Software Model Checking with Ultimate

Jochen Hoenicke

Albert-Ludwigs-Universität
Freiburg, Germany

July 3rd & 5th, 2019

Brief History of Model Checking

- 1980: Emerson & Clarke: Explicit state model-checking.

Brief History of Model Checking

- 1980: Emerson & Clarke: Explicit state model-checking.
- 1992: Burch, Clarke, McMillan, Dill & Hwang. Symbolic model checking. 10^{20} states and beyond.

Brief History of Model Checking

- 1980: Emerson & Clarke: Explicit state model-checking.
- 1992: Burch, Clarke, McMillan, Dill & Hwang. Symbolic model checking. 10^{20} states and beyond.
- 2001: Ball, Majumdar, Millstein & Rajamani: Predicate Abstraction

Brief History of Model Checking

- 1980: Emerson & Clarke: Explicit state model-checking.
- 1992: Burch, Clarke, McMillan, Dill & Hwang. Symbolic model checking. 10^{20} states and beyond.
- 2001: Ball, Majumdar, Millstein & Rajamani: Predicate Abstraction
- 2009: Heizmann, H. & Podelski: Trace Abstraction.

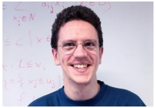
Ultimate

Ultimate – a software model-checker

`https://ultimate.informatik.uni-freiburg.de/`

- Push button verification
- Input: C program
- Output: Correct (plus invariants) or Incorrect (plus counter-example)

The Ultimate Team


Jürgen Christ


Daniel Dietsch


Marius Greitschus


Mathias Heizmann


Jochen Hoenicke


Alexander Nutz


Martin Schäf


Tanja Schindler

. . . and 36 more contributers.

You can contribute too:

`https://github.com/ultimate-pa/ultimate`

Ultimate is a Framework

- Ultimate Automizer: Trace Abstraction for Safety
- Ultimate Büchi Automizer: Trace Abstraction for Termination
- Ultimate Kojak: Predicate Abstraction
- Ultimate Taipan: Abstract Interpretation + Trace Abstraction

Ultimate is a Framework

- Ultimate Automizer: Trace Abstraction for Safety
- Ultimate Büchi Automizer: Trace Abstraction for Termination
- Ultimate Kojak: Predicate Abstraction
- Ultimate Taipan: Abstract Interpretation + Trace Abstraction
- Ultimate LTL Automizer: Checking LTL properties
- Ultimate Petri Automizer: Concurrent Programs

Demo

Example Program

```
int f(int i) {
  int j = 0;
  if (i > 0) {
    int *p = malloc(sizeof(int));    // allocate pointer
    *p = 0;
    while (i > 0) {
      *p += i;                       // use pointer
      if (i == 1) {                  // if in the last iteration:
        j = *p;
        free(p);                     //   free the pointer
      }
      i--;                           // decrement i
    }
  }
  return j;
}
```

Example Program

```
int f(int i) {
  int j = 0;
  if (i > 0) {
    int *p = malloc(sizeof(int));    // allocate pointer
    *p = 0;
    while (i > 0) {
      *p += i;                       // use pointer
      if (i == 1) {                  // if in the last iteration:
        j = *p;
        free(p);                     //   free the pointer
      }
      i--;                           // decrement i
    }
  }
  return j;
}
```

- Is every allocated memory eventually freed?
- Is every pointer dereference valid?

Adding Automatic Annotations

```
int f(int i) {
  int j = 0;
  if (i > 0) {
    int *p = malloc(sizeof(int));    // allocate pointer

    *p = 0;
    while (i > 0) {
      *p += i;                       // use pointer

      if (i == 1) {                  // if in the last iteration:
        j = *p;
        free(p);                     //   free the pointer

      }
      i--;                           // decrement i
    }
  }

  return j;
}
```

Adding Automatic Annotations

```
int f(int i) {
  int j = 0;
  if (i > 0) {
    int *p = malloc(sizeof(int));   // allocate pointer
    palloc = 1;
    *p = 0;
    while (i > 0) {
      *p += i;                      // use pointer
      assert(palloc == 1);
      if (i == 1) {                 // if in the last iteration:
        j = *p;
        free(p);                    //   free the pointer
        palloc = 0;
      }
      i--;                          // decrement i
    }
  }
  assert(palloc == 0);
  return j;
}
```

Adding Automatic Annotations

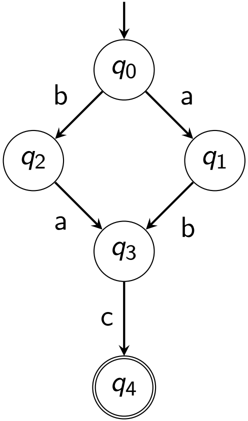
```
int f(int i) {
  ...
  if (i > 0) {
    ...                                    // allocate pointer
    palloc = 1;
    ...
    while (i > 0) {
      ...                                  // use pointer
      assert(palloc == 1);
      if (i == 1) {                        // if in the last iteration:
        ...
        ...                                //   free the pointer
        palloc = 0;
      }
      i--;                                 // decrement i
    }
  }
  assert(palloc == 0);
}
```

Trace Abstraction

Finite Automaton and Program Automaton

A finite automaton $\mathcal{A} = (\Sigma, Q, \rightarrow, q_0, F)$ consists of

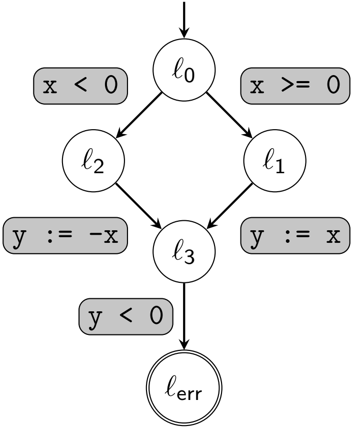
- Σ: a finite alphabet
- Q: a finite set of locations
- $\rightarrow \subseteq Q \times \Sigma \times Q$: a transition relation
- $q_0 \in Q$: the initial location
- $F \subseteq Q$: the accepting locations



Finite Automaton and Program Automaton

A finite automaton $\mathcal{A} = (\Sigma, Q, \rightarrow, q_0, F)$ consists of

- Σ: a finite alphabet
- Q: a finite set of locations
- $\rightarrow \subseteq Q \times \Sigma \times Q$: a transition relation
- $q_0 \in Q$: the initial location
- $F \subseteq Q$: the accepting locations



A program automaton is a finite automaton:

- Σ is the set of statements occurring in the program
- Q are the program locations
- $\rightarrow$ defines the control flow graph
- $\ell_0 \in Q$: the initial location
- F is the set containing the error location ℓ_{err}

A word over Σ is called a trace.
The language of the program automaton is the set of error traces.

Statements

- Σ is the set of statements occuring in the program.

Only two kinds of statements:

- `x := expr` assigns the value of `expr` to variable `x`
- `cond` checks if the condition `cond` is true, blocks otherwise.

Statements

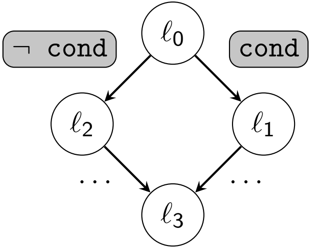
- Σ is the set of statements occuring in the program.

Only two kinds of statements:

- `x := expr` assigns the value of `expr` to variable `x`
- `cond` checks if the condition `cond` is true, blocks otherwise.

Translation of if statement

```
ℓ0:  if (cond)
ℓ1:     ...
     else
ℓ2:     ...
ℓ3:
```



Statements

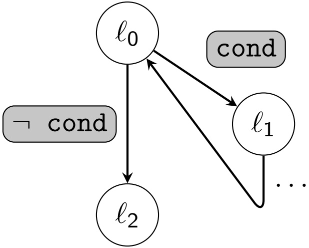
- Σ is the set of statements occuring in the program.

Only two kinds of statements:

- `x := expr` assigns the value of `expr` to variable `x`
- `cond` checks if the condition `cond` is true, blocks otherwise.

Translation of while statement

```
ℓ0:  while (cond)
ℓ1:     ...
ℓ2:
```



Statements

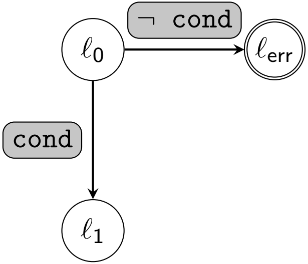
- Σ is the set of statements occuring in the program.

Only two kinds of statements:

- `x := expr` assigns the value of `expr` to variable `x`
- `cond` checks if the condition `cond` is true, blocks otherwise.

Translation of assert statement

ℓ_0: `assert(cond)`
ℓ_1:



Statements

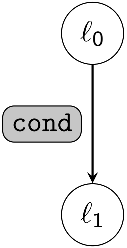
- Σ is the set of statements occuring in the program.

Only two kinds of statements:

- `x := expr` assigns the value of `expr` to variable `x`
- `cond` checks if the condition `cond` is true, blocks otherwise.

Translation of assume statement

ℓ_0: `assume(cond)`
ℓ_1:

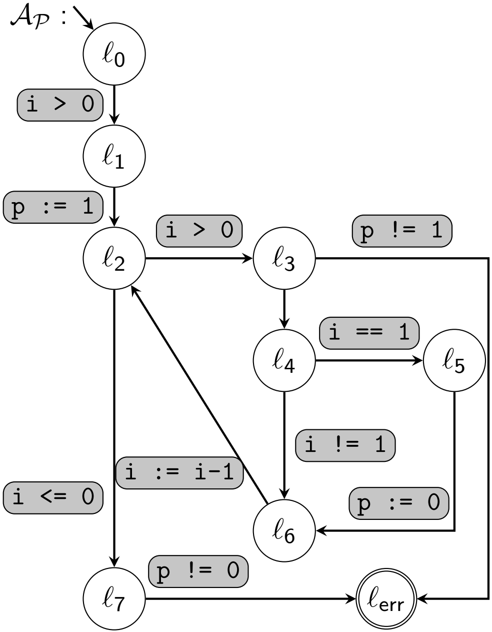


Example – Program and Program Automaton

```
$\ell_0$:  assume i > 0
$\ell_1$:  p := 1
$\ell_2$:  while (i > 0) {
$\ell_3$:    assert p == 1
$\ell_4$:    if (i == 1)
$\ell_5$:      p := 0
$\ell_6$:    i := i - 1
           }
$\ell_7$:  assert p == 0
```

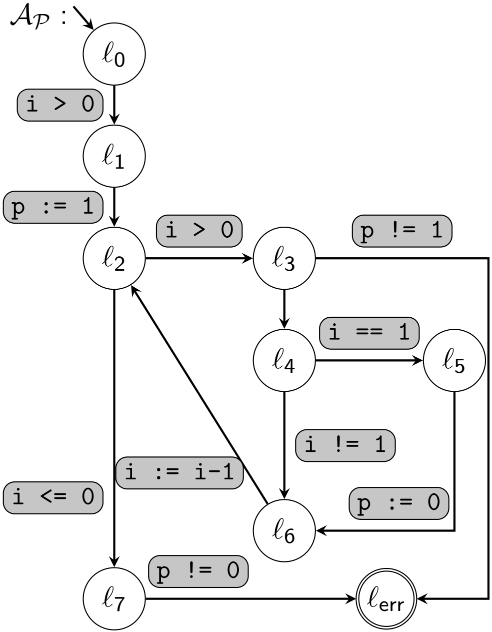
Example – Program and Program Automaton

```
ℓ0:  assume i > 0
ℓ1:  p := 1
ℓ2:  while (i > 0) {
ℓ3:    assert p == 1
ℓ4:    if (i == 1)
ℓ5:      p := 0
ℓ6:    i := i - 1
     }
ℓ7:  assert p == 0
```



