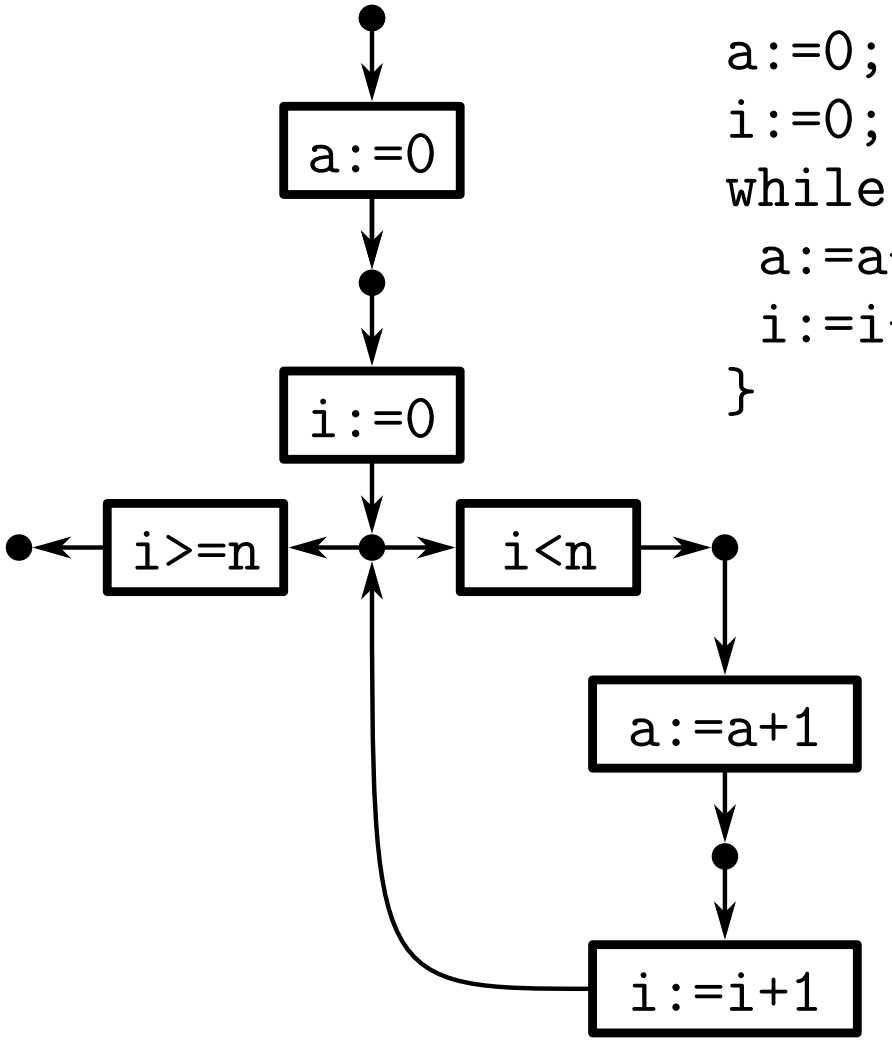
# Interprocedural Information Flow Analysis of XML Processors

