$ gives

$$[u, c] \xrightarrow{\mathsf{enterFunction}_f} [\mathsf{entry}_f, c'] \to \ldots \to [\mathsf{return}_f, c'] \xrightarrow{\mathsf{returnFromFunction}_f} [v, c]$$

Evaluation

On Goblint implementation:

- Manual testing
  - Small crafted programs
  - Manual witness inspection
  - Confirms correct context- and call-site–sensitive inlining

Evaluation

On Goblint implementation:

- Manual testing
  - Small crafted programs
  - Manual witness inspection
  - Confirms correct context- and call-site–sensitive inlining

- SV-COMP testing
  - "SoftwareSystems-DeviceDriversLinux64-ReachSafety" category
  - Confirms acceptance by independent validators

Evaluation

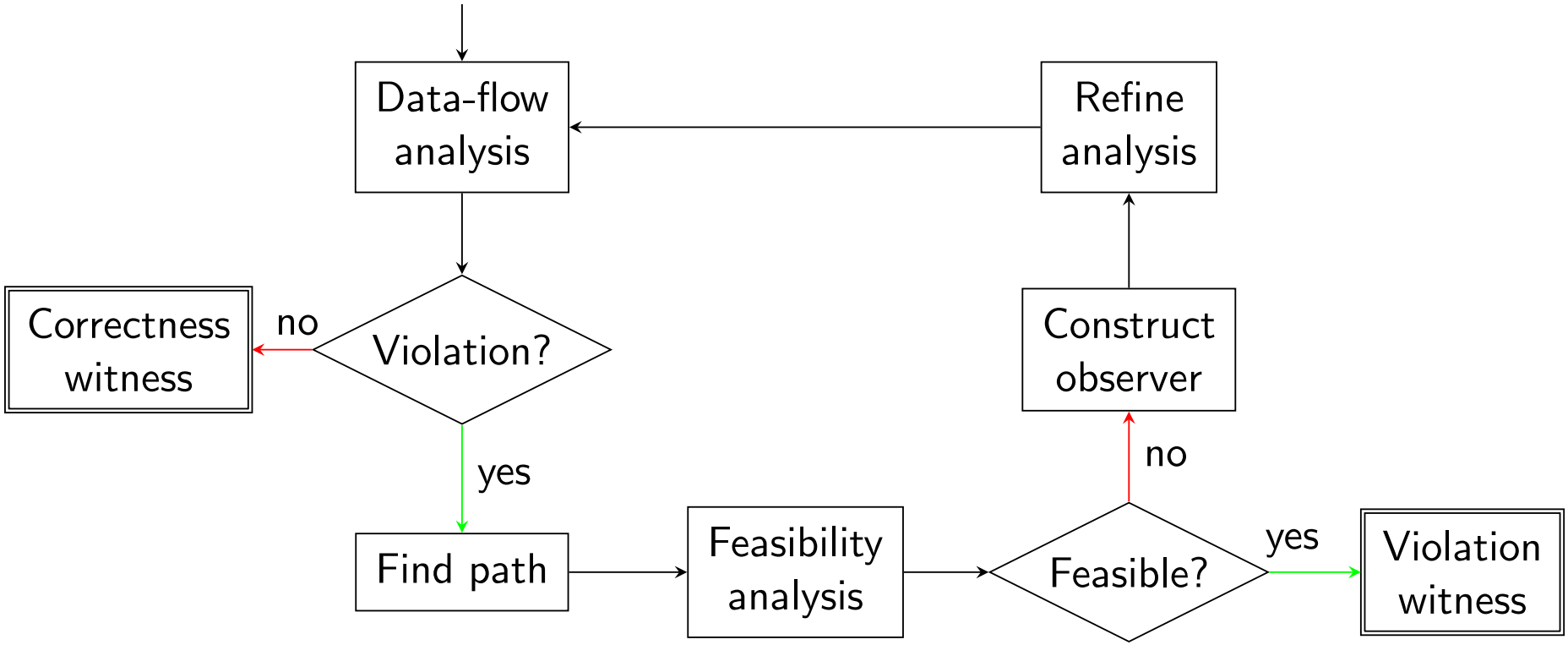
On Goblint implementation:

- Manual testing
  - Small crafted programs
  - Manual witness inspection
  - Confirms correct context- and call-site–sensitive inlining
- SV-COMP testing
  - "SoftwareSystems-DeviceDriversLinux64-ReachSafety" category
  - Confirms acceptance by independent validators