Example – Program and Program Automaton

$\Sigma = \{$ `i > 0`, `i <= 0`, `i := i-1`,
`i == 1`, `i != 1`, `p := 1`,
`p != 0`, `p != 1`, `p := 0` $\}$



Example – Program and Program Automaton

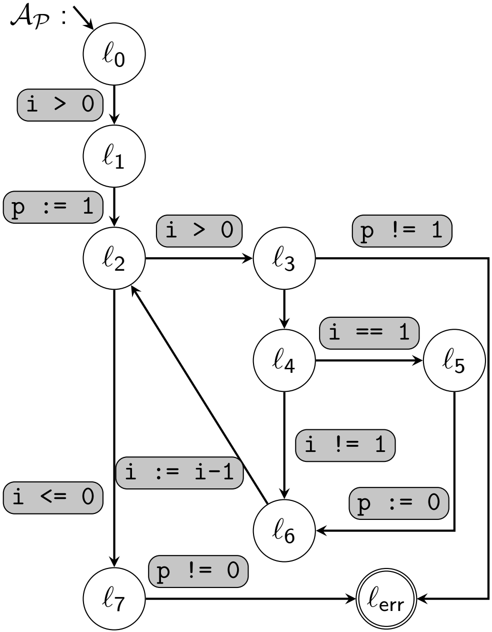
$$\Sigma = \{ \texttt{i > 0}, \texttt{i <= 0}, \texttt{i := i-1}, \texttt{i == 1}, \texttt{i != 1}, \texttt{p := 1}, \texttt{p != 0}, \texttt{p != 1}, \texttt{p := 0} \}$$

Trace

Word over the alphabet of statements.

Example:

$\pi = $ `i == 1` `i := i-1` `i == 1`



Example – Program and Program Automaton

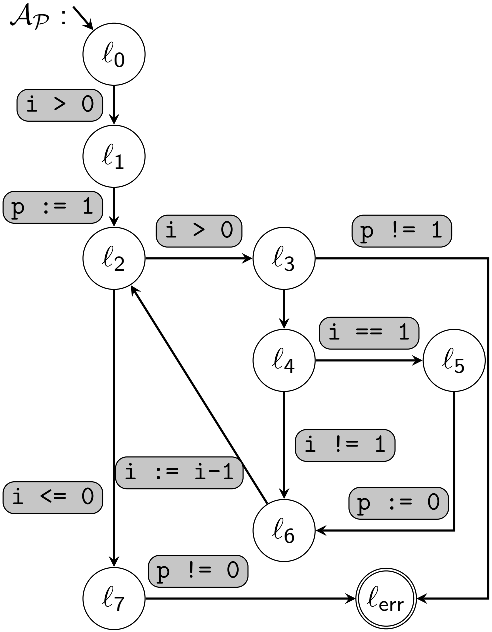
$$\Sigma = \{\boxed{\texttt{i > 0}}, \boxed{\texttt{i <= 0}}, \boxed{\texttt{i := i-1}}, \boxed{\texttt{i == 1}}, \boxed{\texttt{i != 1}}, \boxed{\texttt{p := 1}}, \boxed{\texttt{p != 0}}, \boxed{\texttt{p != 1}}, \boxed{\texttt{p := 0}}\}$$

Error Trace

Word accepted by the program automaton.

Example:

$$\pi = \boxed{\texttt{i > 0}}\boxed{\texttt{p := 1}}\boxed{\texttt{i > 0}}\boxed{\texttt{p != 1}}$$



Example – Program and Program Automaton

$\Sigma = \{$ `i > 0`, `i <= 0`, `i := i-1`,
`i == 1`, `i != 1`, `p := 1`,
`p != 0`, `p != 1`, `p := 0` $\}$

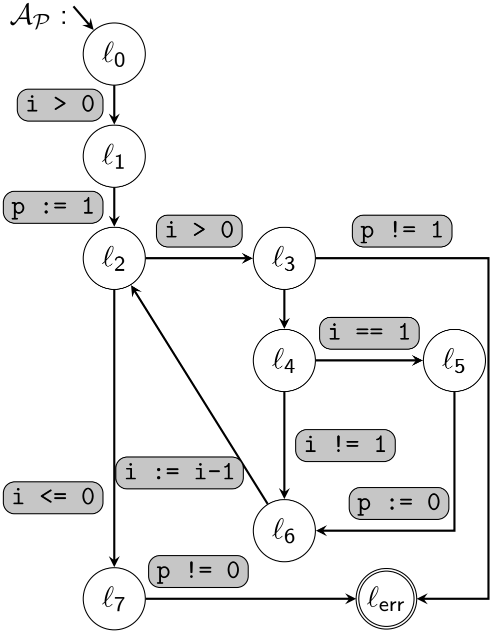
Error Trace

Word accepted by the program automaton.

Example:

$\pi =$ `i > 0` `p := 1` `i > 0` `p != 1`

Does π refute correctness of $\mathcal{P}$?



Valuations

A valuation $\nu : Var \to Value$ maps variables to some value domain.

$$\nu_0 = \begin{Bmatrix} i \mapsto 1 \\ p \mapsto 0 \end{Bmatrix}$$

Valuation are extended to expressions in a natural way.

$$\nu_0(i - 1) = \nu_0(i) - 1 = 0$$

The update of a valuation $\nu[x := c]$ is a copy of valuation ν that maps x to c.

$$\nu_0[i := 0] = \begin{Bmatrix} i \mapsto 0 \\ p \mapsto 0 \end{Bmatrix}$$

Semantics of Statements

The meaning of the statements is given by a transition system.

- Valuations are the states of the transition system.
- Transitions are labelled with statements.

Semantics of Statements

The meaning of the statements is given by a transition system.

- Valuations are the states of the transition system.
- Transitions are labelled with statements.

$$\nu \xrightarrow{\boxed{\texttt{x := expr}}} \nu[x := \nu(\texttt{expr})]$$

$$\nu \xrightarrow{\boxed{\texttt{cond}}} \nu \qquad \text{iff} \quad \nu(\texttt{cond}) = \textbf{true}.$$

Semantics of Statements

The meaning of the statements is given by a transition system.

- Valuations are the states of the transition system.
- Transitions are labelled with statements.

$$\nu \xrightarrow{\boxed{\texttt{x := expr}}} \nu[x := \nu(\texttt{expr})]$$

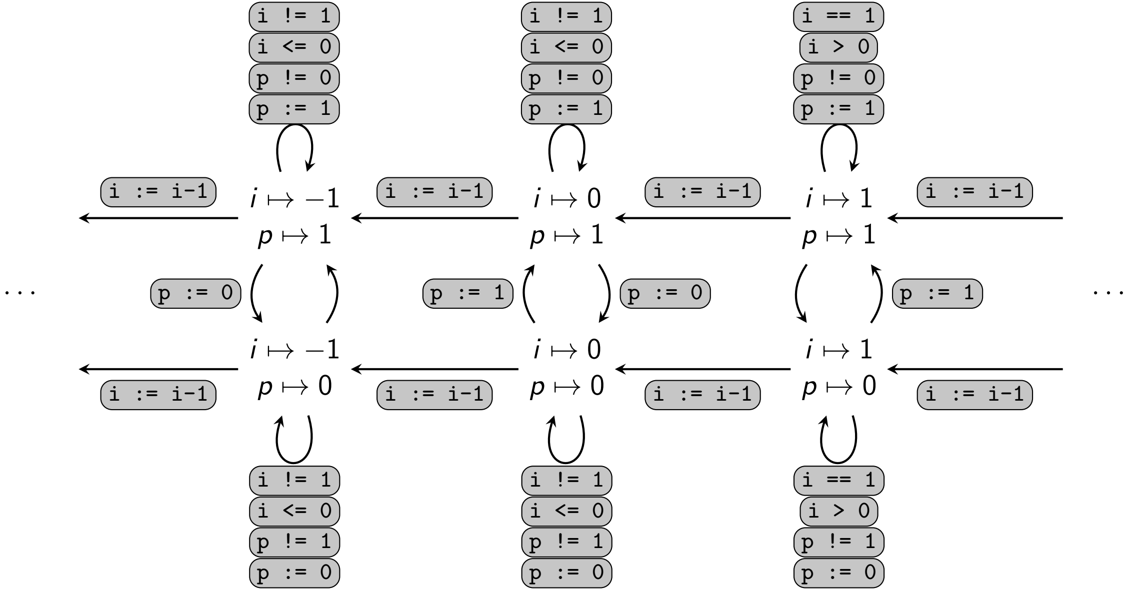
$$\nu \xrightarrow{\boxed{\texttt{cond}}} \nu \qquad \text{iff} \quad \nu(\texttt{cond}) = \textbf{true}.$$

Example: $\pi = \boxed{\texttt{i > 0}}\boxed{\texttt{p := 1}}\boxed{\texttt{i > 0}}\boxed{\texttt{p != 1}}$

$$\begin{array}{l} i \mapsto 1 \\ p \mapsto 0 \end{array} \xrightarrow{\boxed{\texttt{i > 0}}} \begin{array}{l} i \mapsto 1 \\ p \mapsto 0 \end{array} \xrightarrow{\boxed{\texttt{p := 1}}} \begin{array}{l} i \mapsto 1 \\ p \mapsto 1 \end{array} \xrightarrow{\boxed{\texttt{i > 0}}} \begin{array}{l} i \mapsto 1 \\ p \mapsto 1 \end{array} \not\xrightarrow{\boxed{\texttt{p != 1}}}$$

Transition System

The transition system is infinite and has infinitely many initial states.



Feasibility of Traces

Intuitively, there is no sequence of valuations for the trace:
$\pi =$ `i > 0` `p := 1` `i > 0` `p != 1`.
How can we show this?

Feasibility of Traces

Intuitively, there is no sequence of valuations for the trace:

$\pi =$ `i > 0` `p := 1` `i > 0` `p != 1`.

How can we show this?

SMT Solver!

Feasibility of Traces

Intuitively, there is no sequence of valuations for the trace:
$\pi =$ `i > 0` `p := 1` `i > 0` `p != 1`.
How can we show this?

SMT Solver!

- SSA (Single Static Assignment): copy the variable each time it is assigned.

$$\boxed{\texttt{i}_0\ \texttt{>}\ \texttt{0}}\boxed{\texttt{p}_1\ \texttt{:=}\ \texttt{1}}\boxed{\texttt{i}_0\ \texttt{>}\ \texttt{0}}\boxed{\texttt{p}_1\ \texttt{!=}\ \texttt{1}}$$

Feasibility of Traces

Intuitively, there is no sequence of valuations for the trace:
$\pi = \boxed{\texttt{i > 0}}\boxed{\texttt{p := 1}}\boxed{\texttt{i > 0}}\boxed{\texttt{p != 1}}$.
How can we show this?