Helmut Seidl and Máté Kovács

Technische Universität München, Germany

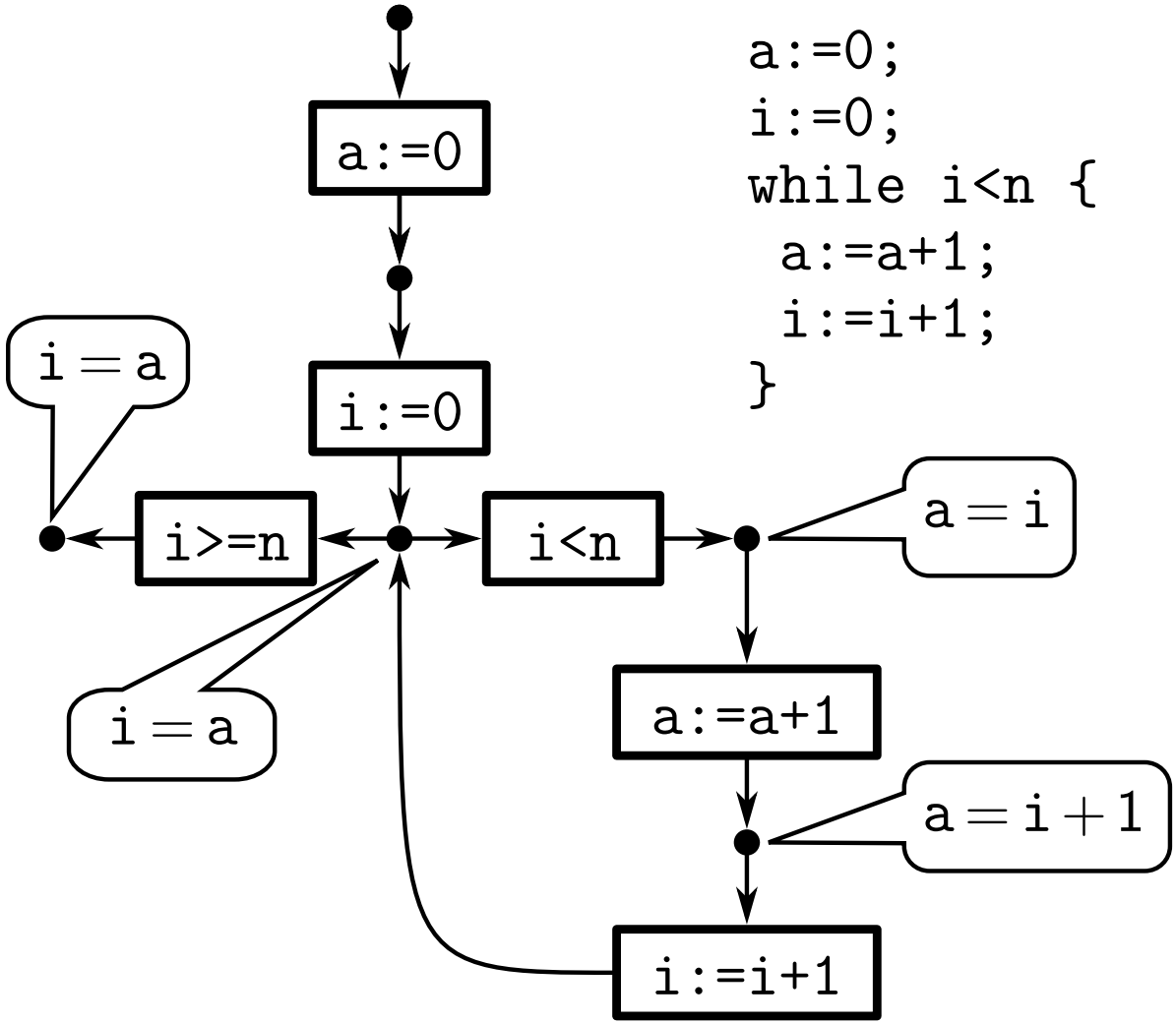
March 10, 2014

# Program Verification




```
a:=0;
i:=0;
while i<n {
 a:=a+1;
 i:=i+1;
}
```

# Program Verification



# Program Verification

Invariant $==$ Safety property

- Violation proved by finite counterexample

# Program Verification

Invariant $==$ Safety property

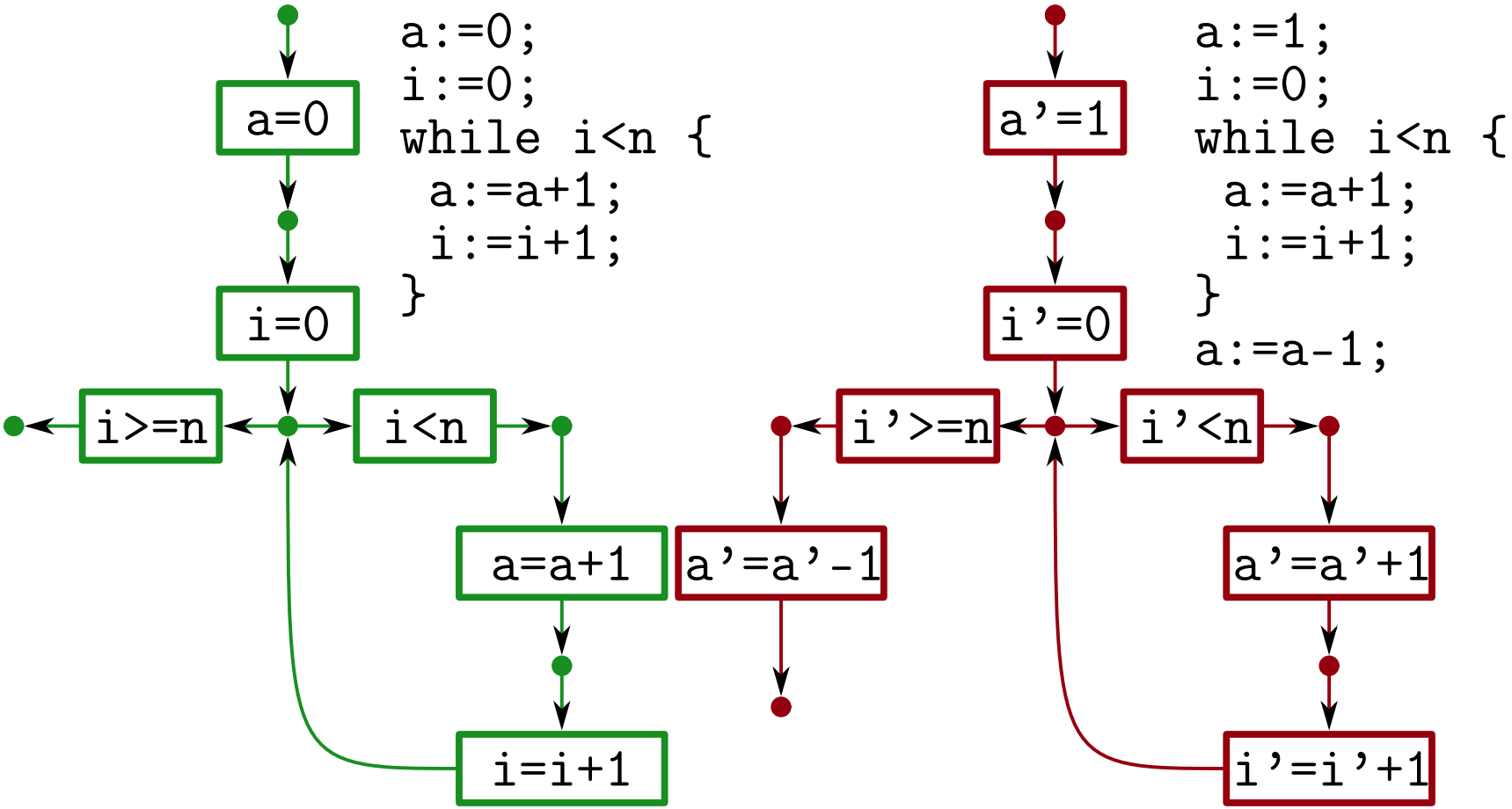
- Violation proved by finite counterexample
- In general undecidable

# Program Verification

Invariant $==$ Safety property

- Violation proved by finite counterexample
- In general undecidable
- Abstract interpretation provides computable over-approximation

# Program Equivalence



# Program Equivalence

Program equivalence $\Longleftrightarrow$ 2-Hypersafety property

// (modulo termination)

# Program Equivalence

Program equivalence $\Longleftrightarrow$ 2-Hypersafety property

$\Longleftarrow$ (modulo termination)

- Violation proved by pair of finite paths

# Program Equivalence

Program equivalence $\Longleftrightarrow$ 2-Hypersafety property

// (modulo termination)

- Violation proved by pair of finite paths
- Equally undecidable!

# Program Equivalence

Program equivalence $\Longleftrightarrow$ 2-Hypersafety property

// (modulo termination)

- Violation proved by pair of finite paths
- Equally undecidable!
- Required: computable approximation?

## Applications

- translation validation
- correctness of program transformation, e.g., for improvement, refactoring, ...

## Applications

- translation validation
- correctness of program transformation, e.g., for improvement, refactoring, ...
- information flow security

# Information Flow

Secret 

# Information Flow

Secret 

Adversary 

# Information Flow

Secret 

Adversary 

Observations 

# Information Flow

Secret 

Adversary 

Observations 

Question **Can the adversary learn about the secret?**

## Example

```
x := 0;
if (secret > 0) {x := 1;};
```

## Example

```
x := 0;
if (secret > 0) {x := 1;};
```

Observation: `x` at program exit.

## Example

```
x := 0;
if (secret > 0) {x := 1;};
```

Observation: `x` at program exit.

Formally,

$$[\![p]\!]\, s_1 \stackrel{?}{=}_{\mathtt{x}} [\![p]\!]\, s_2$$

## Example

```
x := 0;
if (secret > 0) {x := 1;};
```

Observation: `x` at program exit.

Formally,

$$[\![p]\!]\, s_1 \stackrel{?}{=}_{\mathtt{x}} [\![p]\!]\, s_2$$

Can adversary distinguish between two secret values?

# Related Work on IF

- A lattice model for secure information flow — Denning 1976
- A sound type system — Volpano et al. 1996

# Related Work on IF

- A lattice model for secure information flow — Denning 1976
- A sound type system — Volpano et al. 1996
- Jif, a type system for Java — Myers 1999
- Runtime roles and principals — Broberg, Sands 2010

# Related Work on IF

- A lattice model for secure information flow — Denning 1976
- A sound type system — Volpano et al. 1996
- Jif, a type system for Java — Myers 1999
- Runtime roles and principals — Broberg, Sands 2010
- Program dependence graphs — Hammer, Snelting 2009

# Related Work on IF

- A lattice model for secure information flow — Denning 1976
- A sound type system — Volpano et al. 1996
- Jif, a type system for Java — Myers 1999
- Runtime roles and principals — Broberg, Sands 2010
- Program dependence graphs — Hammer, Snelting 2009
- Self-composition — Barthe et al. 2004, 2011, 2013; Banerjee 2011; Kovacs et al. 2013

# Outline of the Talk

- Our approach
- Aligning of programs
- Relational abstract interpretation
- Procedures
- Structured data and tree automata

# Our Approach

## Idea

- Proceed as for program verification!