Problems revealed by SV-COMP testing

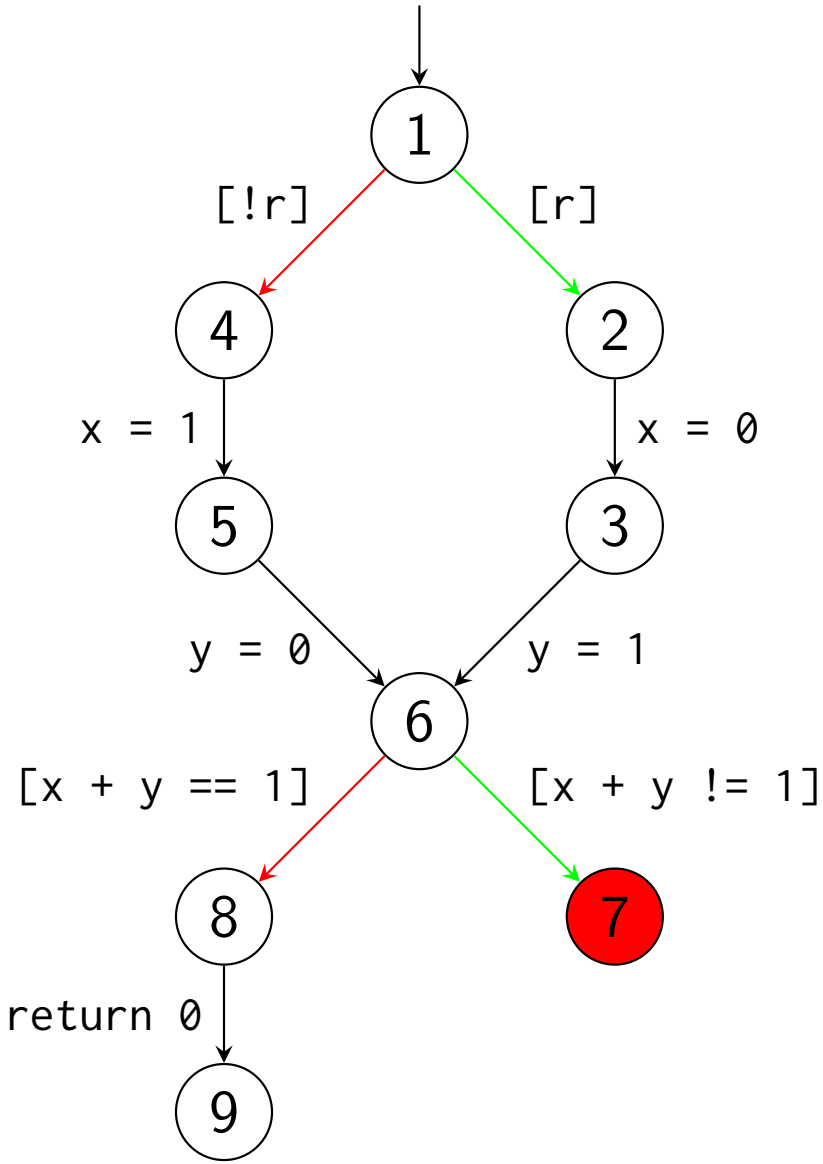
- Many false positive violations
- Imprecise (violation) witnesses

Refinement loop algorithm



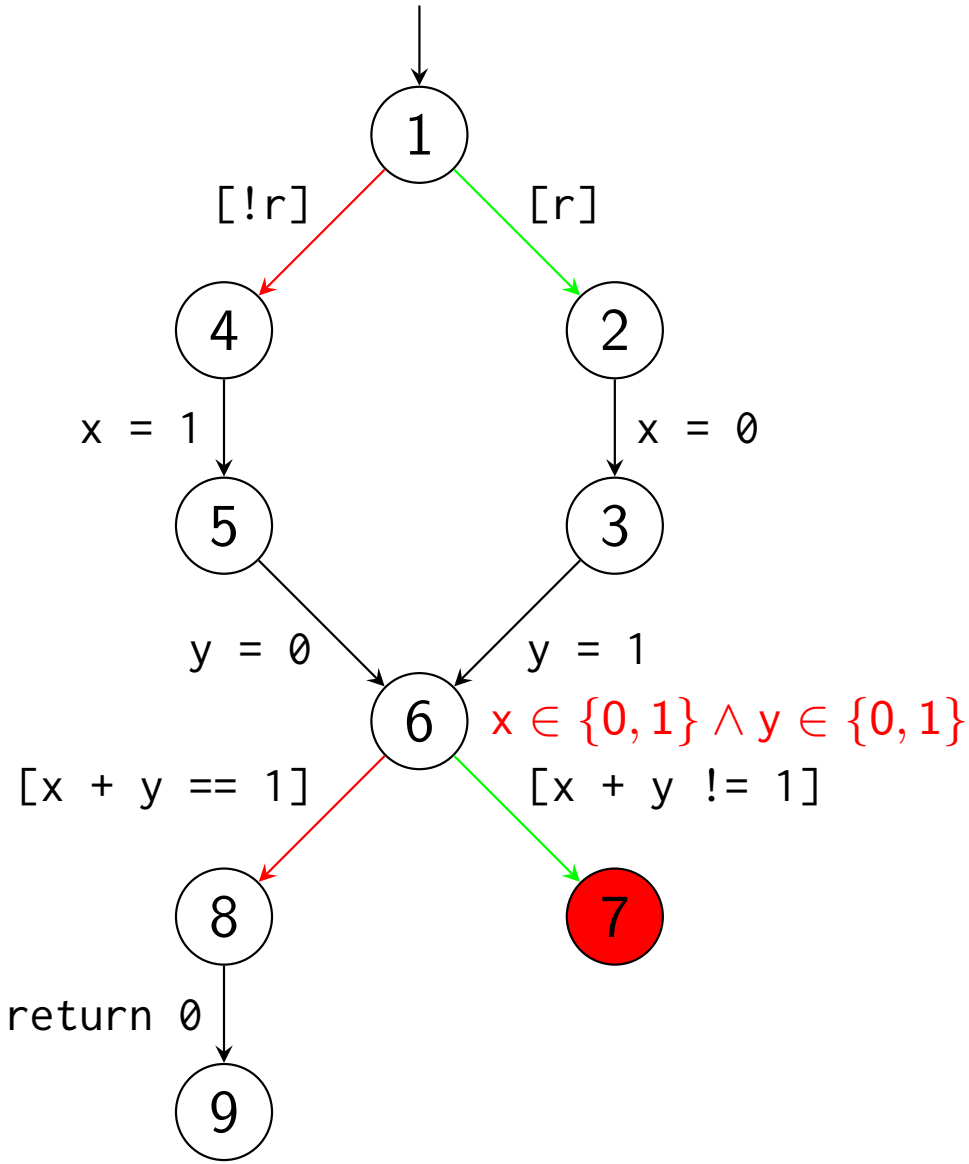
Refinement example

```c
int main()
{
    int r, x, y;
    if (r)
    {
        x = 0;
        y = 1;
    }
    else
    {
        x = 1;
        y = 0;
    }
    if (x + y != 1)
        error();
    return 0;
}
```