SMT Solver!

- SSA (Single Static Assignment): copy the variable each time it is assigned.

$$\boxed{\texttt{i}_0\ \texttt{> 0}}\boxed{\texttt{p}_1\ \texttt{:= 1}}\boxed{\texttt{i}_0\ \texttt{> 0}}\boxed{\texttt{p}_1\ \texttt{!= 1}}$$

- Replace := by logical equality and conjunct all statements.

$$SSA(\pi) : \quad i_0 > 0 \land p_1 = 1 \land i_0 > 0 \land p_1 \neq 1$$

Feasibility of Traces

Intuitively, there is no sequence of valuations for the trace:
$\pi = \boxed{\texttt{i > 0}}\boxed{\texttt{p := 1}}\boxed{\texttt{i > 0}}\boxed{\texttt{p != 1}}$.
How can we show this?

SMT Solver!

- SSA (Single Static Assignment): copy the variable each time it is assigned.

$$\boxed{\texttt{i}_0\texttt{ > 0}}\boxed{\texttt{p}_1\texttt{ := 1}}\boxed{\texttt{i}_0\texttt{ > 0}}\boxed{\texttt{p}_1\texttt{ != 1}}$$

- Replace `:=` by logical equality and conjunct all statements.

$$SSA(\pi): \quad i_0 > 0 \land p_1 = 1 \land i_0 > 0 \land p_1 \neq 1$$

- Ask SMT solver, if there is a solution for the formula:

```
unsat
```

Excursion: SMT Solvers

SMT solvers are programs that decide satisfiability.
`Ultimate` uses z3, CVC4, mathsat and our own SMT solver `SMTInterpol`.

- Input a formula, for example:

$$i_0 > 0 \wedge p_1 = 1 \wedge i_1 = i_0 - 1 \wedge i_1 \leq 0$$

- Either sat (satisfiable) and optionally a model:

$$i_0 = 1, p_1 = 1, i_1 = 0$$

or unsat.

Demo: SMT Solvers

https://ultimate.informatik.uni-freiburg.de/smtinterpol/

```
(set-option :produce-models true)
(set-logic QF_LIA)
(declare-const i0 Int)
(declare-const p0 Int)
(declare-const i1 Int)
(declare-const p1 Int)

(assert (and (> i0 0)
             (= p1 1)
             (= i1 (- i0 1))
             (<= i1 0)))
(check-sat)
(get-model)
```

Naïve Model Checking Algorithm

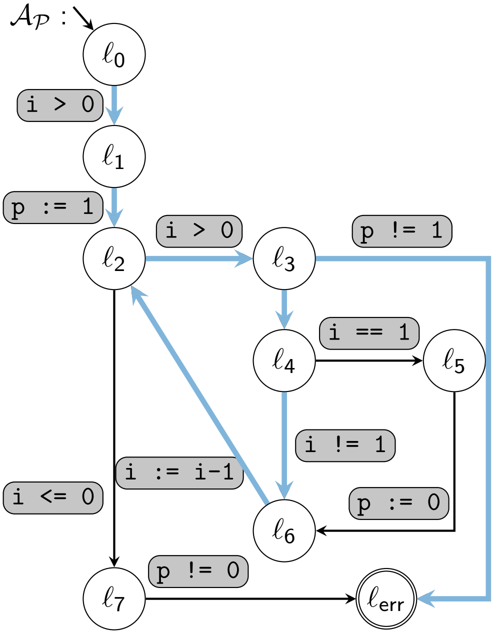
- Build program automaton.
- Collect error traces.
- For each error trace ask SMT solver.

Naïve Model Checking Algorithm

- Build program automaton.
- Collect error traces.
- For each error trace ask SMT solver.

Problem: There are infinitely many error traces.

Example: Error Traces

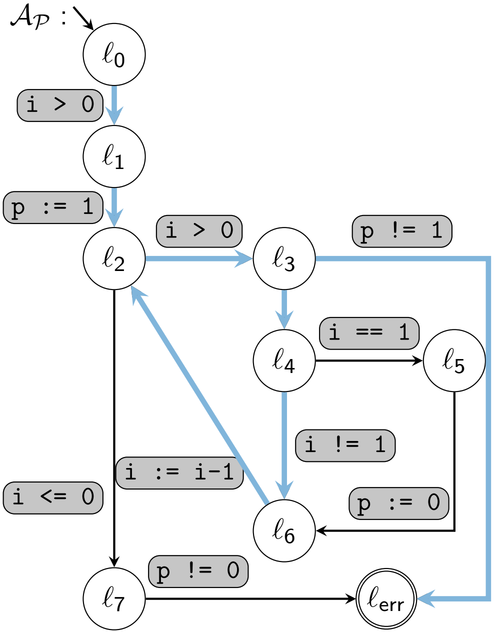


Some error traces:

```
i > 0 p := 1 i > 0 p != 1

i > 0 p := 1
      i > 0 i != 1 i := i-1
      i > 0 p != 1

i > 0 p := 1
      i > 0 i != 1 i := i-1
      i > 0 i != 1 i := i-1
      i > 0 p != 1
```

⋮

Example: Error Traces



Some error traces:

`i > 0` `p := 1` `i > 0` `p != 1`

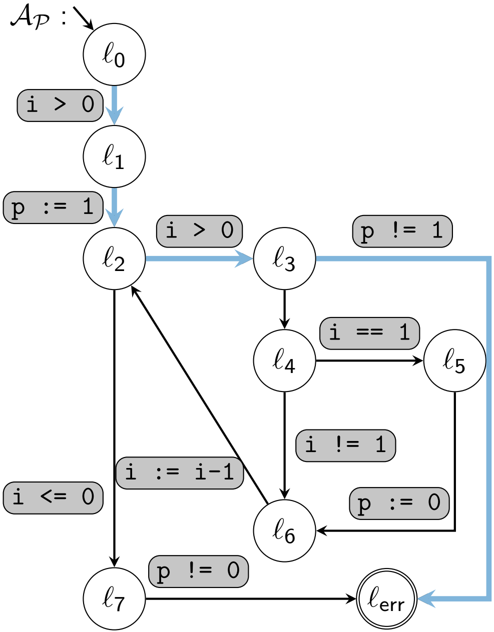
`i > 0` `p := 1`
`i > 0` `i != 1` `i := i-1`
`i > 0` `p != 1`

`i > 0` `p := 1`
`i > 0` `i != 1` `i := i-1`
`i > 0` `i != 1` `i := i-1`
`i > 0` `p != 1`

⋮

All infeasible for the same reason.

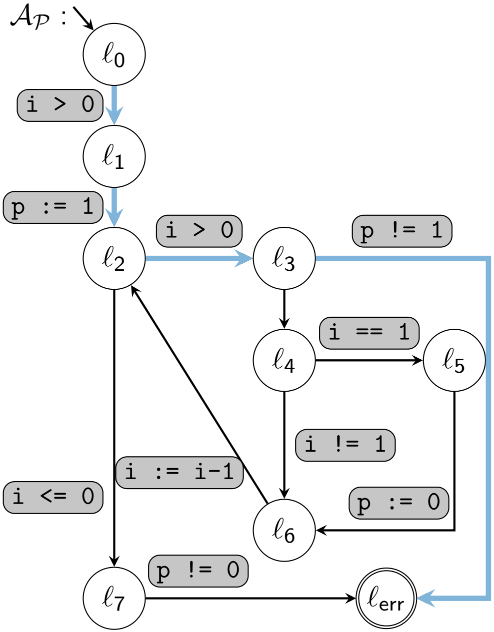
Trace Abstraction



Observation

Every trace . . . `p := 1` . . . `p != 1` . . . is infeasible, as long as there is no statement `p := 0` in the middle

Trace Abstraction

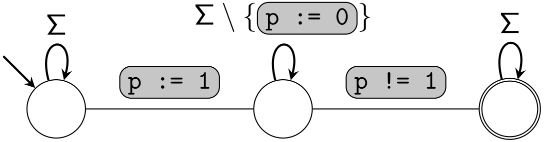


Observation

Every trace ...`p := 1`...`p != 1`... is infeasible, as long as there is no statement `p := 0` in the middle

Traces can be described by a finite automaton:

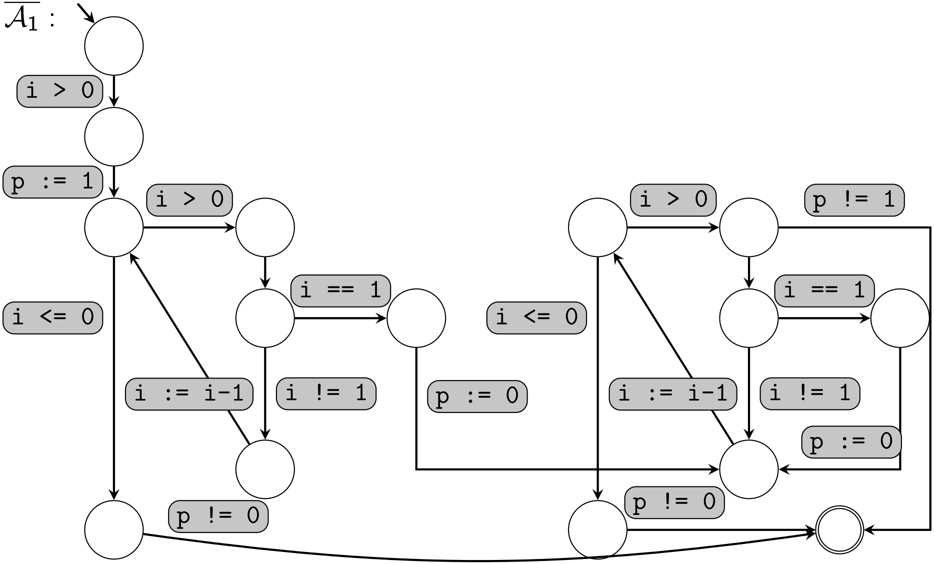
$\mathcal{A}_1$:



Subtracting Finite Automata from Each Other

There are algorithm to complement and intersect finite automata.