# Our Approach

## Idea

- Proceed as for program verification!
- In order to relate program semantics,

  Relate program structure

  $\Longrightarrow$ syntactical

# Our Approach

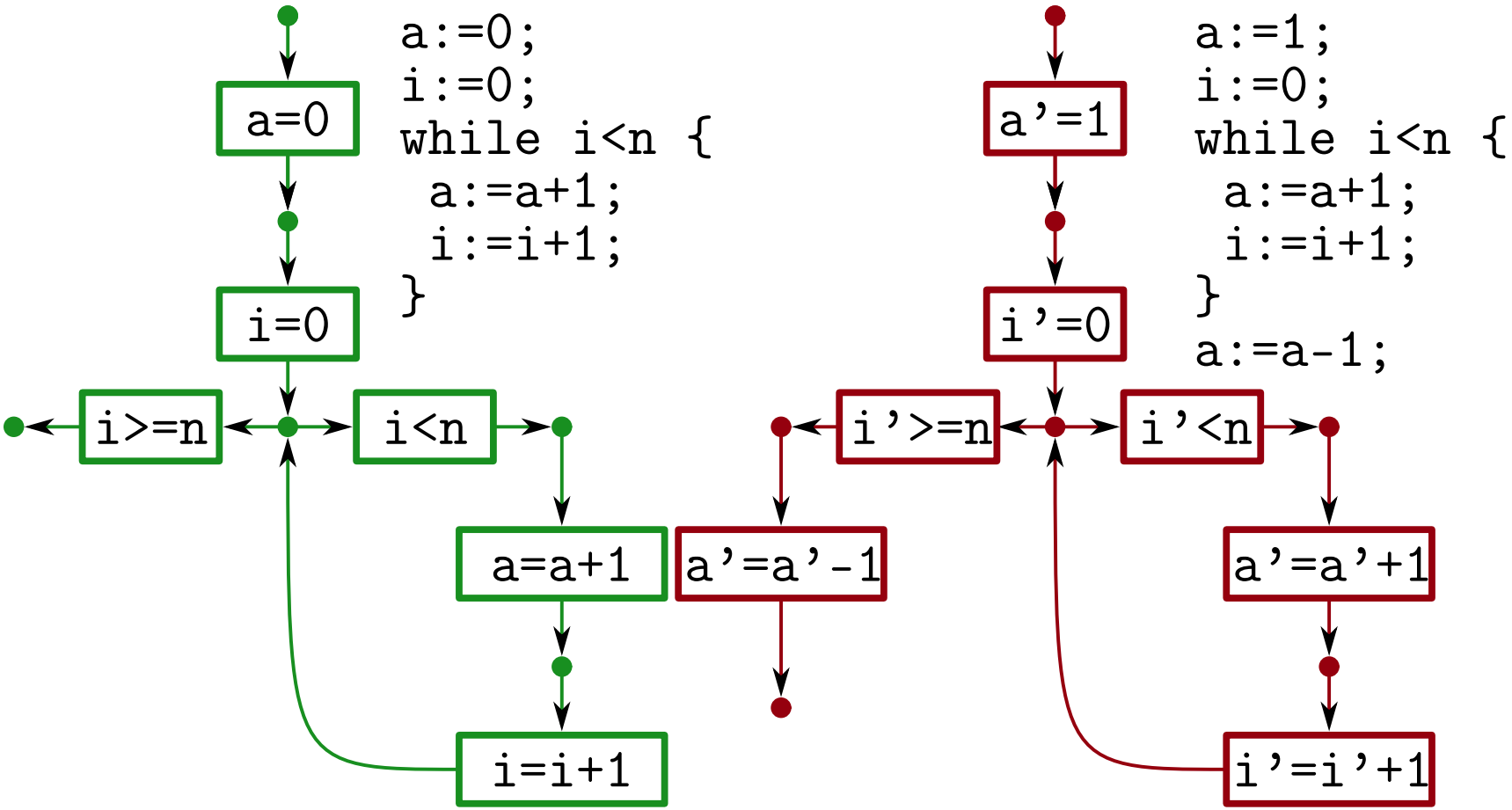
## Idea

- Proceed as for program verification!
- In order to relate program semantics,

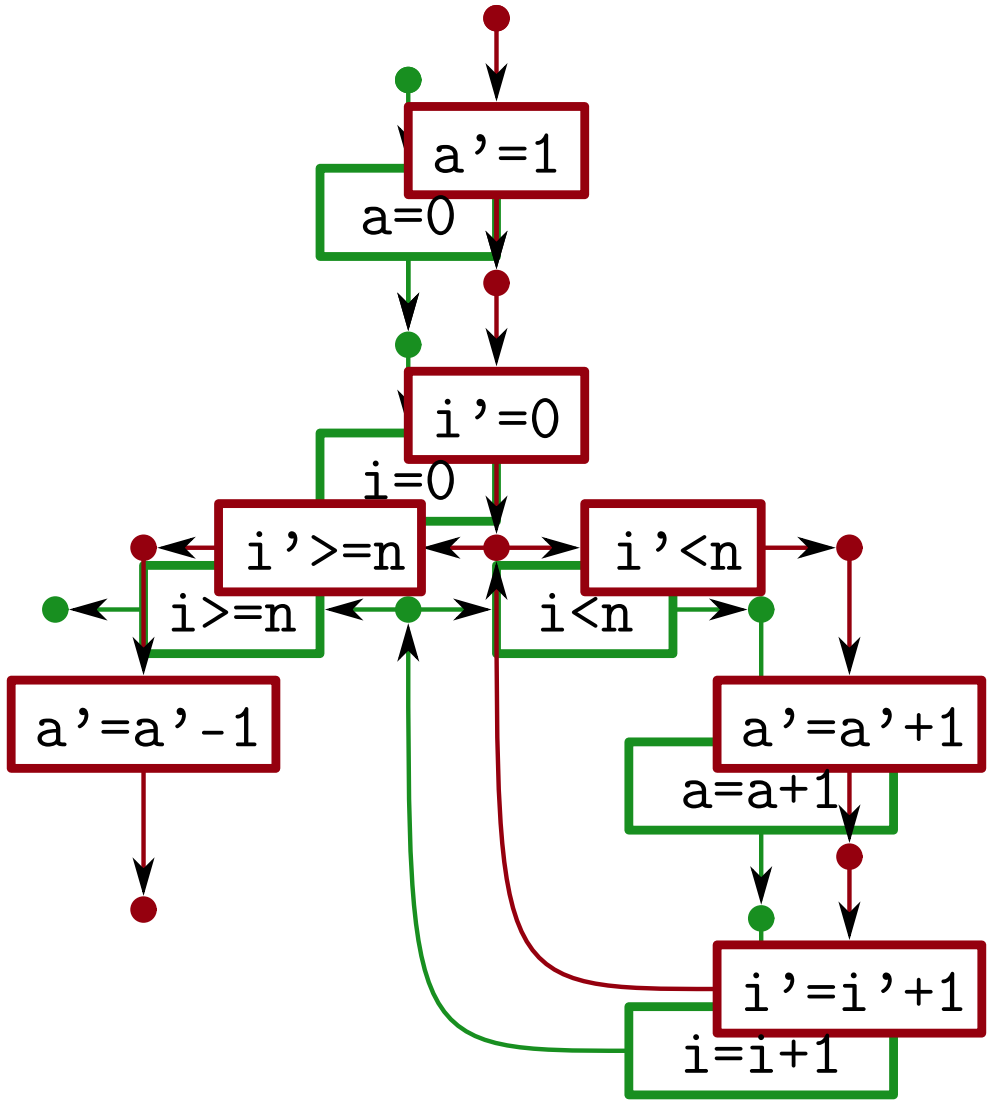
  Relate program structure
  $\Longrightarrow$ syntactical

  Relate program states at corresponding program points
  $\Longrightarrow$ semantical