Refinement example

```
int main()
{
    int r, x, y;
    if (r)
    {
        x = 0;
        y = 1;
    }
    else
    {
        x = 1;
        y = 0;
    }
    if (x + y != 1)
        error();
    return 0;
}
```

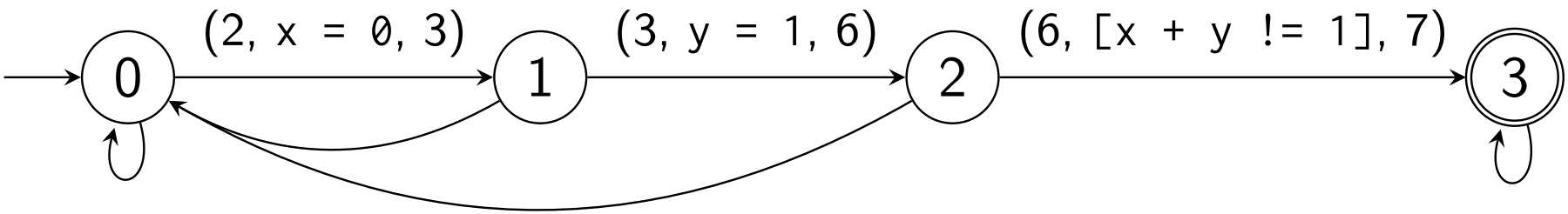