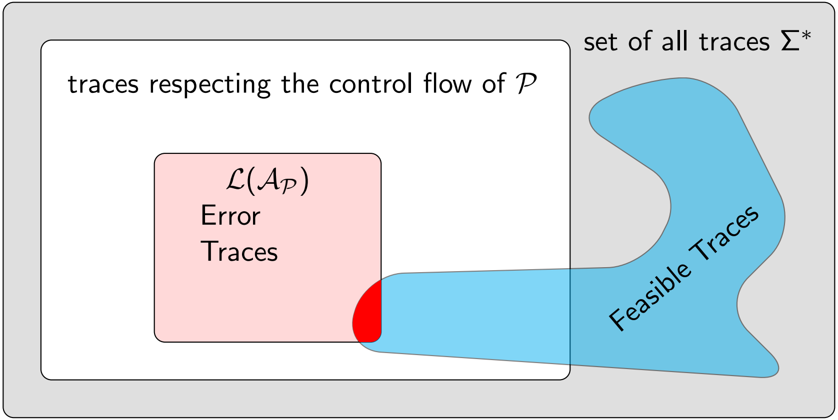
$\mathcal{A}_{\mathcal{P}} \cap \overline{\mathcal{A}_1}$:



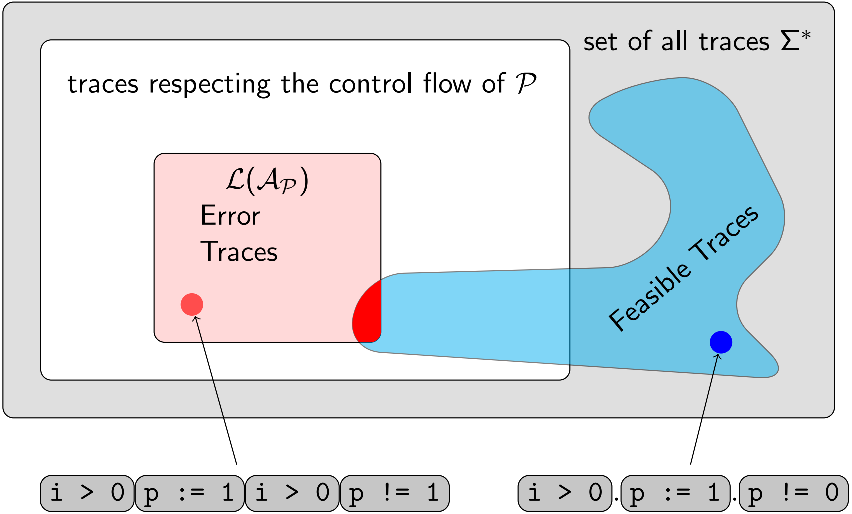
Model Checking with Trace Abstraction

1. Build program automaton.
2. Pick an error traces. If none, program is safe.
3. Ask SMT solver. If sat, program is unsafe.
4. Generalize error trace to an automaton.
5. Subtract from program automaton.
6. Go to step 2.

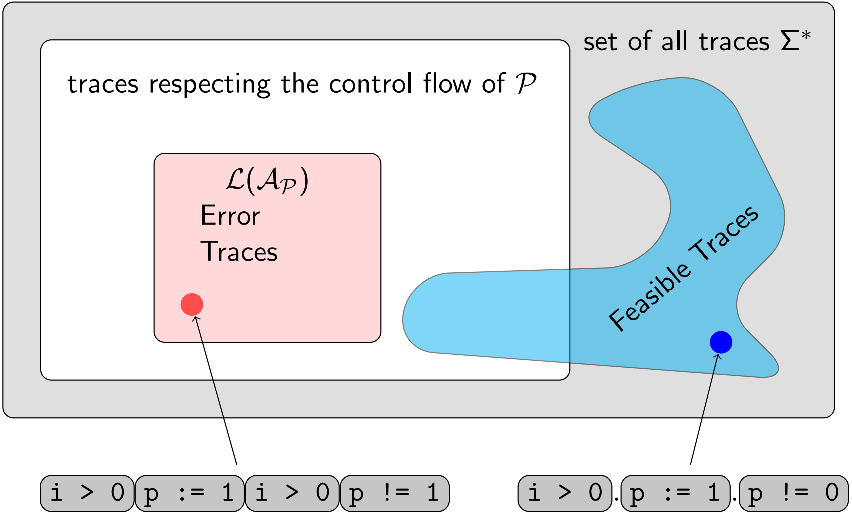
Set Theoretic View of Trace Abstraction



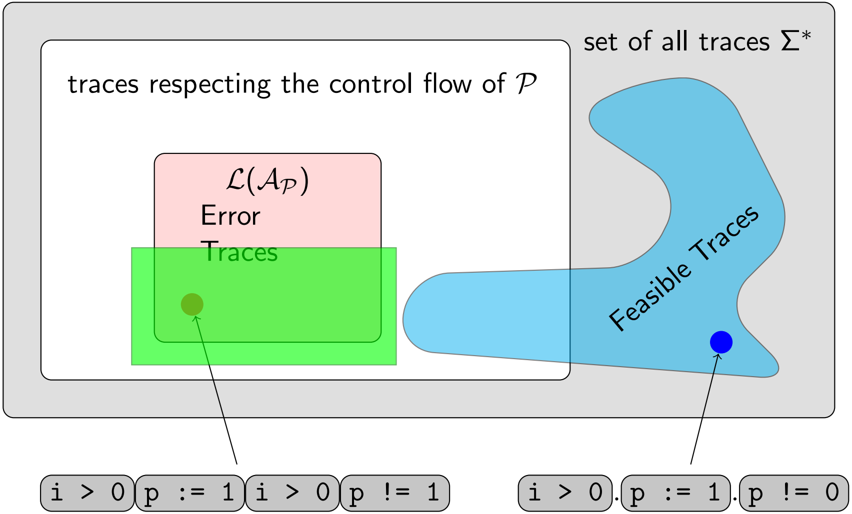
Set Theoretic View of Trace Abstraction



Set Theoretic View of Trace Abstraction



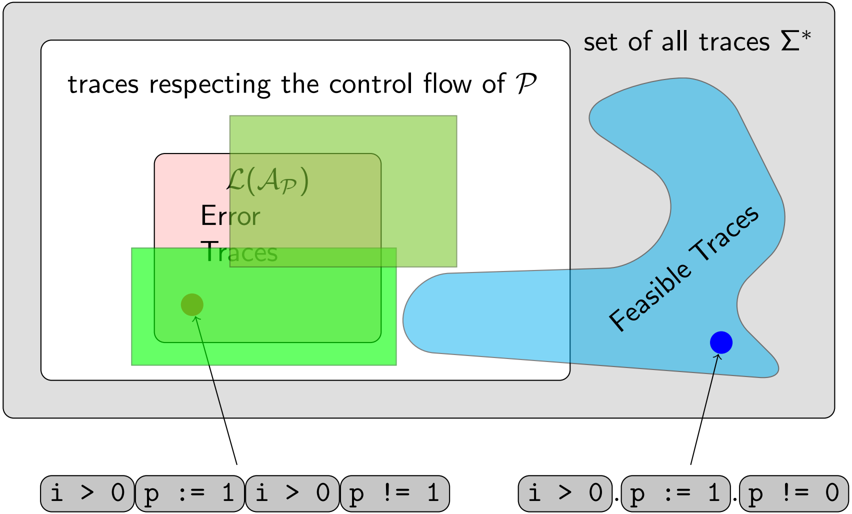
Set Theoretic View of Trace Abstraction



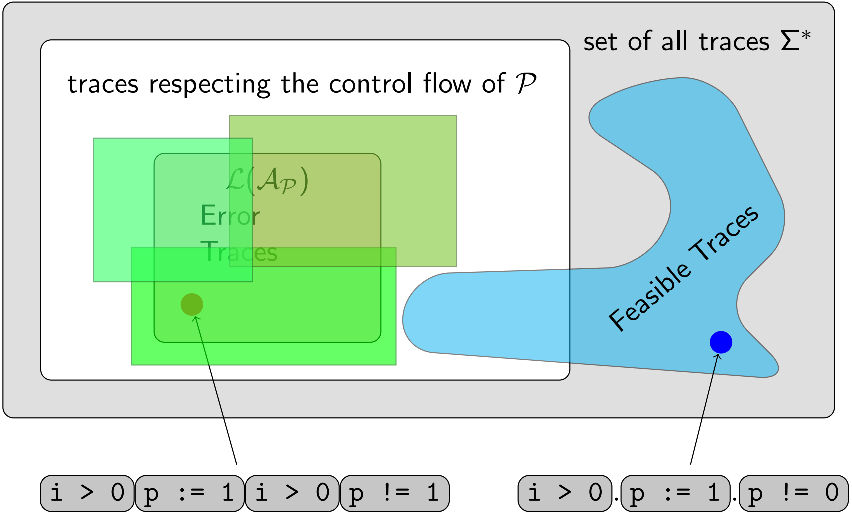
`i > 0` `p := 1` `i > 0` `p != 1`

`i > 0`.`p := 1`.`p != 0`

Set Theoretic View of Trace Abstraction



Set Theoretic View of Trace Abstraction



`i > 0` `p := 1` `i > 0` `p != 1`

`i > 0`.`p := 1`.`p != 0`

Trace Abstraction

Definition (Trace Abstraction)

A trace abstraction is given by a tuple of automata $(\mathcal{A}_1, \ldots, \mathcal{A}_n)$ such that each $\mathcal{A}_i$ recognizes a subset of infeasible traces, for $i = 1, \ldots, n$.

We say that the trace abstraction $(\mathcal{A}_1, \ldots, \mathcal{A}_n)$ does not admit an error trace if $\mathcal{A}_\mathcal{P} \cap \overline{\mathcal{A}_1} \cap \ldots \cap \overline{\mathcal{A}_n}$ is empty.

Trace Abstraction

Definition (Trace Abstraction)

A trace abstraction is given by a tuple of automata $(\mathcal{A}_1, \ldots, \mathcal{A}_n)$ such that each $\mathcal{A}_i$ recognizes a subset of infeasible traces, for $i = 1, \ldots, n$.

We say that the trace abstraction $(\mathcal{A}_1, \ldots, \mathcal{A}_n)$ does not admit an error trace if $\mathcal{A}_\mathcal{P} \cap \overline{\mathcal{A}_1} \cap \ldots \cap \overline{\mathcal{A}_n}$ is empty.

Theorem (Soundness)

$$\mathcal{L}(\mathcal{A}_\mathcal{P} \cap \overline{\mathcal{A}_1} \cap \ldots \cap \overline{\mathcal{A}_n}) = \emptyset \qquad \Rightarrow \qquad \mathcal{P} \textit{ is correct}$$

Theorem (Completeness)

If $\mathcal{P}$ is correct, there is a trace abstraction $(\mathcal{A}_1, \ldots, \mathcal{A}_n)$ such that

$$\mathcal{L}(\mathcal{A}_\mathcal{P} \cap \overline{\mathcal{A}_1} \cap \ldots \cap \overline{\mathcal{A}_n}) = \emptyset$$

How to Get a Trace Abstraction?

Naïve Approach:

Exclude infeasible error traces.

. . . but there are infinitely many.

Interpolant Based Approach:

Generalize infeasible error traces.

Exclude classes of infeasible traces.

How to Get a Trace Abstraction?

Naïve Approach:

Exclude infeasible error traces.

. . . but there are infinitely many.

Interpolant Based Approach:

Generalize infeasible error traces.

Exclude classes of infeasible traces.