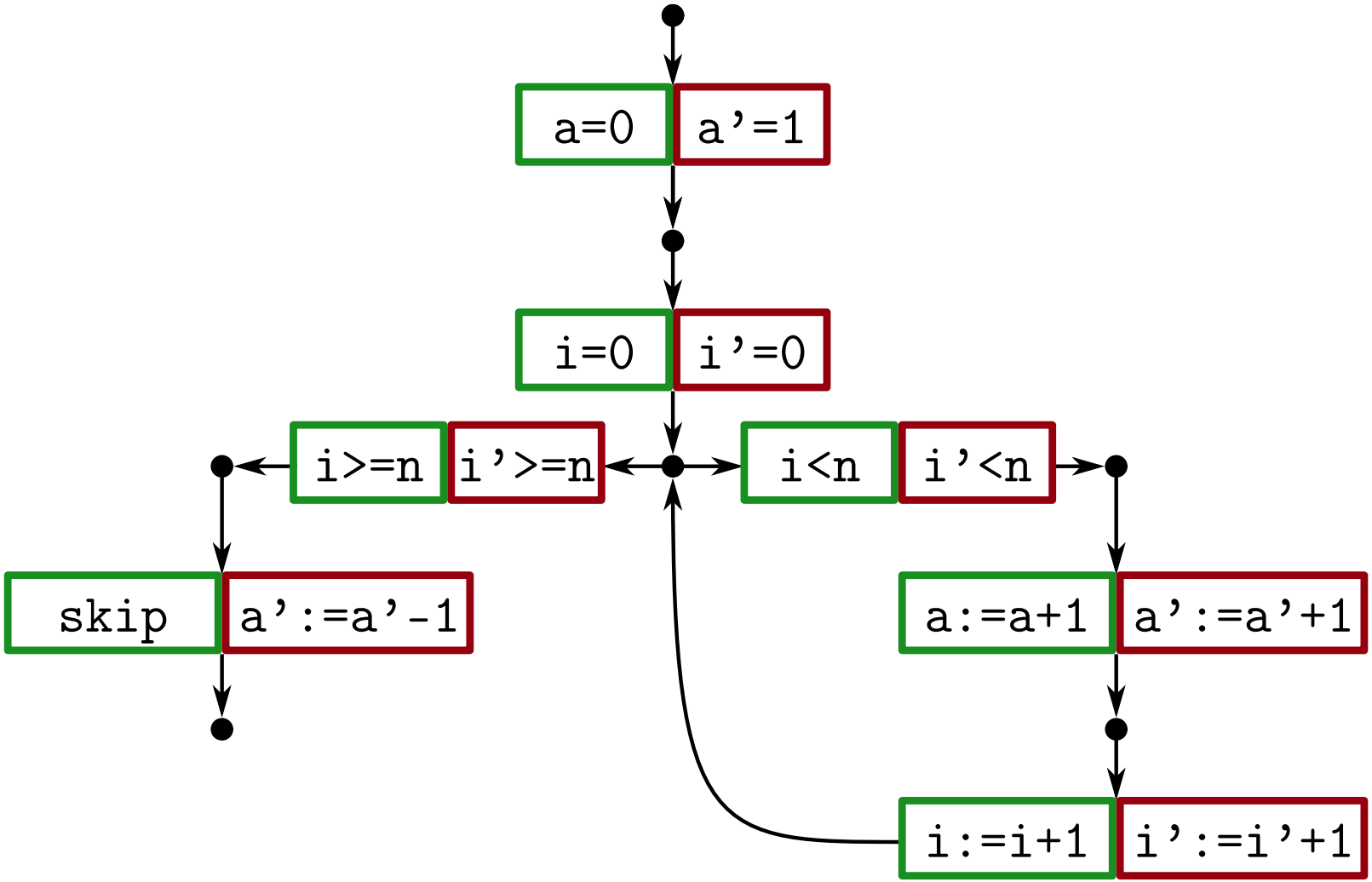
# Our Approach



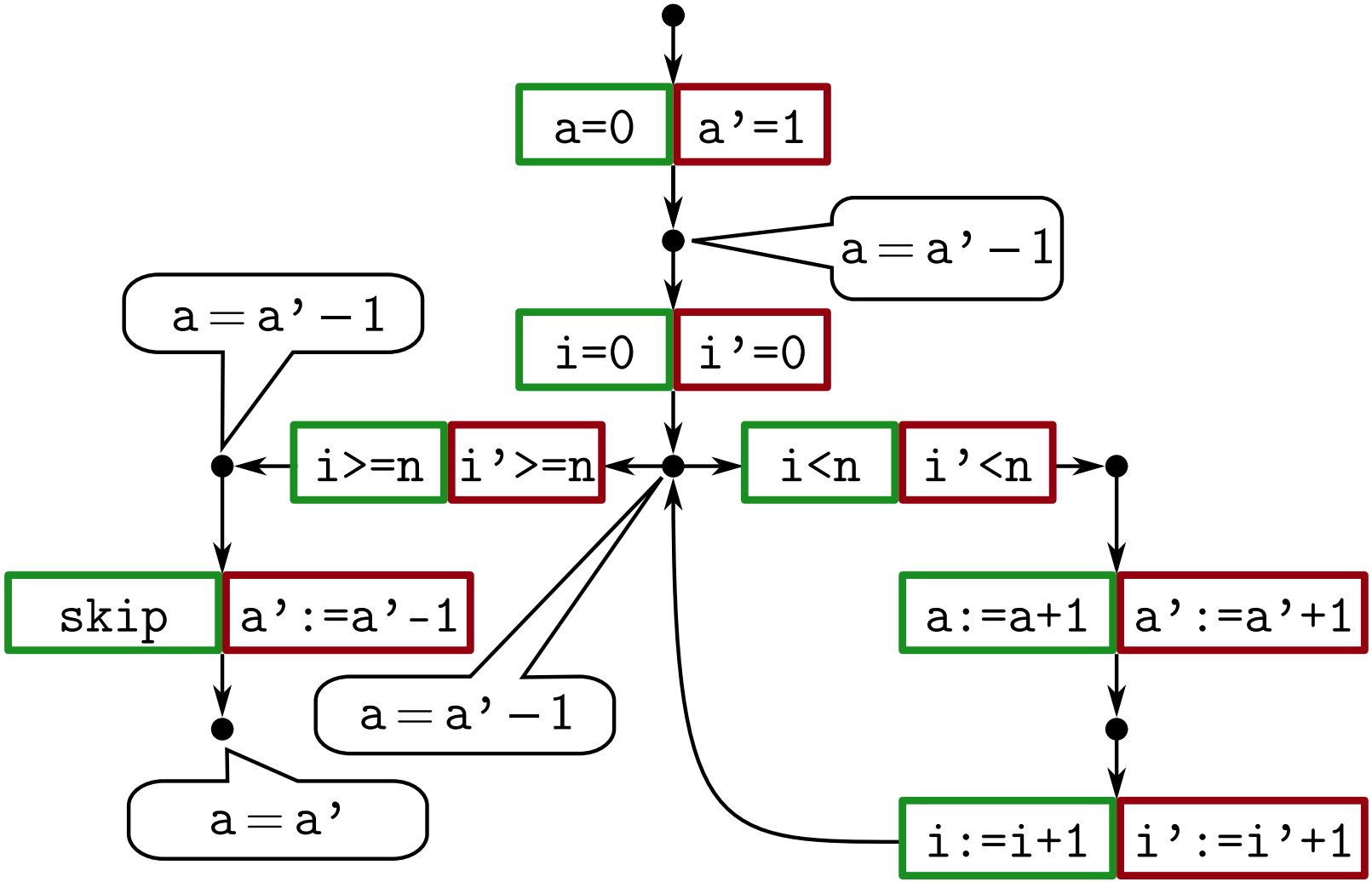
# Our Approach



# Our Approach



# Our Approach



# Relating Program Structure

Intput: programs $p_1, p_2$

Output: 2-program $[p_1, p_2]$

# Relating Program Structure

Intput: programs $p_1, p_2$

Output: 2-program $[p_1, p_2]$

## Idea

- Each pair of finite runs of $p_1, p_2$ corresponds to some finite run of $[p_1, p_2]$;

# Relating Program Structure

Intput: programs $p_1, p_2$

Output: 2-program $[p_1, p_2]$

## Idea

- Each pair of finite runs of $p_1, p_2$ corresponds to some finite run of $[p_1, p_2]$;
- The two programs are executed as synchronously as possible!

$\Longrightarrow$ Similar statements are preferably aligned ...

# Grammar of 2-Programs

$$pp \quad ::= \quad \varepsilon \quad | \quad c\,;pp$$

# Grammar of 2-Programs

$$pp ::= \varepsilon \mid c\,;pp$$

$$c ::= [x:=e, x:=e] \mid$$

# Grammar of 2-Programs

$$
\begin{array}{lcl}
pp & ::= & \varepsilon \mid c\,;pp \\
c & ::= & [x:=e, x:=e] \mid \\
 & & [\texttt{skip}, x:=e] \mid [x:=e, \texttt{skip}] \mid
\end{array}
$$

# Grammar of 2-Programs

$$
\begin{array}{lcl}
pp & ::= & \varepsilon \quad | \quad c\,;pp \\
c & ::= & [x:=e, x:=e] \quad | \\
 & & [\texttt{skip}, x:=e] \quad | \quad [x:=e, \texttt{skip}] \quad | \\
 & & \texttt{if } [b,b] \ \{p_1\} \ \texttt{else} \ \{ \\
 & & \quad p_2 \\
 & & \}
\end{array}
$$

# Grammar of 2-Programs

$$
\begin{array}{lcl}
pp & ::= & \varepsilon \quad | \quad c\,;pp \\
c & ::= & [x:=e, x:=e] \quad | \\
 & & [\texttt{skip}, x:=e] \quad | \quad [x:=e, \texttt{skip}] \quad | \\
 & & \texttt{if } [b,b] \ \{p_1\} \ \texttt{else } \{ \\
 & & \ \texttt{if } [\neg b,b] \ \{p_3\} \ \texttt{else } \{ \\
 & & \ \texttt{if } [b,\neg b]\{p_4\} \ \texttt{else } \{ \\
 & & \ p_2 \\
 & & \ \} \\
 & & \} \\
 & & \} \quad |
\end{array}
$$

# Grammar of 2-Programs (cont.)

$$
\begin{array}{rcl}
c & ::= & \texttt{if } [b,\texttt{tt}] \ \{p_1\} \ \texttt{else} \ \{p_1\} \ \mid \\
 & & \texttt{if } [\texttt{tt},b] \ \{p_2\} \ \texttt{else} \ \{p_2\} \ \mid
\end{array}
$$

# Grammar of 2-Programs (cont.)

$$
\begin{array}{rcl}
c & ::= & \texttt{if [}b\texttt{,tt] \{}p_1\texttt{\} else \{}p_1\texttt{\}} \quad | \\
& & \texttt{if [tt,}b\texttt{] \{}p_2\texttt{\} else \{}p_2\texttt{\}} \quad | \\
& & \texttt{while [}b\texttt{,}b\texttt{] \{}p_1\texttt{\};}
\end{array}
$$