Refinement example — observer

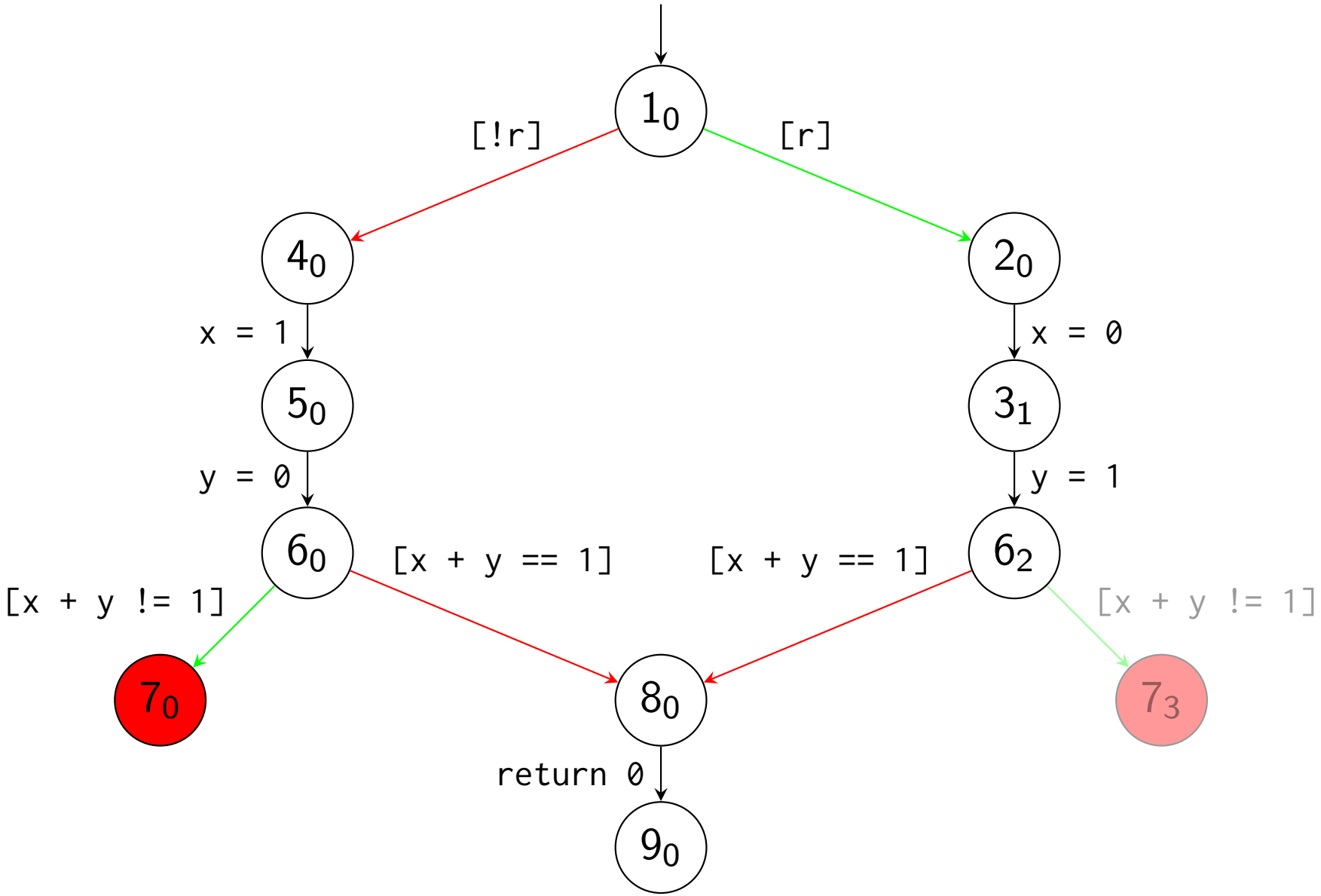
Infeasible program subpath:

$$2 \xrightarrow{\texttt{x = 0}} 3 \xrightarrow{\texttt{y = 1}} 6 \xrightarrow{\texttt{[x + y != 1]}} 7$$

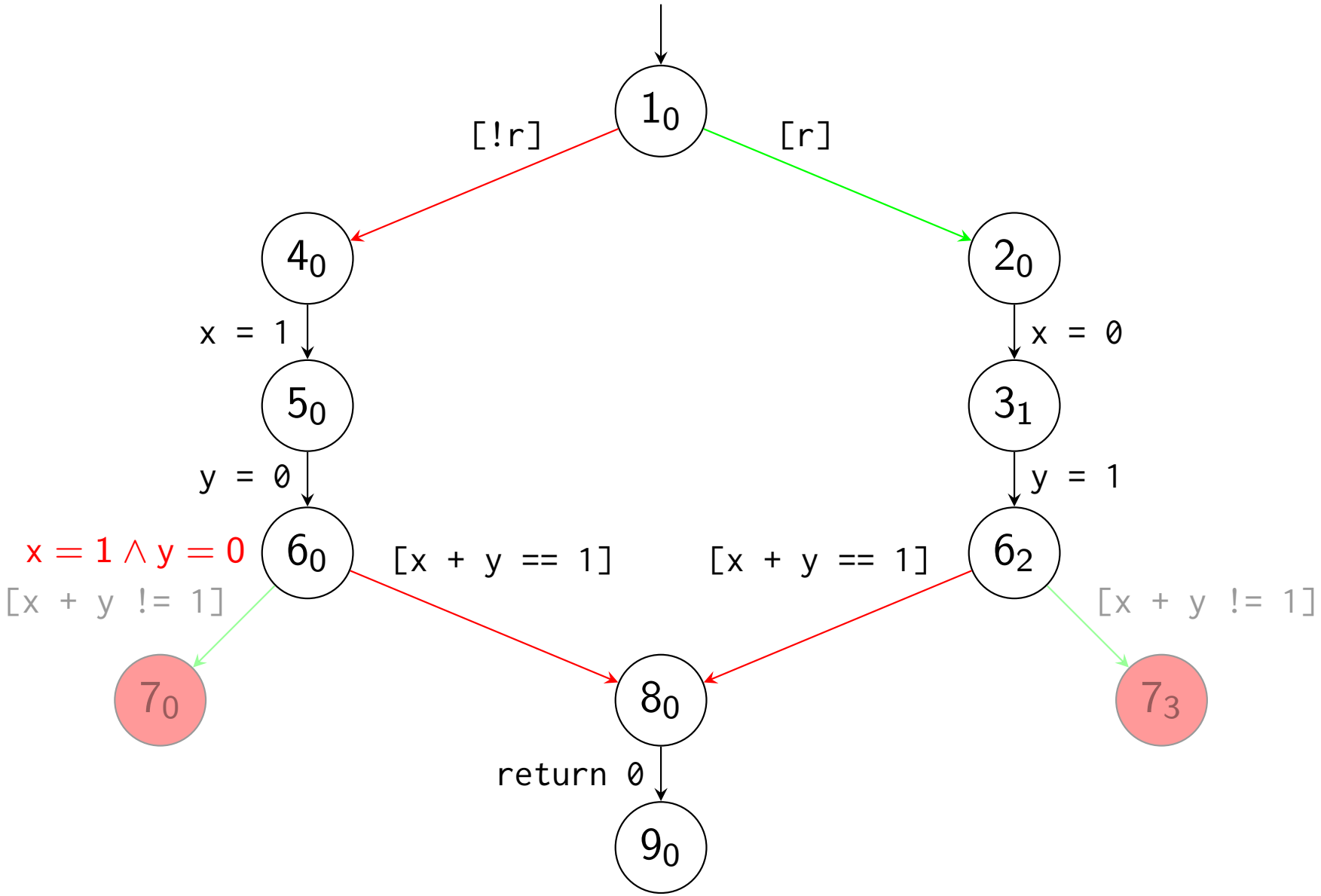
Observer automaton:



Refinement example — refined



Refinement example — refined



Conclusion

Contribution:

- Designed methods for generating interprocedural witnesses with data-flow analysis
- Adapted a model-checking method to improve precision
- Implemented both in Goblint
- Evaluated both experimentally

Conclusion

Contribution:

- Designed methods for generating interprocedural witnesses with data-flow analysis
- Adapted a model-checking method to improve precision
- Implemented both in Goblint
- Evaluated both experimentally

Conclusion

Witnesses can be generated from existing data-flow analysis

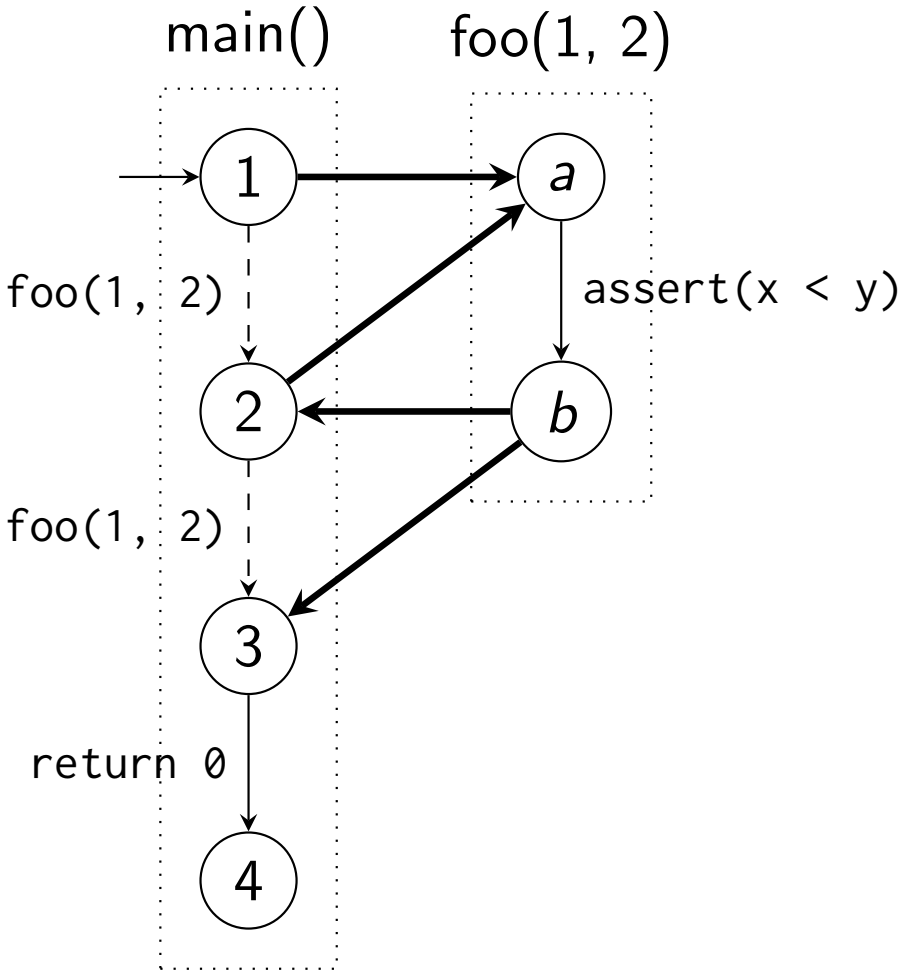
Thank you!