Interpolants

Interpolants for Infeasible Traces

Let $st_1 \land \cdots \land st_n$ be an infeasible trace. There exists a sequence of predicates $I_0, \ldots, I_n$ such that

$$I_0 = \textbf{true} \qquad I_i \land st_{i+1} \Rightarrow I_{i+1} \qquad I_n = \textbf{false}$$

In particular:

$$st_1 \land \cdots \land st_i \Rightarrow I_i \Rightarrow \neg(st_{i+1} \land \cdots \land st_n)$$

Example:

$$\boxed{\textbf{true}} \quad i_0 > 0 \quad \boxed{\textbf{true}} \quad p_1 = 1 \quad \boxed{p_1 = 1} \quad i_0 > 0 \quad \boxed{p_1 = 1} \quad p_1 \neq 1 \quad \boxed{\textbf{false}}$$

Interpolants

Interpolants for Infeasible Traces

Let $st_1 \wedge \cdots \wedge st_n$ be an infeasible trace. There exists a sequence of predicates $I_0, \ldots, I_n$ such that

$$I_0 = \textbf{true} \qquad I_i \wedge st_{i+1} \Rightarrow I_{i+1} \qquad I_n = \textbf{false}$$

In particular:

$$st_1 \wedge \cdots \wedge st_i \Rightarrow I_i \Rightarrow \neg(st_{i+1} \wedge \cdots \wedge st_n)$$

Example:

$$\textbf{true} \quad i_0 > 0 \quad \textbf{true} \quad p_1 = 1 \quad p_1 = 1 \quad i_0 > 0 \quad p_1 = 1 \quad p_1 \neq 1 \quad \textbf{false}$$

Interpolants are intermediate assertions in a Hoare proof.

Interpolants as Hoare Proofs

true $i_0 > 0$ **true** $p_1 = 1$ $p_1 = 1$ $i_0 > 0$ $p_1 = 1$ $p_1 \neq 1$ **false**

Interpolants as Hoare Proofs

true $i_0 > 0$ **true** $p_1 = 1$ $p_1 = 1$ $i_0 > 0$ $p_1 = 1$ $p_1 \neq 1$ **false**

$\{\textbf{true}\}$ `i > 0` $\{\textbf{true}\}$
$\{\textbf{true}\}$ `p := 1` $\{p = 1\}$
$\{p = 1\}$ `i > 0` $\{p = 1\}$
$\{p = 1\}$ `p != 1` $\{\textbf{false}\}$

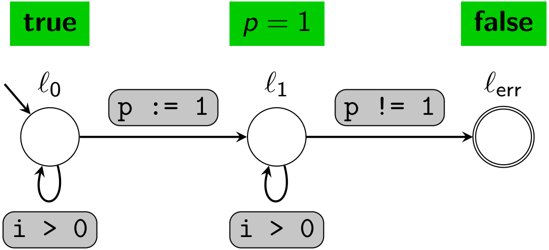
This proves that the trace is infeasible:

$\{\textbf{true}\}$ `i > 0` `p := 1` `i > 0` `p != 1` $\{\textbf{false}\}$

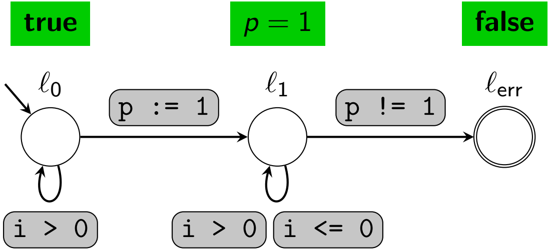
Demo: Computing Interpolants with SMTInterpol

```
(set-option :produce-interpolants true)
(set-logic QF_LIA)
(declare-const i0 Int)
(declare-const p0 Int)
(declare-const i1 Int)
(declare-const p1 Int)

(assert (! (> i0 0) :name st1))
(assert (! (= p1 1) :name st2))
(assert (! (> i0 0) :name st3))
(assert (! (not (= p1 1)) :name st4))

(check-sat)
(get-interpolants st1 st2 st3 st4)
```

Example – Use Interpolants to Generalize Infeasible Traces

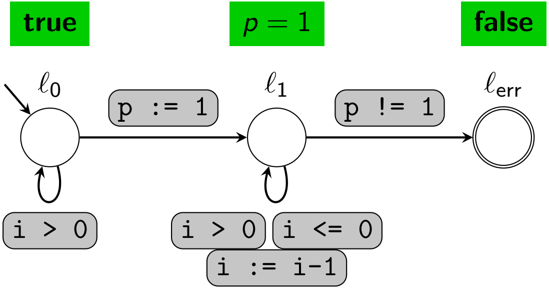


Example – Use Interpolants to Generalize Infeasible Traces



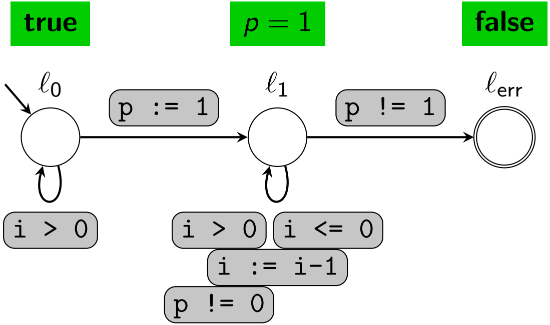
$\{p = 1\}$ `i <= 0` $\{p = 1\}$

Example – Use Interpolants to Generalize Infeasible Traces



$\{p = 1\}$ `i <= 0` $\{p = 1\}$ $\{p = 1\}$ `i := i-1` $\{p = 1\}$

Example – Use Interpolants to Generalize Infeasible Traces

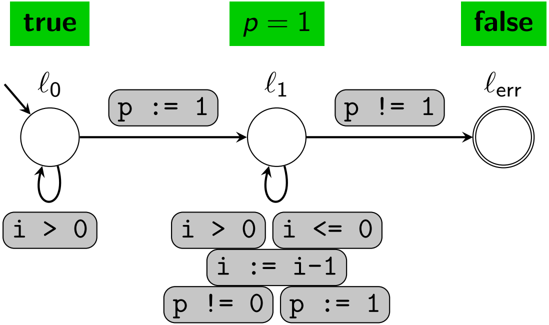


$\{p = 1\}$ `i <= 0` $\{p = 1\}$

$\{p = 1\}$ `p != 0` $\{p = 1\}$

$\{p = 1\}$ `i := i-1` $\{p = 1\}$

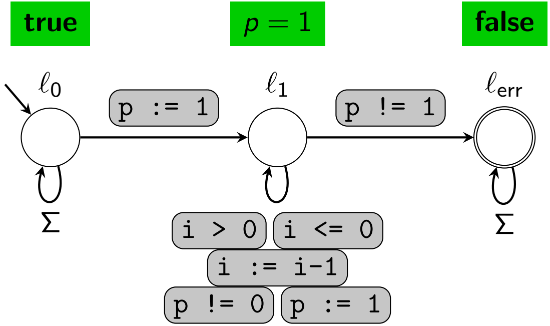
Example – Use Interpolants to Generalize Infeasible Traces



$\{p = 1\}$ `i <= 0` $\{p = 1\}$ $\{p = 1\}$ `i := i-1` $\{p = 1\}$

$\{p = 1\}$ `p != 0` $\{p = 1\}$ $\{p = 1\}$ `p := 1` $\{p = 1\}$

Example – Use Interpolants to Generalize Infeasible Traces



$\{p = 1\}$ `i <= 0` $\{p = 1\}$ $\{p = 1\}$ `i := i-1` $\{p = 1\}$

$\{p = 1\}$ `p != 0` $\{p = 1\}$ $\{p = 1\}$ `p := 1` $\{p = 1\}$

Interpolant Automata

Given: Sequence of interpolants $\mathcal{I} = I_0, I_1, \ldots, I_n$

Definition (Interpolant Automaton $\mathcal{A}_\mathcal{I}$)

$$\mathcal{A}_\mathcal{I} = \langle Q_\mathcal{I}, \delta_\mathcal{I}, Q_\mathcal{I}^{\text{init}}, Q_\mathcal{I}^{\text{fin}} \rangle \qquad Q_\mathcal{I} = \mathcal{I}$$

$$(I_i, st, I_j) \in \delta_\mathcal{I} \quad \text{iff} \quad \{I_i\}\ st\ \{I_j\} \text{ holds}$$

$$q_0 := \textbf{true} \in Q_\mathcal{I}$$

$$Q^{\text{fin}} := \{\textbf{false}\} \subseteq Q_\mathcal{I}$$

Interpolant Automata

Given: Sequence of interpolants $\mathcal{I} = I_0, I_1, \ldots, I_n$

Definition (Interpolant Automaton $\mathcal{A}_\mathcal{I}$)

$$\mathcal{A}_\mathcal{I} = \langle Q_\mathcal{I}, \delta_\mathcal{I}, Q_\mathcal{I}^{\text{init}}, Q_\mathcal{I}^{\text{fin}} \rangle \qquad\qquad Q_\mathcal{I} = \mathcal{I}$$

$$(I_i, st, I_j) \in \delta_\mathcal{I} \quad \text{iff} \quad \{I_i\}\; st\; \{I_j\} \text{ holds}$$

$$q_0 := \textbf{true} \in Q_\mathcal{I}$$

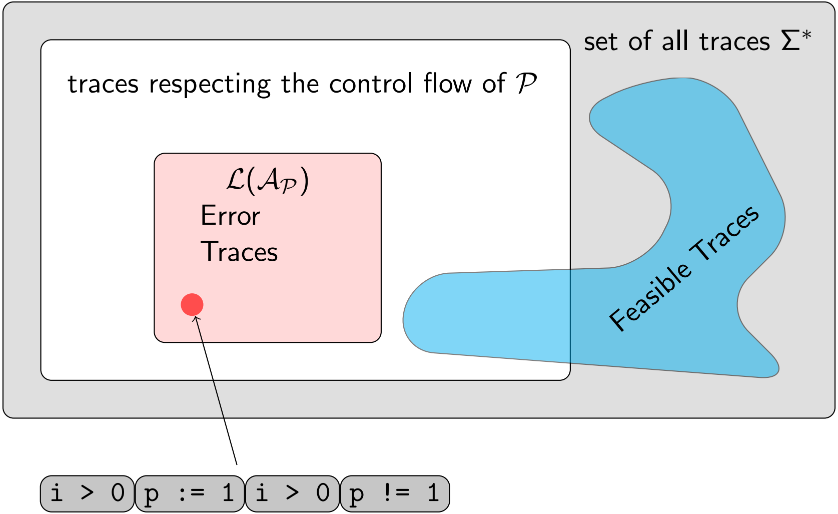
$$Q^{\text{fin}} := \{\textbf{false}\} \subseteq Q_\mathcal{I}$$