# Grammar of 2-Programs (cont.)

$$
\begin{array}{rcl}
c & ::= & \texttt{if [}b\texttt{,tt] \{}p_1\texttt{\} else \{}p_1\texttt{\}} \quad | \\
& & \texttt{if [tt,}b\texttt{] \{}p_2\texttt{\} else \{}p_2\texttt{\}} \quad | \\
& & \texttt{while [}b\texttt{,}b\texttt{] \{}p_1\texttt{\};} \\
& & \texttt{while [}b\texttt{,tt] \{}p_2\texttt{\};} \\
& & \texttt{while [tt,}b\texttt{] \{}p_3\texttt{\};} \quad |
\end{array}
$$

# Grammar of 2-Programs (cont.)

$$
\begin{array}{rcl}
c & ::= & \texttt{if [}b\texttt{,tt] \{}p_1\texttt{\} else \{}p_1\texttt{\}} \;\; | \\
& & \texttt{if [tt,}b\texttt{] \{}p_2\texttt{\} else \{}p_2\texttt{\}} \;\; | \\
& & \texttt{while [}b\texttt{,}b\texttt{] \{}p_1\texttt{\};} \\
& & \texttt{while [}b\texttt{,tt] \{}p_2\texttt{\};} \\
& & \texttt{while [tt,}b\texttt{] \{}p_3\texttt{\};} \;\; | \\
& & \texttt{while [}b\texttt{,tt] \{}pp\texttt{\};} \;\; | \\
& & \texttt{while [tt,}b\texttt{] \{}pp\texttt{\};}
\end{array}
$$

# Aligning Two Programs

**Programs** $p_1 = \mathtt{c};\mathtt{d}$ $p_2 = \mathtt{e};\mathtt{f}$

# Aligning Two Programs

## Programs

$p_1 = \mathtt{c}\,;\mathtt{d}$ $\quad$ $p_2 = \mathtt{e}\,;\mathtt{f}$

## Some alignments

$$[c, \mathtt{skip}]\,;[d, \mathtt{skip}]\,;[\mathtt{skip}, e]\,;[\mathtt{skip}, f],$$
$$[c, \mathtt{skip}]\,;[d, e]\,;[\mathtt{skip}, f],$$
$$[c, e]\,;[d, f],$$
$$[\mathtt{skip}, e]\,;[c, f]\,;[d, \mathtt{skip}],$$
$$[\mathtt{skip}, e]\,;[\mathtt{skip}, f]\,;[c, \mathtt{skip}]\,;[d, \mathtt{skip}]$$

# Aligning Two Programs

**Programs** $p_1 = \mathtt{c};\mathtt{d}$ $p_2 = \mathtt{e};\mathtt{f}$

**Some alignments**

$$[c, \mathtt{skip}];[d, \mathtt{skip}];[\mathtt{skip}, e];[\mathtt{skip}, f],$$
$$[c, \mathtt{skip}];[d, e];[\mathtt{skip}, f],$$
$$[c, e];[d, f],$$
$$[\mathtt{skip}, e];[c, f];[d, \mathtt{skip}],$$
$$[\mathtt{skip}, e];[\mathtt{skip}, f];[c, \mathtt{skip}];[d, \mathtt{skip}]$$

**Idea** Minimize tree distance!

# Aligning Two Programs (cont.)

Robust Tree Edit Distance — Pawlik, Augsten 2011

$\Longrightarrow$ minimal number of insertions, deletions, renamings

# Aligning Two Programs (cont.)

Robust Tree Edit Distance — Pawlik, Augsten 2011

$\Longrightarrow$ minimal number of insertions, deletions, renamings

Aligning sequences $c_1, ..., c_k$ and $d_1, ..., d_l$

- Compute the sequence of pairs with minimal distances
- Then, compute the composition of each aligned pair $[c_i, d_j]$ ...

# Aligning Two Programs (cont.)

Robust Tree Edit Distance — Pawlik, Augsten 2011

$\Longrightarrow$ minimal number of insertions, deletions, renamings

Aligning sequences $c_1, ..., c_k$ and $d_1, ..., d_l$

- Compute the sequence of pairs with minimal distances
- Then, compute the composition of each aligned pair $[c_i, d_j]$ ...
  - If roots of ASTs are identical, proceed recursively.
  - Otherwise, align $c_i$ and $d_j$ with `skip`.

# Aligning Two Programs (cont.)

## Example

`z:=42; x:=y+1` `z:=3*z+6; x:=y+1`

are aligned to:

# Aligning Two Programs (cont.)

## Example

`z:=42; x:=y+1` `z:=3*z+6; x:=y+1`

are aligned to:

`[z:=42,skip]; [skip,z:=3*z+6];[x:=y+1,x:=y+1]`

# Aligning Two Programs (cont.)

**The program**

```
x := 0;
if (secret>0) {x := 1;};
```

**is self-aligned to**

```
[x := 0,x := 0];
if [secret>0,secret>0] [x := 1,x := 1];
```

# Aligning Two Programs (cont.)

**The program**

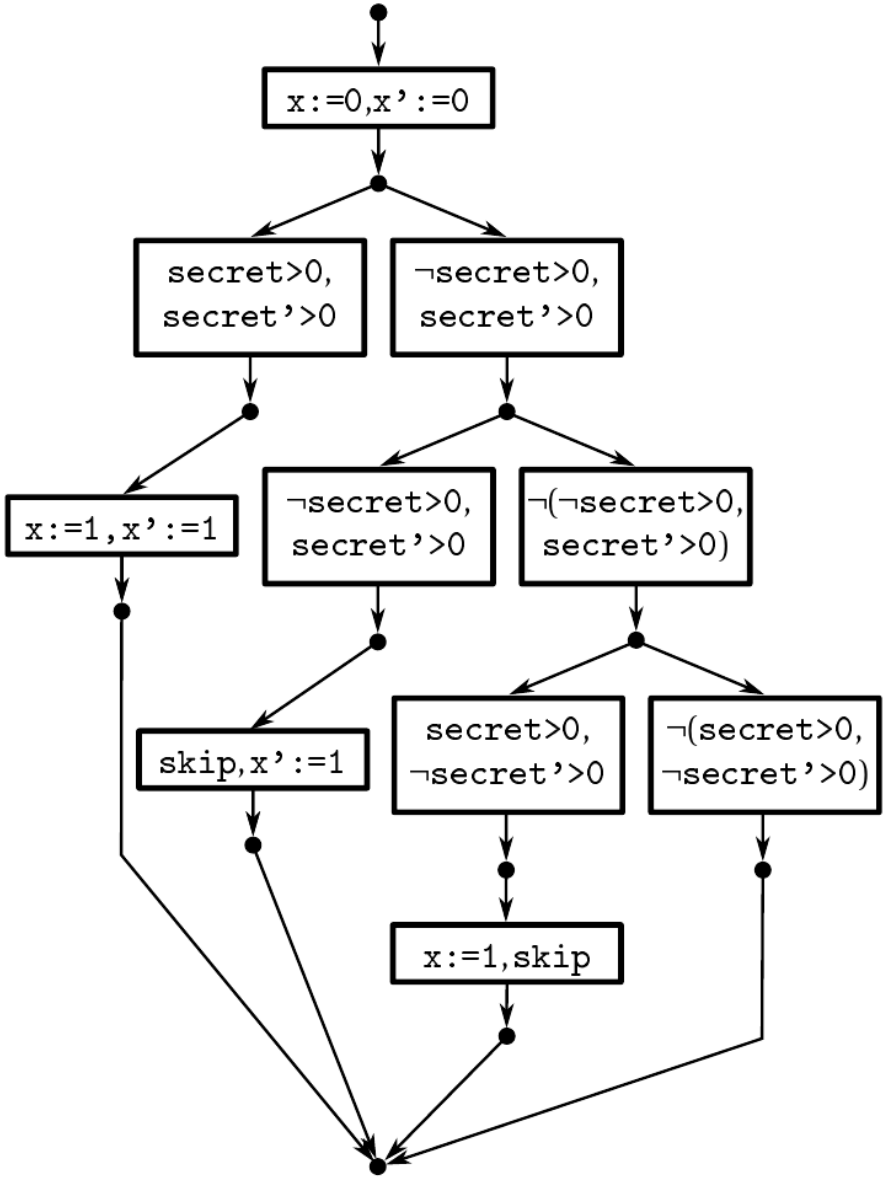
```
x := 0;
if (secret>0) {x := 1;};
```

**is self-aligned to**

```
[x := 0,x := 0];
if [secret>0,secret>0] [x := 1,x := 1];

else if [¬secret>0,secret>0]
        [skip,x := 1];
else if [secret>0,¬secret>0]
        [x := 1,skip];
```

# The CFG of the Example



# Relating Program States of 2-Programs

## Goal

Compute invariants of pairs of program states!