Interprocedural ARGs

Call-site–sensitive inlining

```
void foo(int x, int y)
{
    assert(x < y);
}

int main()
{
    foo(1, 2);
    foo(1, 2); // changed
    return 0;
}
```

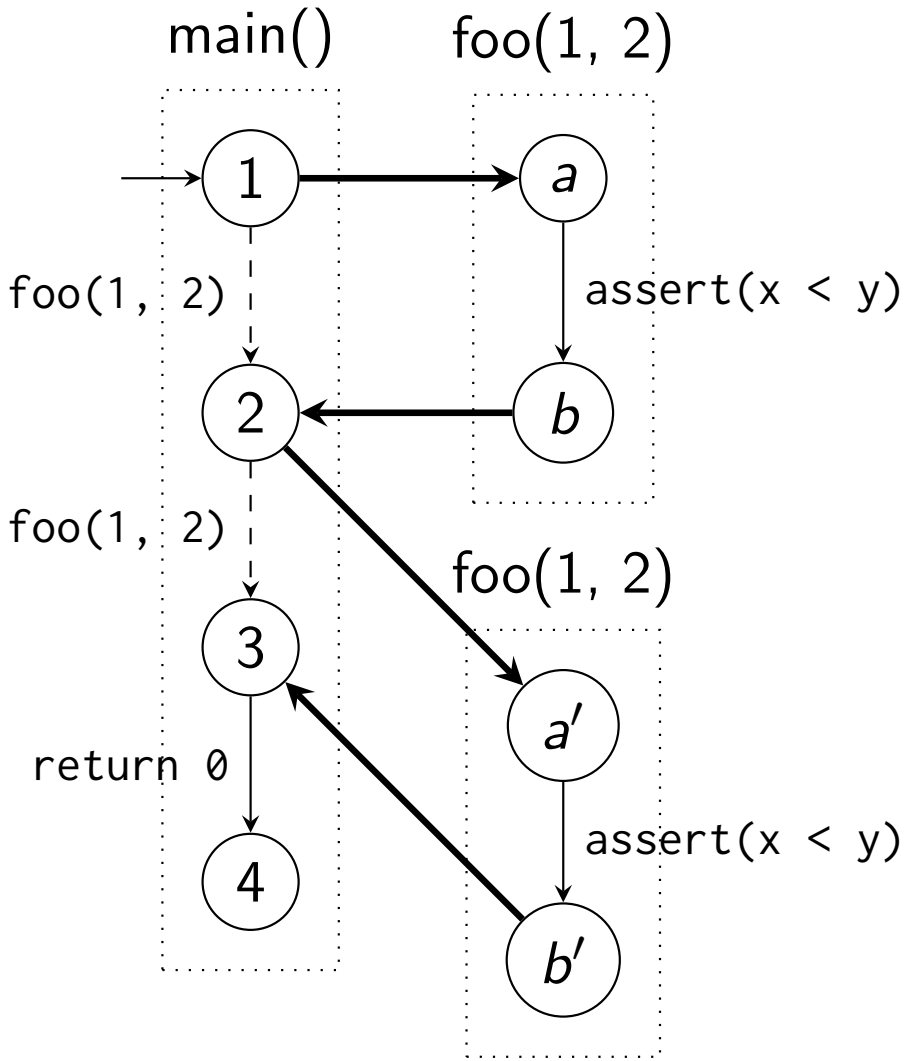


Interprocedural ARGs

Call-site–sensitive inlining

```
void foo(int x, int y)
{
    assert(x < y);
}

int main()
{
    foo(1, 2);
    foo(1, 2); // changed
    return 0;
}
```



Call-string lifting

Given ARG $A_{\text{context}} = (V, S', \delta, [\text{entry}_{\text{main}}, c_{\text{main}}])$

Define ARG $A_{\text{callsite}} = (V^*, S', \delta', [[\text{entry}_{\text{main}}, c_{\text{main}}]])$

- V^* is the set of lists with elements from V
- δ' is defined for all $[u_1, \ldots, u_n] \in V^*$ by:

$$[u_1, \ldots, u_n, u] \xrightarrow[\delta']{s} [u_1, \ldots, u_n, v] \iff u \xrightarrow[\delta]{s} v$$