Theorem

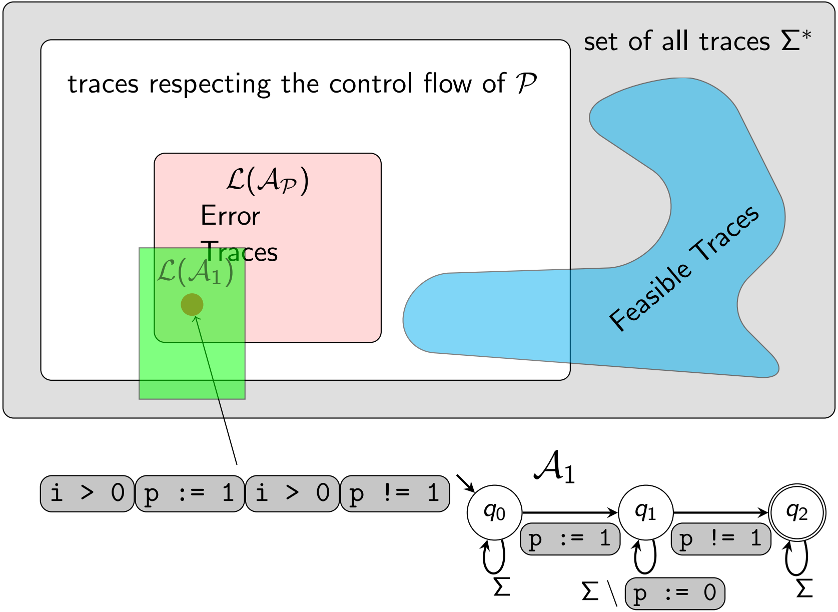
An interpolant automaton $\mathcal{A}_\mathcal{I}$ recognizes a subset of infeasible traces.

$$\mathcal{L}(\mathcal{A}_\mathcal{I}) \subseteq \mathit{Infeasible}$$

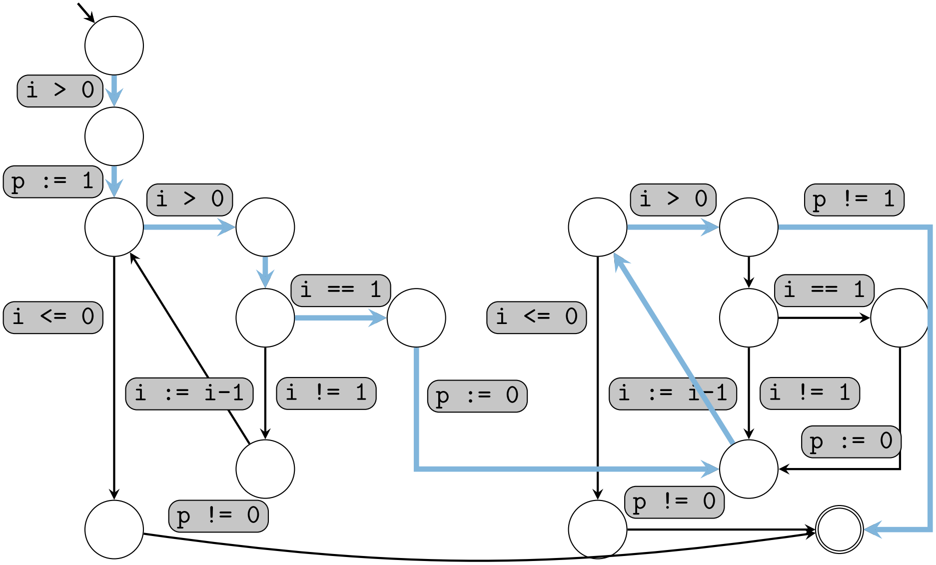
Example – Refinement Using Interpolant Automata



Example – Refinement Using Interpolant Automata



Remaining Program Automaton

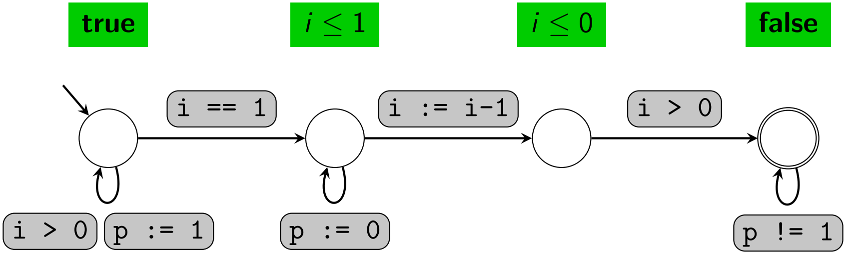


Demo: SMTInterpol

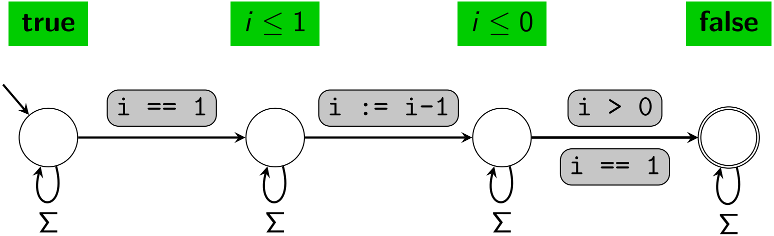
`i > 0` `p := 1` `i > 0` `i == 1` `p := 0` `i := i-1` `i > 0` `p != 1`

```
(set-option :produce-interpolants true)
(set-logic QF_LIA)
(declare-const i0 Int)
(declare-const i1 Int)
(declare-const p0 Int)
(declare-const p1 Int)
(declare-const p2 Int)

(assert (! (...) :named st1))
...

(check-sat)
(get-interpolants st1 st2 st3 ...)
```

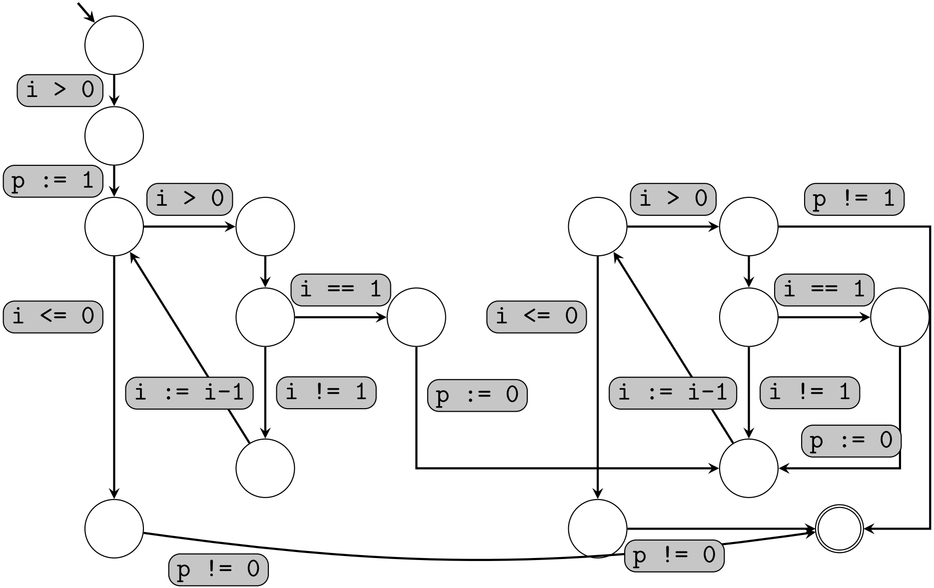
Interpolant Automaton for Second Trace



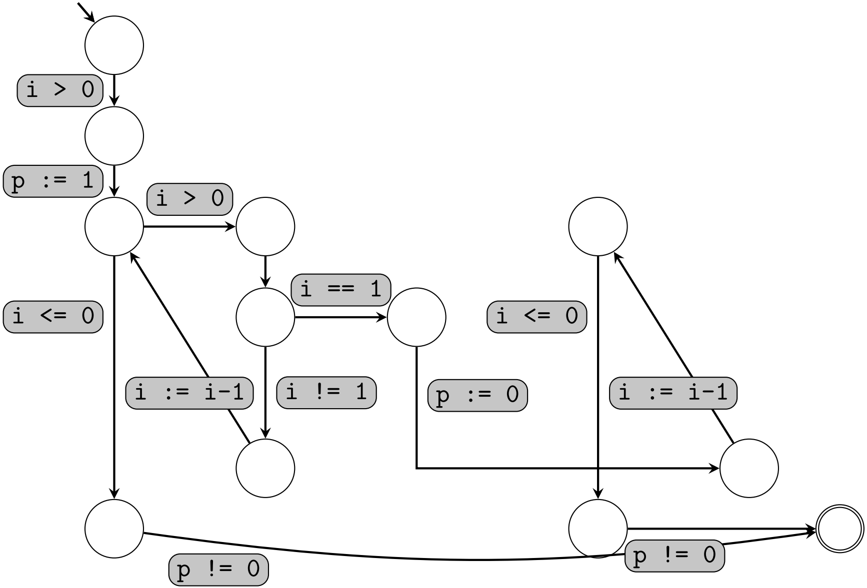
Interpolant Automaton for Second Trace



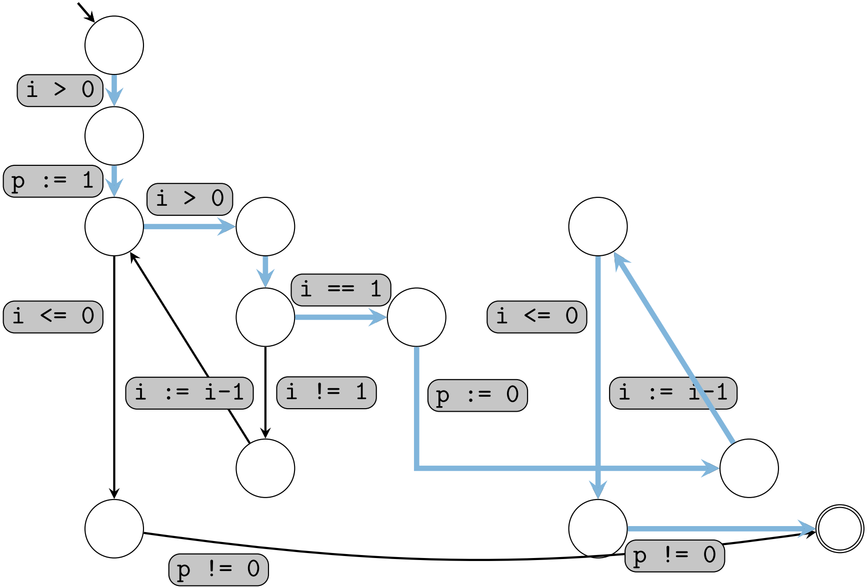
Remaining Program Automaton



Remaining Program Automaton



Remaining Program Automaton



Demo: SMTInterpol

`i > 0` `p := 1` `i > 0` `i == 1` `p := 0` `i := i-1` `i <= 0` `p != 0`

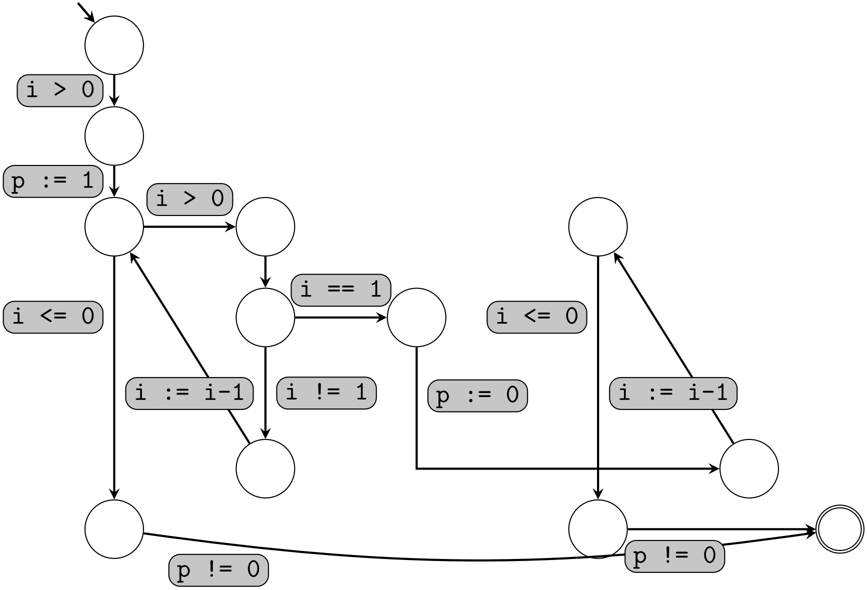
```
(set-option :produce-interpolants true)
(set-logic QF_LIA)
(declare-const i0 Int)
(declare-const i1 Int)
(declare-const p0 Int)
(declare-const p1 Int)
(declare-const p2 Int)

(assert (! (...) :named st1))
...

(check-sat)
(get-interpolants st1 st2 st3 ...)
```