$\Longrightarrow$ relational abstract interpretation

# Relating Program States of 2-Programs

## Goal

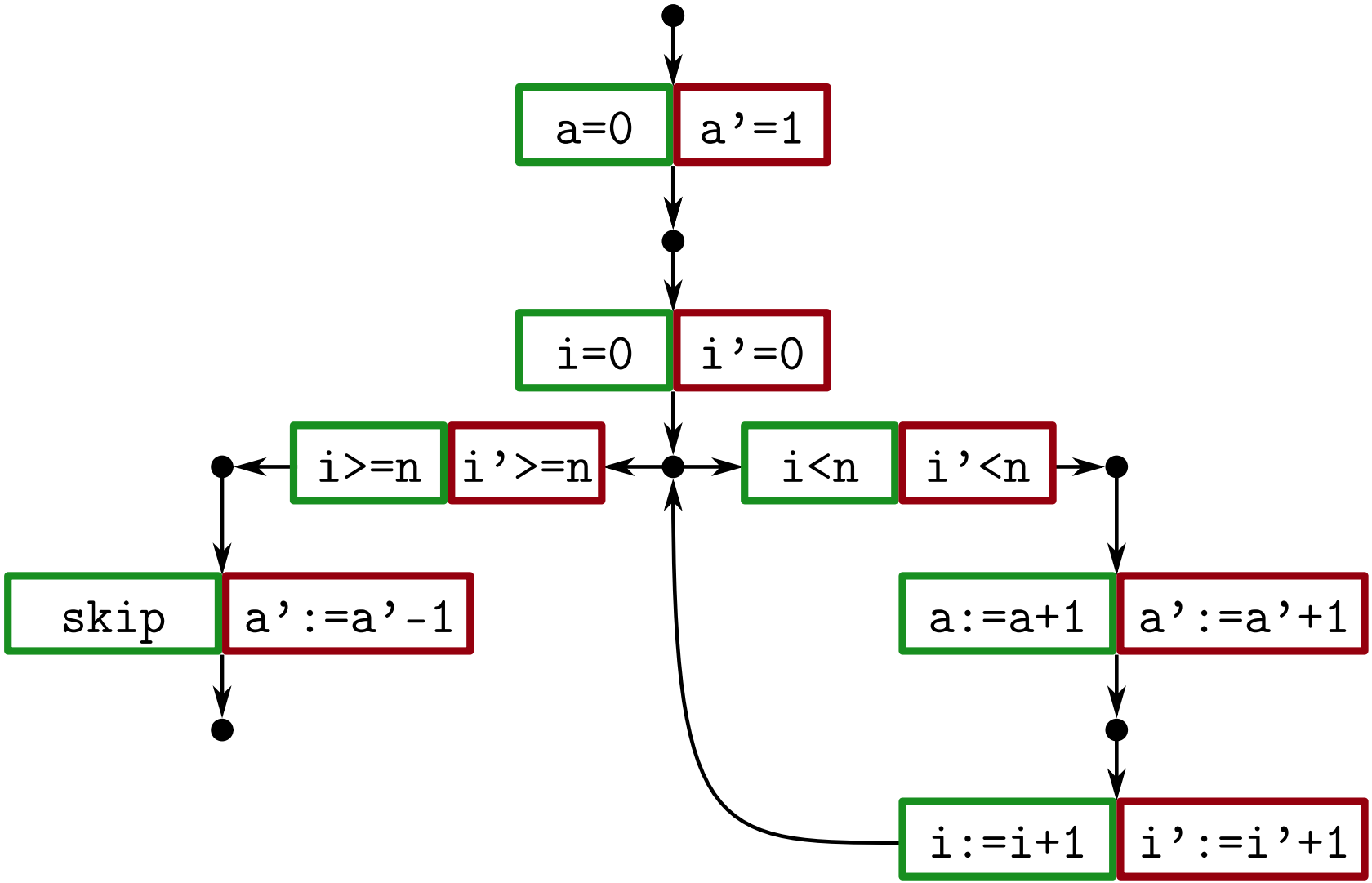
Compute invariants of pairs of program states!

$\Longrightarrow$ relational abstract interpretation

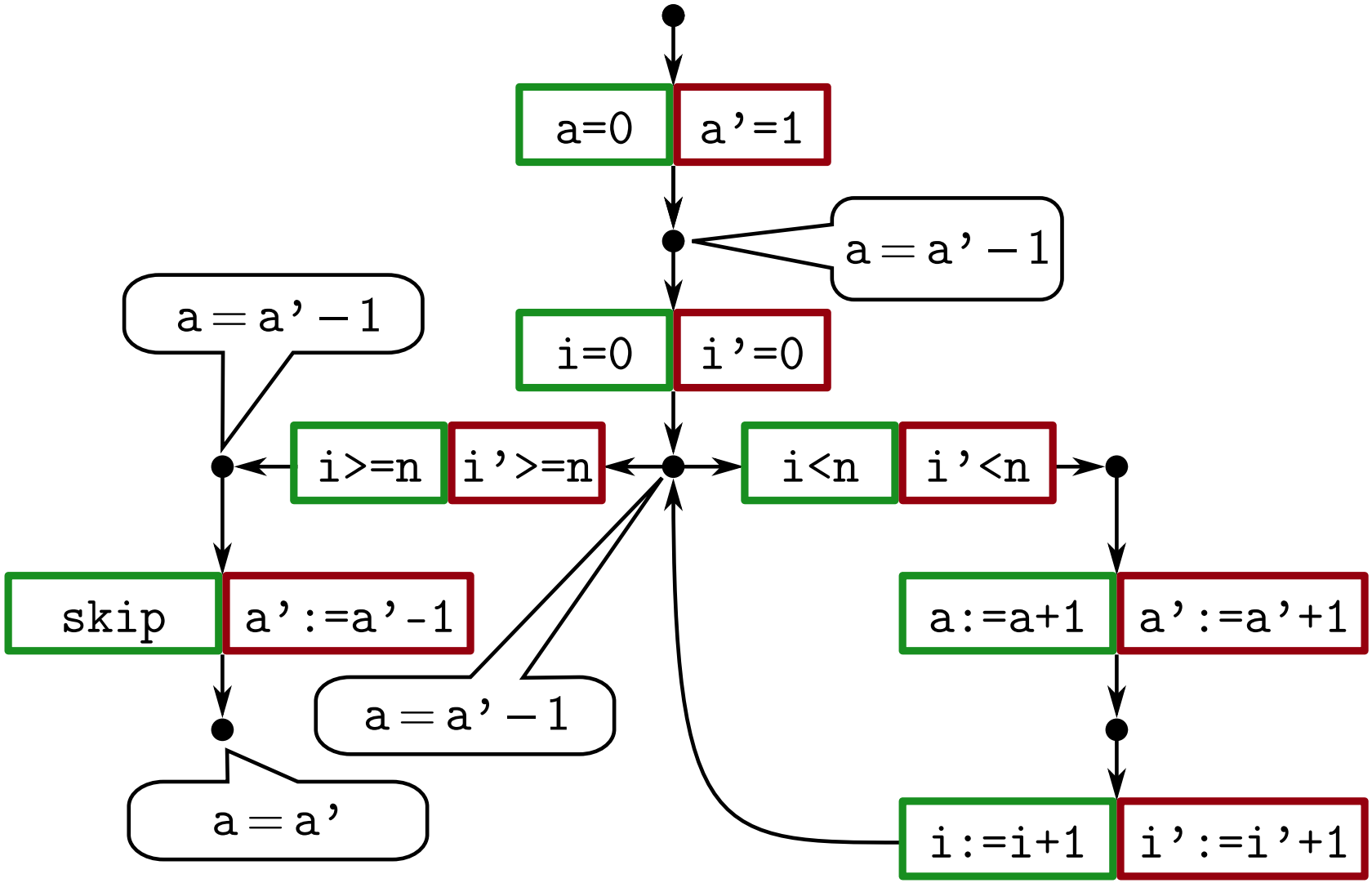
## Example

Integer programs

# Relating Program States (cont.)



# Relating Program States (cont.)



# Relating Program States (cont.)

## Invariants

$\Longrightarrow$ Affine equations $3x - 2y + 7x' = 19$.