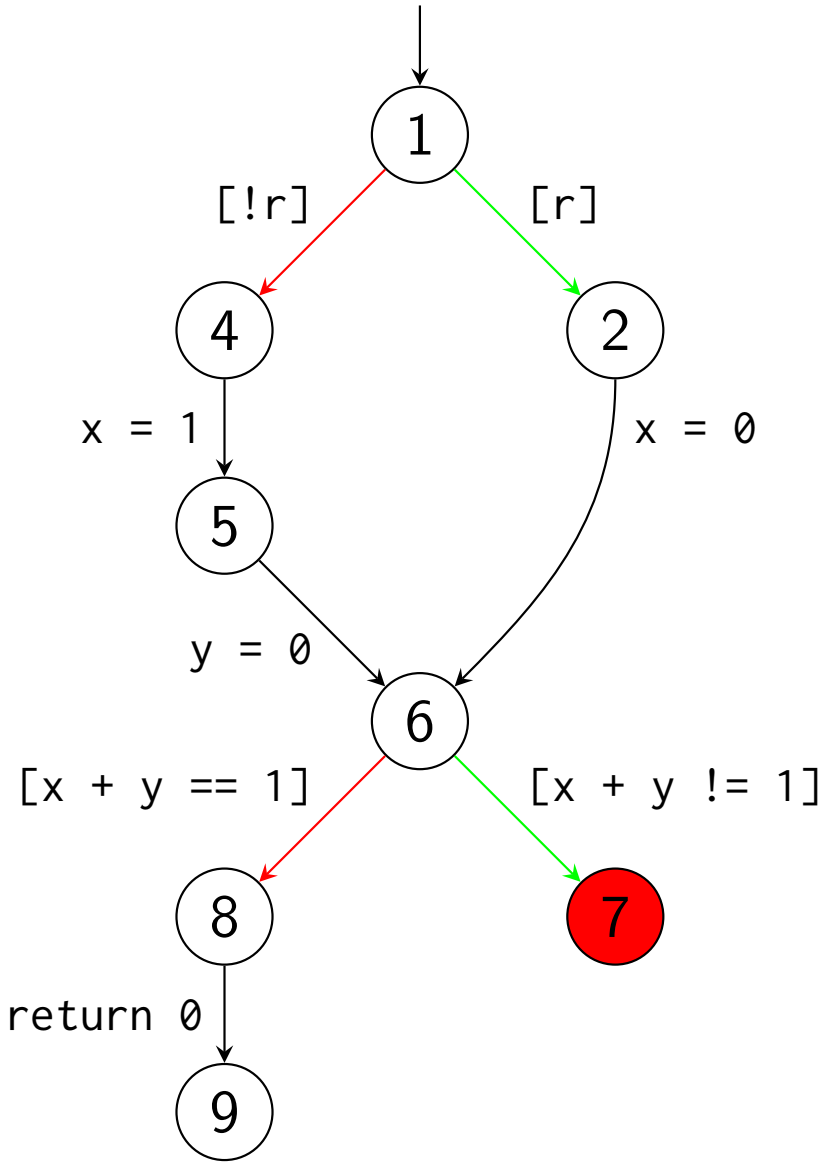
$$[u_1, \ldots, u_n, u] \xrightarrow[\delta']{\text{enterFunction}_f} [u_1, \ldots, u_n, u, v] \iff u \xrightarrow[\delta]{\text{enterFunction}_f} v$$

$$[u_1, \ldots, u_n, u] \xrightarrow[\delta']{\text{returnFromFunction}_f} [u_1, \ldots, u_{n-1}, v] \iff$$

$$\iff u \xrightarrow[\delta]{\text{returnFromFunction}_f} v \ \wedge\ \langle u_n \rangle_1 \xrightarrow[E]{f()} \langle v \rangle_1 \ \wedge\ \langle u_n \rangle_2 = \langle v \rangle_2$$

Violation example

```
int main()
{
    int r, x, y;
    if (r)
    {
        x = 0;
        // removed
    }
    else
    {
        x = 1;
        y = 0;
    }
    if (x + y != 1)
        error();
    return 0;
}
```



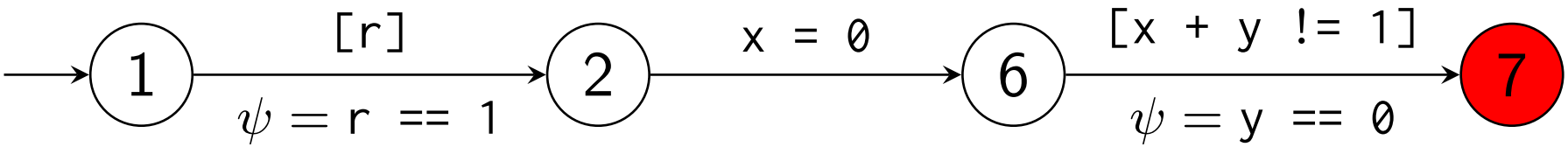
Violation example — error path

Feasible program path:

$$1 \xrightarrow{\texttt{[r]}} 2 \xrightarrow{\texttt{x = 0}} 6 \xrightarrow{\texttt{[x + y != 1]}} 7$$

Feasible model: $\texttt{r} = 1$ and $\texttt{y} = 0$.

Violation witness with edge assumptions ψ:



Feasibility analysis

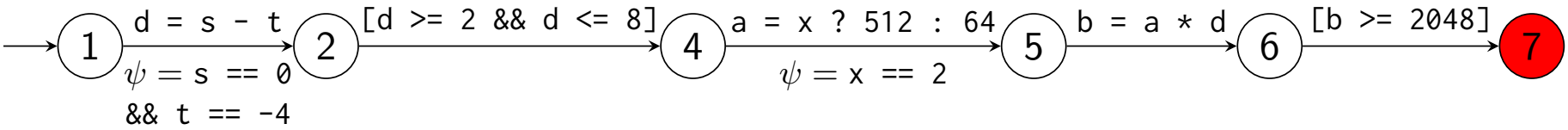
Feasible path

Feasible program subpath:

$$1 \xrightarrow{\texttt{d = s - t}} 2 \xrightarrow{\texttt{[d >= 2 \&\& d <= 8]}} 4 \xrightarrow{\texttt{a = x ? 512 : 64}} 5 \xrightarrow{\texttt{b = a * d}} 6 \xrightarrow{\texttt{[b >= 2048]}} 7.$$

Feasible model: `s == 0`, `t == -4` and `x == 2`.