Demo: SMTInterpol

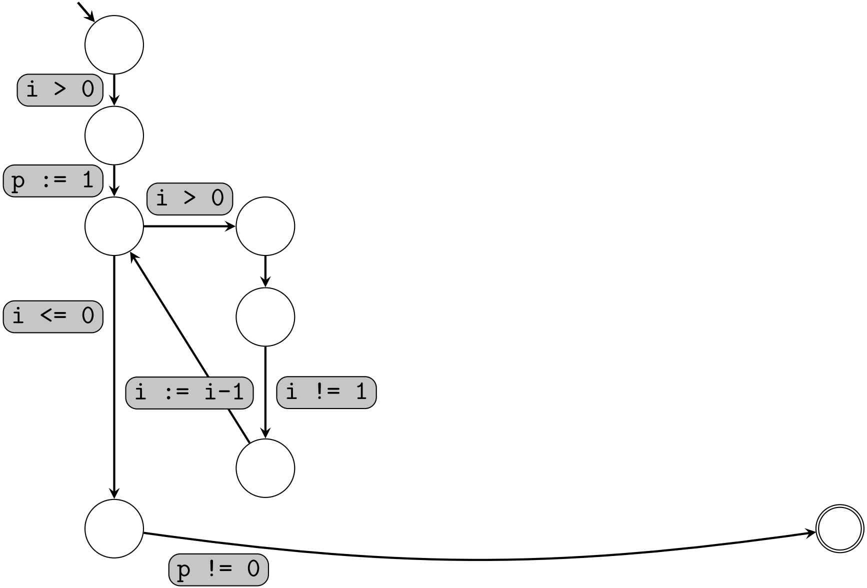
`i > 0` `p := 1` `i > 0` `i == 1` `p := 0` `i := i-1` `i <= 0` `p != 0`

```
(set-option :produce-interpolants true)
(set-logic QF_LIA)
(declare-const i0 Int)
(declare-const i1 Int)
(declare-const p0 Int)
(declare-const p1 Int)
(declare-const p2 Int)

(assert (! (...) :named st1))
...

(check-sat)
(get-interpolants st1 st2 st3 ...)

;;(true true true true (= p2 0) (= p2 0) (= p2 0))
```

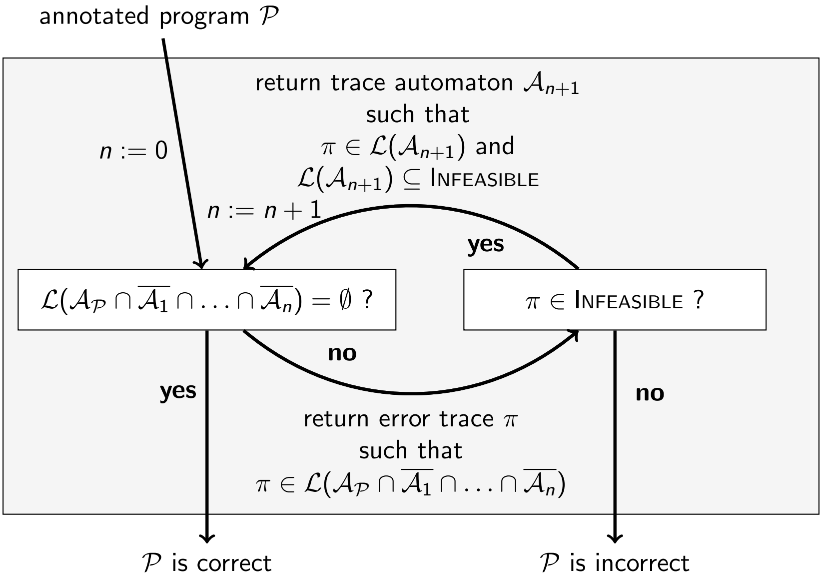
Remaining Program Automaton



Remaining Program Automaton



CEGAR for Trace Abstraction



annotated program $\mathcal{P}$

$n := 0$

return trace automaton $\mathcal{A}_{n+1}$
such that
$\pi \in \mathcal{L}(\mathcal{A}_{n+1})$ and
$\mathcal{L}(\mathcal{A}_{n+1}) \subseteq \textsc{Infeasible}$

$n := n + 1$

yes

$\mathcal{L}(\mathcal{A}_\mathcal{P} \cap \overline{\mathcal{A}_1} \cap \ldots \cap \overline{\mathcal{A}_n}) = \emptyset$?

$\pi \in \textsc{Infeasible}$?

no

yes

no

return error trace π
such that
$\pi \in \mathcal{L}(\mathcal{A}_\mathcal{P} \cap \overline{\mathcal{A}_1} \cap \ldots \cap \overline{\mathcal{A}_n})$

$\mathcal{P}$ is correct

$\mathcal{P}$ is incorrect

Recursive Function

Example: McCarthy 91 Function

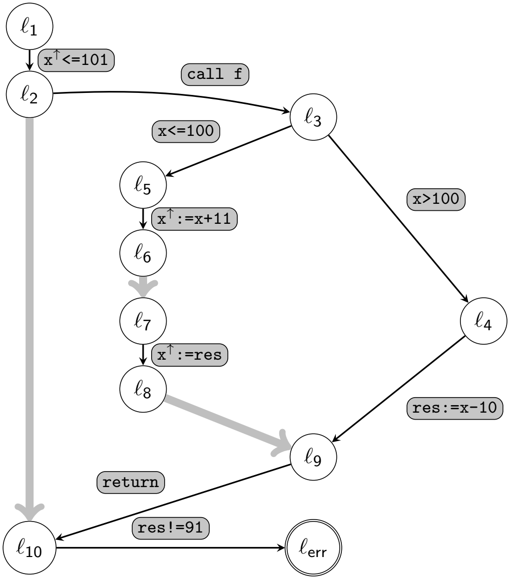
```
int f91(int x) {
    if (x > 100)
       return x - 10;
    else
       return f91(f91(x + 11));
}
```

Example: McCarthy 91 Function

```
int f91(int x) {
    if (x > 100)
       return x - 10;
    else
       return f91(f91(x + 11));
}

int main(int x) {
    int res;
    if (x <= 101) {
        res = f91(x);
        //@assert(res == 91);
    }
}
```

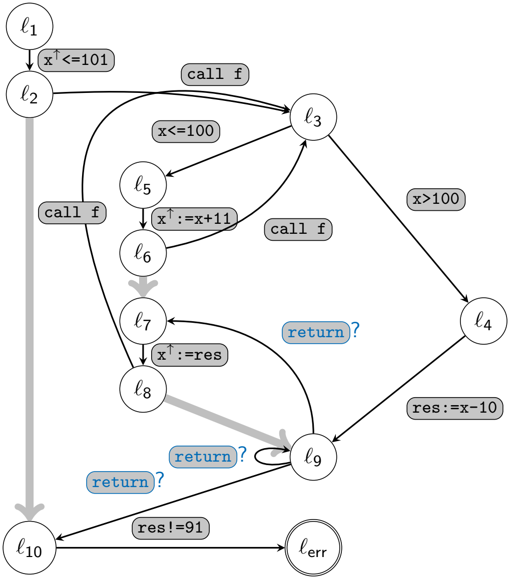
McCarthy91 as Automaton

```
      f(x) {
ℓ3:     if (x > 100) {
ℓ4:       res := x - 11
        } else {
ℓ5:       x↑:= x + 10
ℓ6:       call f
ℓ7:       x↑:= res
ℓ8:       call f
        }
ℓ9:     return res
      }
      main() {
ℓ1:     if (x↑ <= 101) {
ℓ2:       call f
ℓ10:      assert(res == 91)
        }
      }
```



McCarthy91 as Automaton

```
             f(x) {
$\ell_3$:      if (x > 100) {
$\ell_4$:        res := x - 11
               } else {
$\ell_5$:        x$^\uparrow$:= x + 10
$\ell_6$:        call f
$\ell_7$:        x$^\uparrow$:= res
$\ell_8$:        call f
               }
$\ell_9$:      return res
             }

             main() {
$\ell_1$:      if (x$^\uparrow$ <= 101) {
$\ell_2$:        call f
$\ell_{10}$:     assert(res == 91)
               }
             }
```

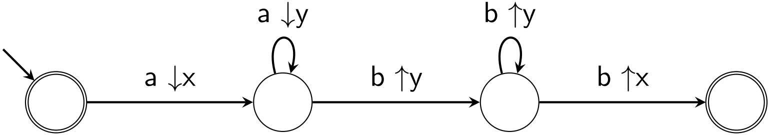


Reminder Push-down Automaton

A push-down automaton $\mathcal{A} = (\Sigma, \Gamma, Q, \rightarrow, q_0, F)$ consists of

- Σ: a finite alphabet
- Γ: a stack alphabet
- Q: a finite set of locations
- $\rightarrow \subseteq Q \times \Sigma \times Op \times Q$: a transition relation,
  where Op is a stack operation: $\downarrow\gamma$ (push), $\uparrow\gamma$ (pop), or none.
- $q_0 \in Q$: the initial location
- $F \subseteq Q$: the accepting locations

Example:

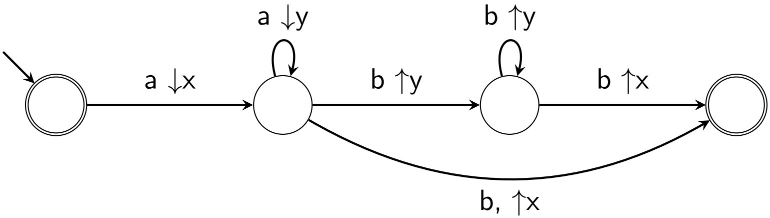


Reminder Push-down Automaton

A push-down automaton $\mathcal{A} = (\Sigma, \Gamma, Q, \rightarrow, q_0, F)$ consists of

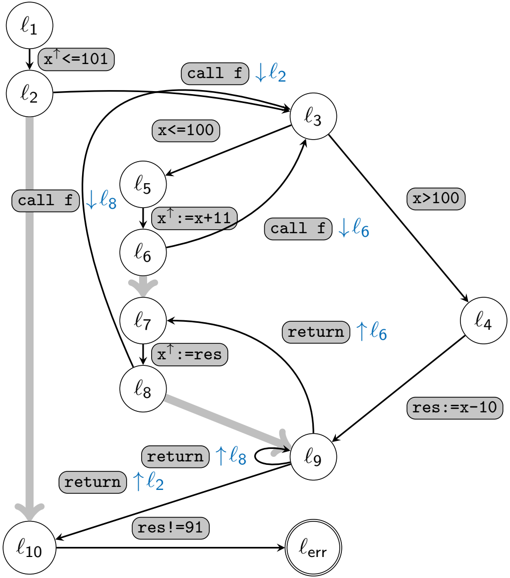
- Σ: a finite alphabet
- Γ: a stack alphabet
- Q: a finite set of locations
- $\rightarrow \subseteq Q \times \Sigma \times Op \times Q$: a transition relation,
  where Op is a stack operation: $\downarrow\gamma$ (push), $\uparrow\gamma$ (pop), or none.
- $q_0 \in Q$: the initial location
- $F \subseteq Q$: the accepting locations

Example:



McCarthy as Push-down Automaton

```
       f(x) {
ℓ3:      if (x > 100) {
ℓ4:        res := x - 11
         } else {
ℓ5:        x↑:= x + 10
ℓ6:        call f
ℓ7:        x↑:= res
ℓ8:        call f
         }
ℓ9:      return res
       }

       main() {
ℓ1:      if (x↑ <= 101) {
ℓ2:        call f
ℓ10:       assert(res == 91)
         }
       }
```



Trace Abstraction with Push-down Automaton?