# Relating Program States (cont.)

## Invariants

$\Longrightarrow$ Affine equations $3x - 2y + 7x' = 19$.

$\Longrightarrow$ All valid equations at a program point form a vector space.

# Relating Program States (cont.)

## Invariants

$\Longrightarrow$ Affine equations $3x - 2y + 7x' = 19$.

$\Longrightarrow$ All valid equations at a program point form a vector space.

$\Longrightarrow$ Vector spaces of equations form a complete lattice.

# Relating Program States (cont.)

## Invariants

$\Longrightarrow$ Affine equations $3x - 2y + 7x' = 19$.

$\Longrightarrow$ All valid equations at a program point form a vector space.

$\Longrightarrow$ Vector spaces of equations form a complete lattice.

$\Longrightarrow$ Occurring vector spaces cannot grow infinitely!

# Relating Program States (cont.)

## Invariants

$\Longrightarrow$ Affine equations $3x - 2y + 7x' = 19$.

$\Longrightarrow$ All valid equations at a program point form a vector space.

$\Longrightarrow$ Vector spaces of equations form a complete lattice.

$\Longrightarrow$ Occurring vector spaces cannot grow infinitely!

$\Longrightarrow$ Can be computed by fixpoint iteration!!

Karr 1976

Müller-Olm, S. 2004

# Interprocedural 2-Hypersafety Properties

```
f() { b1 }
g() { b2 }
if (secret = 1) f(); else g();
```

# Interprocedural 2-Hypersafety Properties

```
f() { b1 }
g() { b2 }
if (secret = 1) f(); else g();
```

## Idea

- Align procedure calls with procedure calls only!

# Interprocedural 2-Hypersafety Properties

```
f() { b1 }
g() { b2 }
if (secret = 1) f(); else g();
```

## Idea

- Align procedure calls with procedure calls only!
- The resulting 2-program has procedures

  $$[\mathtt{f},\mathtt{g}] \quad [\mathtt{f},\mathtt{skip}] \quad [\mathtt{skip},\mathtt{g}]$$

  where the body of $[\mathtt{f},\mathtt{g}]$ is the alignment of `b1`, `b2`.

# Interprocedural 2-Hypersafety Properties

```
f() { b1 }
g() { b2 }
if (secret = 1) f(); else g();
```

## Idea

- Align procedure calls with procedure calls only!
- The resulting 2-program has procedures

  $$[\mathtt{f},\mathtt{g}] \quad [\mathtt{f},\mathtt{skip}] \quad [\mathtt{skip},\mathtt{g}]$$

  where the body of $[\mathtt{f},\mathtt{g}]$ is the alignment of `b1`, `b2`.
- Then apply standard interprocedural abstract interpretation!

# Interprocedural 2-Hypersafety Properties

```
f() { b1 }
g() { b2 }
if (secret = 1) f(); else g();
```

## Idea

- Align procedure calls with procedure calls only!
- The resulting 2-program has procedures

  $$[\mathtt{f},\mathtt{g}] \quad [\mathtt{f},\mathtt{skip}] \quad [\mathtt{skip},\mathtt{g}]$$

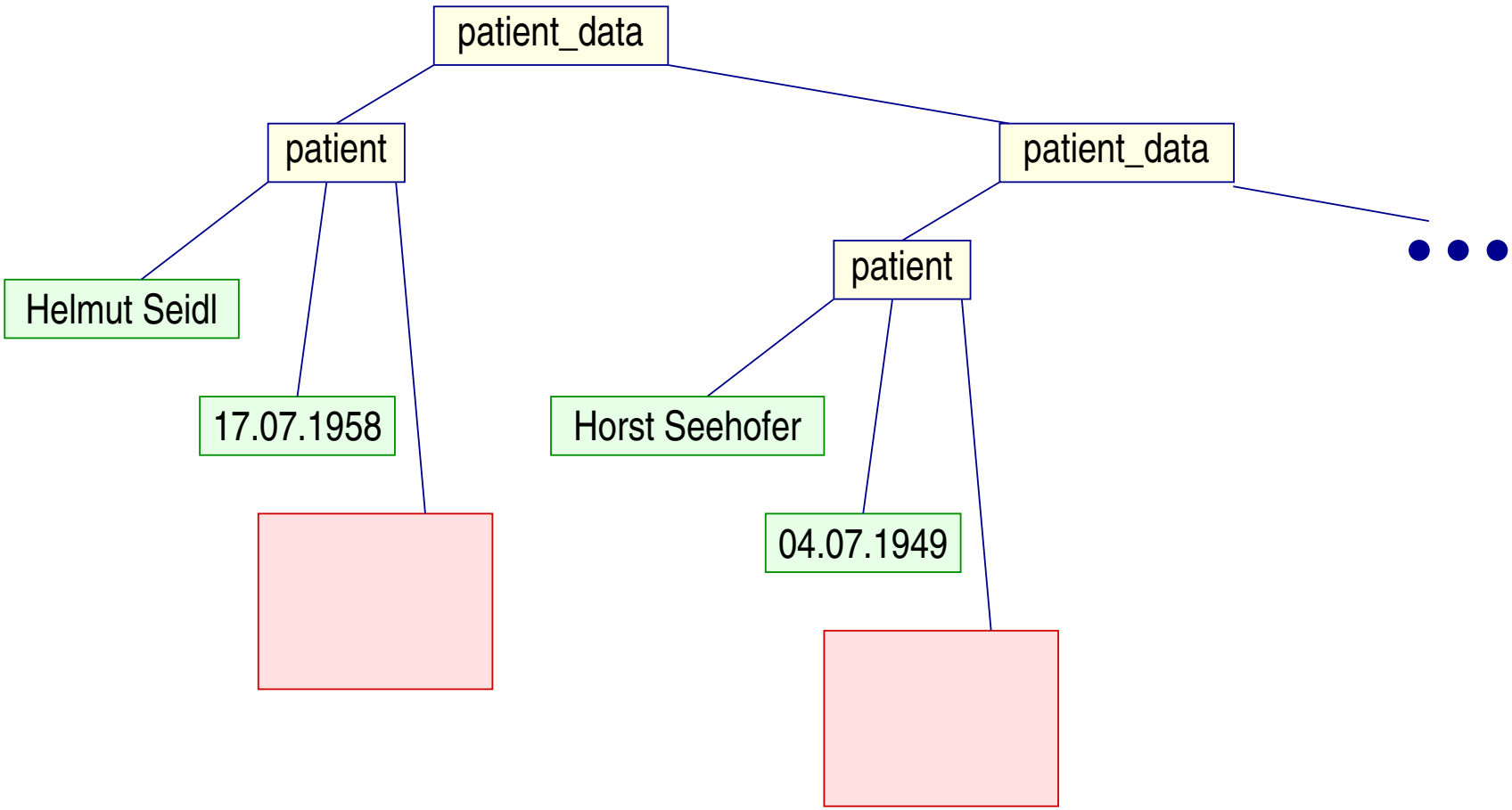
  where the body of $[\mathtt{f},\mathtt{g}]$ is the alignment of `b1`, `b2`.
- Then apply standard interprocedural abstract interpretation!
  E.g., with affine equations Müller-Olm, S. 2003, 2005