Violation witness with edge assumptions ψ:



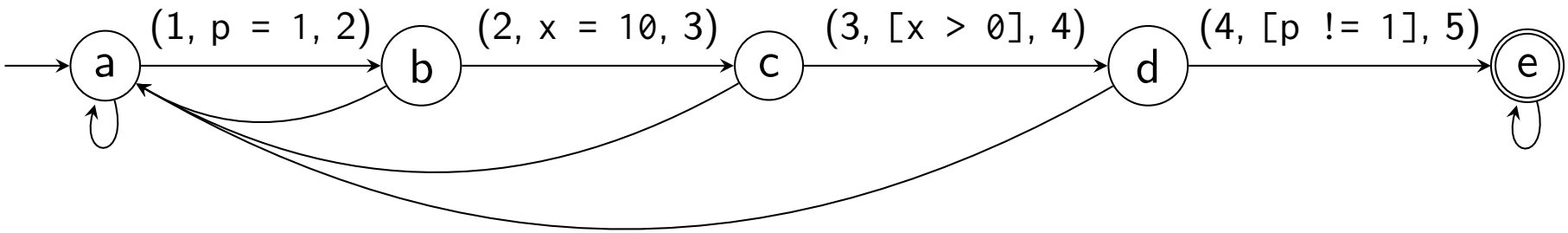
Feasibility analysis

Infeasible path

Infeasible program subpath:

$$1 \xrightarrow{\texttt{p = 1}} 2 \xrightarrow{\texttt{x = 10}} 3 \xrightarrow{\texttt{[x > 0]}} 4 \xrightarrow{\texttt{[p != 1]}} 5.$$

Non-deterministic observer automaton:



Observer analysis

Original constraint system:

$$[\mathsf{entry}_{\mathsf{main}}, c_{\mathsf{main}}] \quad \sqsupseteq \quad d_{\mathsf{main}}$$

$$\forall u \xrightarrow[E]{s} v$$

$$[v, c] \quad \sqsupseteq \quad [\![s]\!]^\sharp\,(\mathsf{get}\ \ [u, c] \quad)$$

$$\forall u \xrightarrow[E]{f()} v$$

$$[v, c] \quad \sqsupseteq \quad \mathbf{let}\,(c', d') = \mathsf{enter}^\sharp\,(\mathsf{get}\ \ [u, c] \quad)$$
$$() = \mathsf{set}\ \ [\mathsf{entry}_f, c'] \quad d'$$
$$\mathbf{in}\ \bot$$

$$\forall u \xrightarrow[E]{f()} v$$

$$[v, c] \quad \sqsupseteq \quad \mathbf{let}\,(c', d') = \mathsf{enter}^\sharp\,(\mathsf{get}\ \ [u, c] \quad)$$
$$\mathbf{in}\,\mathsf{combine}^\sharp\,(\mathsf{get}\ \ [u, c] \quad)\,(\mathsf{get}\ \ [\mathsf{return}_f, c'] \quad)$$

Observer analysis

Extended constraint system with observer automaton (Q, ω, q_0, F):

If $q_0 \notin F$

$$([\text{entry}_{\text{main}}, c_{\text{main}}], q_0) \sqsupseteq d_{\text{main}}$$

$$\forall u \xrightarrow[E]{s} v \quad \forall q \xrightarrow[\omega_F]{(u,s,v)} q'$$

$$([v, c], q') \sqsupseteq [\![s]\!]^\sharp\,(\text{get}\,([u, c], q))$$

$$\forall u \xrightarrow[E]{f()} v \quad \forall q \xrightarrow[\omega_F]{(u,\text{enterFunction}_f,\text{entry}_f)} q' \quad \forall q''' \in Q$$

$$([v, c], q''') \sqsupseteq \textbf{let}\,(c', d') = \text{enter}^\sharp\,(\text{get}\,([u, c], q))$$
$$() = \text{set}\,([\text{entry}_f, c'], q')\,d'$$
$$\textbf{in}\,\bot$$

$$\forall u \xrightarrow[E]{f()} v \quad \forall q \xrightarrow[\omega_F]{(u,\text{enterFunction}_f,\text{entry}_f)} q' \quad \forall q'' \xrightarrow[\omega_F]{(\text{return}_f,\text{returnFromFunction}_f,v)} q'''$$

$$([v, c], q''') \sqsupseteq \textbf{let}\,(c', d') = \text{enter}^\sharp\,(\text{get}\,([u, c], q))$$
$$\textbf{in}\,\text{combine}^\sharp\,(\text{get}\,([u, c], q))\,(\text{get}\,([\text{return}_f, c'], q''))$$