**Example** Tree-manipulating Programs

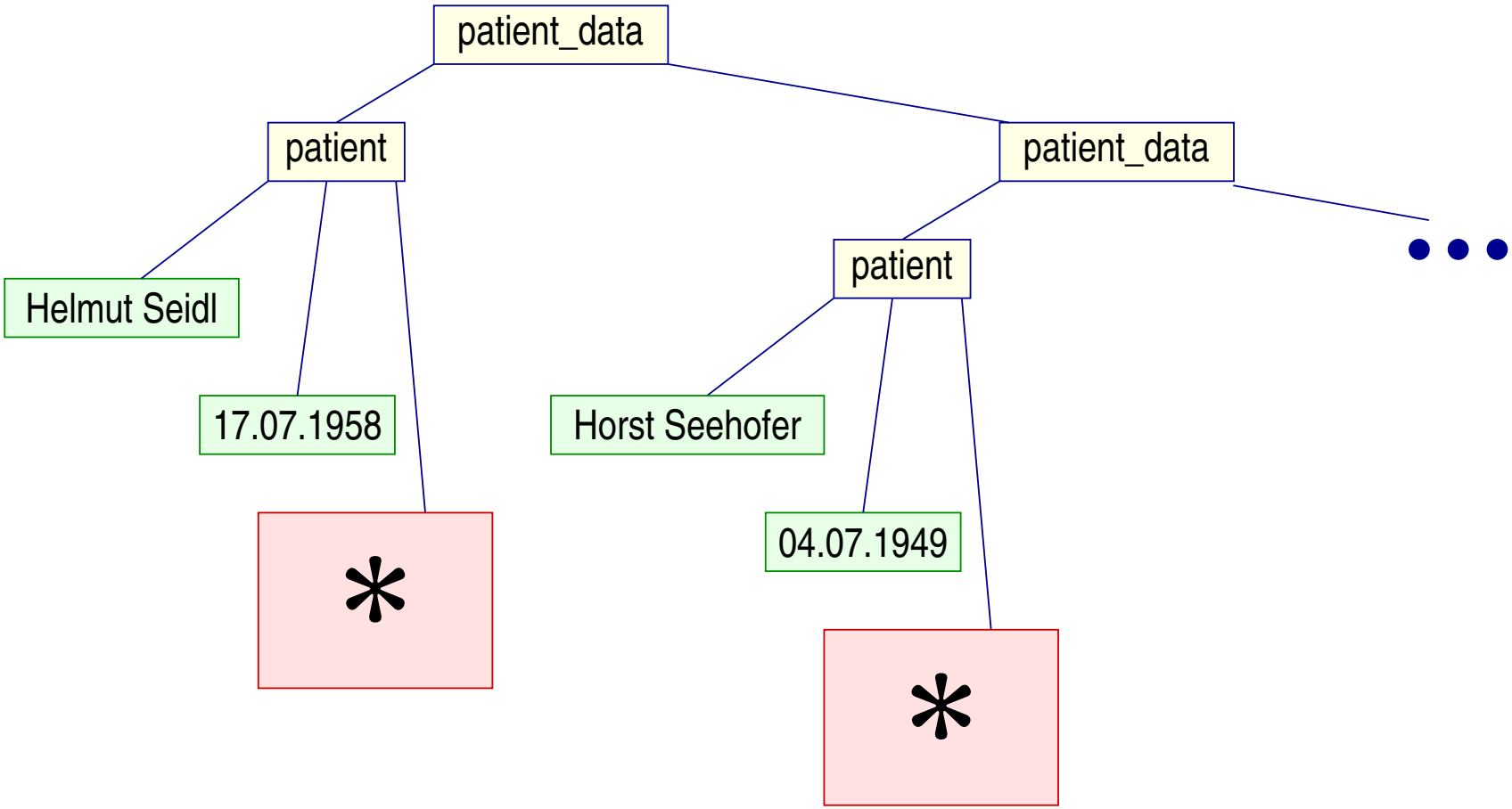
// Xml processors
// workflow languages

```
while (patient_data != []) {
  patient = patient_data.1;
  if (patient.1 = 'Helmut Seidl') {
    result = patient.2;
    patient_data = [];
  } else
    patient_data = patient_data.2;
}
```

# Tree Data Structure



# Relational Invariant



# Relational Invariant

- Leaf $*$ represents data depending on secrets.
- Tree describes data which are independent of secrets

$\Longrightarrow$ public view

# Relational Invariant

- Leaf $*$ represents data depending on secrets.
- Tree describes data which are independent of secrets

$\Longrightarrow$ public view

## Goal

Compute for every program point of a 2-program all public views of variables!

# Analysis of Tree-Manipulating Programs

Unknowns $\langle u, x \rangle$

// $u$ 2-program point

// $x$ program variable

# Analysis of Tree-Manipulating Programs

Unknowns $\langle u, x \rangle$

// $u$ 2-program point

// $x$ program variable

Constraints Horn clauses ...

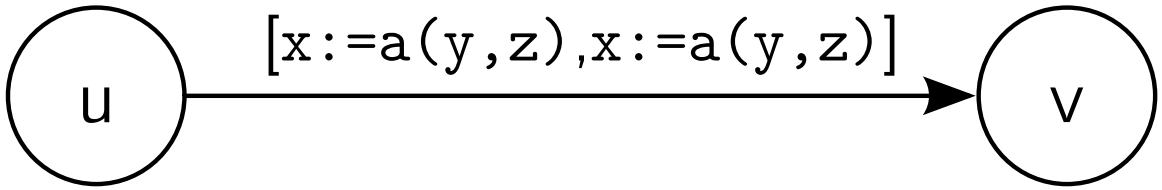
# Analysis of Tree-Manipulating Programs

Unknowns $\langle u, x \rangle$

// $u$ 2-program point

// $x$ program variable

Constraints Horn clauses ...




$$\langle v, \mathtt{x} \rangle(\mathtt{a}(H, T)) \Longleftarrow \langle u, \mathtt{y} \rangle(H), \langle u, \mathtt{z} \rangle(T)$$

## Observation

- Sets of Horn clauses have a unique least model.
- In general, the least model cannot be computed.

## Observation

- Sets of Horn clauses have a unique least model.
- In general, the least model cannot be computed.

- The Horn clauses required by the analysis, though, fall into the special class $\mathscr{H}_1$ ...

## Observation

- Sets of Horn clauses have a unique least model.
- In general, the least model cannot be computed.

- The Horn clauses required by the analysis, though, fall into the special class $\mathscr{H}_1$ ...

$\Longrightarrow$ the least model is regular

$\Longrightarrow$ and can be effectively computed!

Weidenbach 1999
Nielson, Nielson, S. 2002

## Format

$$\begin{aligned} p(a(X_1,\ldots,X_k)) &\Longleftarrow \text{any} \\ p(X) &\Longleftarrow \text{any} \end{aligned}$$

# The Class $\mathscr{H}_1$

## Format

$$\begin{aligned} p(a(X_1,\ldots,X_k)) &\Longleftarrow \text{any} \\ p(X) &\Longleftarrow \text{any} \end{aligned}$$

## Method

- Replace complicated clauses by implied simpler clauses!

# The Class $\mathscr{H}_1$

## Format

$$\begin{aligned} p(a(X_1,\ldots,X_k)) &\Longleftarrow \text{any} \\ p(X) &\Longleftarrow \text{any} \end{aligned}$$

## Method

- Replace complicated clauses by implied simpler clauses!
- Simplified Horn clauses:

$$\begin{aligned} p(a(X_1,\ldots,X_k)) &\Longleftarrow q_1(X_{j_1}),\ldots,q_m(X_{j_m}) \\ p(X) &\Longleftarrow q_1(X),\ldots,q_r(X) \end{aligned}$$

# The Class $\mathscr{H}_1$

## Format

$$\begin{aligned} p(a(X_1,\ldots,X_k)) &\Longleftarrow \text{any} \\ p(X) &\Longleftarrow \text{any} \end{aligned}$$

## Method

- Replace complicated clauses by implied simpler clauses!
- Simplified Horn clauses:

$$\begin{aligned} p(a(X_1,\ldots,X_k)) &\Longleftarrow q_1(X_{j_1}),\ldots,q_m(X_{j_m}) \\ p(X) &\Longleftarrow q_1(X),\ldots,q_r(X) \end{aligned}$$

$$\Longrightarrow \quad \text{(alternating) tree automata}$$

# Summary

- Information flow security can be formulated as a 2-hypersafety property.

# Summary

- Information flow security can be formulated as a 2-hypersafety property.
- 2-hypersafety properties can be analyzed by relational abstract interpretation of 2-programs.
- This approach even scales to programs with procedures.

# Summary

- Information flow security can be formulated as a 2-hypersafety property.
- 2-hypersafety properties can be analyzed by relational abstract interpretation of 2-programs.
- This approach even scales to programs with procedures.
- Tree language technology helps
  - to find good alignments
  - to analyze tree-manipulating programs