Problem

Push-down Automata can't be subtracted/complemented/intersected.

Trace Abstraction with Push-down Automaton?

Problem

Push-down Automata can't be subtracted/complemented/intersected.

Solution: Alur & Madhusudan: Visibly Push-down Languages, 2004

- Closed under complementation, intersection
- The symbol decides whether to push, pop, or do nothing
- Suitable for call/return statements.

Visibly Push-down Automaton

A visibly push-down automaton $\mathcal{A} = (\Sigma_i, \Sigma_c, \Sigma_r \Gamma, Q, \rightarrow, q_0, F)$ consists of

- $\Sigma_i, \Sigma_c, \Sigma_r$: three distinct finite alphabet for internal, call, and return statements.
- Γ: a stack alphabet
- Q: a finite set of locations
- $$\rightarrow \subseteq \left(\begin{array}{cc} & Q \times \Sigma_i \times Q \\ \cup & Q \times \Sigma_c \times \downarrow \Gamma \times Q \\ \cup & Q \times \Sigma_r \times \uparrow \Gamma \times Q \end{array} \right).$$
  Call statements always push a value, return statements always pop a value, and internal statements do not change stack.
- $q_0 \in Q$: the initial location
- $F \subseteq Q$: the accepting locations

Visibly Push-down Automaton

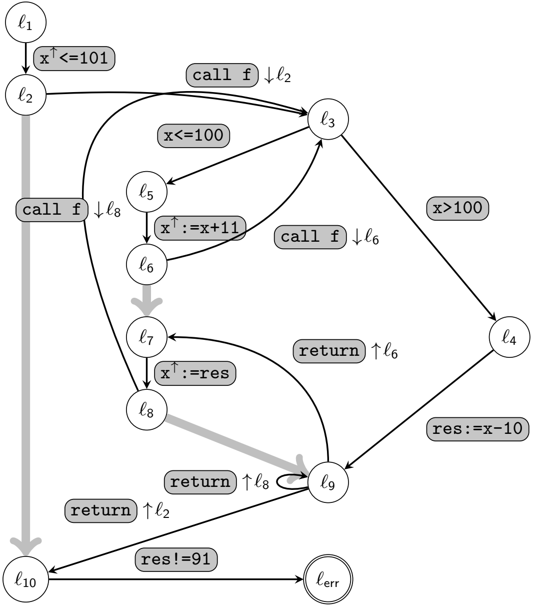
A visibly push-down automaton $\mathcal{A} = (\Sigma_i, \Sigma_c, \Sigma_r \Gamma, Q, \rightarrow, q_0, F)$ consists of

- $\Sigma_i, \Sigma_c, \Sigma_r$: three distinct finite alphabet for internal, call, and return statements.
- Γ: a stack alphabet
- Q: a finite set of locations
- $\rightarrow \subseteq \left(\begin{array}{cc} & Q \times \Sigma_i \times Q \\ \cup & Q \times \Sigma_c \times \downarrow \Gamma \times Q \\ \cup & Q \times \Sigma_r \times \uparrow \Gamma \times Q \end{array} \right).$
  Call statements always push a value, return statements always pop a value, and internal statements do not change stack.
- $q_0 \in Q$: the initial location
- $F \subseteq Q$: the accepting locations

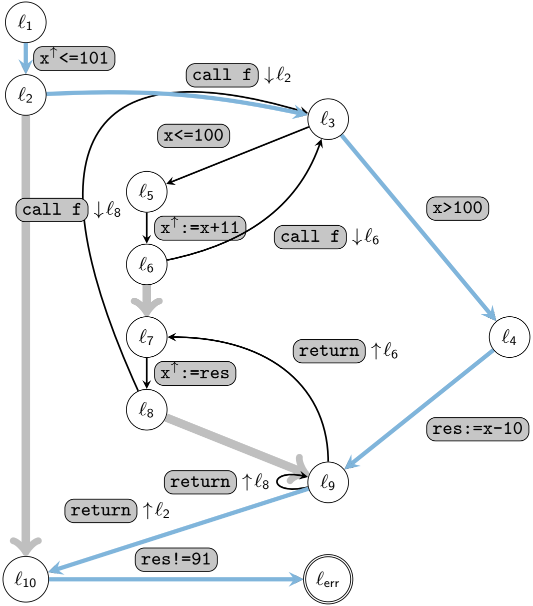
Remark

Nested word automata have equivalent power, differ only in details.

Error Traces



Error Traces



Error Trace:

x$^\uparrow$<=101 call f x>100 res:=x-10
return res!=91

SSA for Recursive Traces

- Call statements copy parameters, e.g.

$$\boxed{\texttt{call f}} \longrightarrow x_1 = x_0^{\uparrow}$$

- Return statements do nothing.

$$\boxed{\texttt{return}} \longrightarrow \textbf{true}$$

- SSA numbering must obey scoping rules, e.g.,

$$\boxed{\texttt{x}^{\uparrow}\texttt{:=x}}\boxed{\texttt{call f(x)}}\boxed{\texttt{x:=x+1}}\boxed{\texttt{res:=x}}\boxed{\texttt{return}}\boxed{\texttt{z:=x+res}}$$

$$\downarrow$$

$$x_1^{\uparrow} = x_0 \land x_1 = x_1^{\uparrow} \land x_2 = x_1 + 1 \land res_1 = x_2 \land z_1 = x_? + res_?$$

SSA for Recursive Traces

- Call statements copy parameters, e.g.

$$\boxed{\texttt{call f}} \longrightarrow x_1 = x_0^{\uparrow}$$

- Return statements do nothing.

$$\boxed{\texttt{return}} \longrightarrow \textbf{true}$$

- SSA numbering must obey scoping rules, e.g.,

$$\boxed{\texttt{x}^{\uparrow}\texttt{:=x}}\boxed{\texttt{call f(x)}}\boxed{\texttt{x:=x+1}}\boxed{\texttt{res:=x}}\boxed{\texttt{return}}\boxed{\texttt{z:=x+res}}$$

$$\downarrow$$

$$x_1^{\uparrow} = x_0 \wedge x_1 = x_1^{\uparrow} \wedge x_2 = x_1 + 1 \wedge res_1 = x_2 \wedge z_1 = x_0 + res_1$$

Error Trace to SSA

Error Trace:

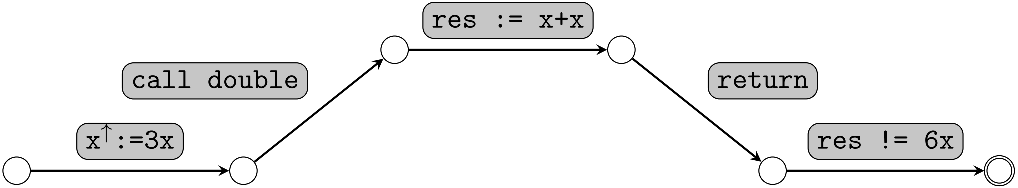
$$\pi = \boxed{\texttt{x}^{\uparrow}\texttt{<=101}}\boxed{\texttt{call f}}\boxed{\texttt{x>100}}\boxed{\texttt{res:=x-10}}\boxed{\texttt{return}}\boxed{\texttt{res!=91}}$$

SSA:

$$SSA(\pi) = x_0^{\uparrow} \leq 101 \land x_1 = x_0^{\uparrow} \land x_1 > 100 \land res_1 = x_1 - 10nn \land res_1 \neq 91$$

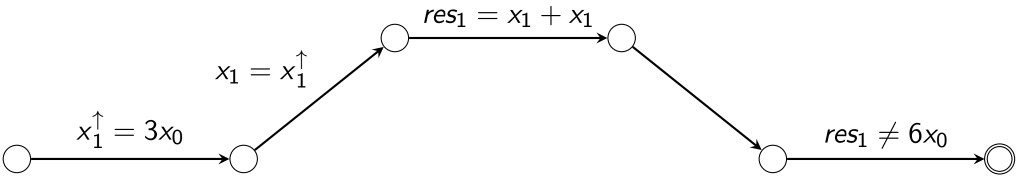
Interpolation for Recursive Traces

```
int main(x) { assert double(3*x) == 6*x; }
int double(x) { return x+x; }
```



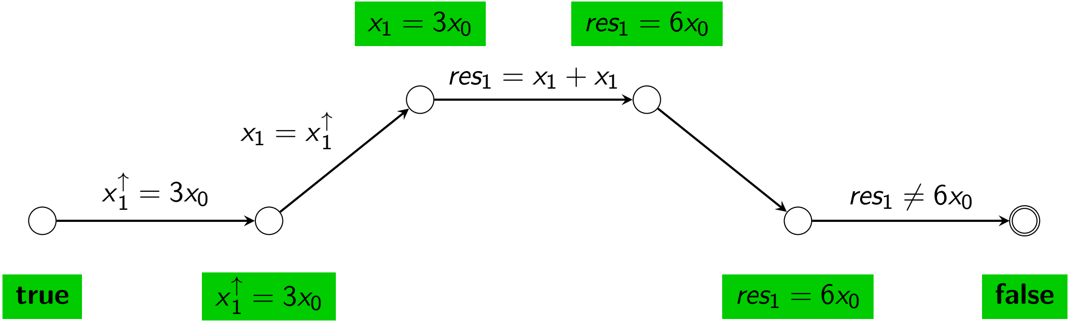
Interpolation for Recursive Traces

```
int main(x) { assert double(3*x) == 6*x; }
int double(x) { return x+x; }
```



Interpolation for Recursive Traces

```
int main(x) { assert double(3*x) == 6*x; }
int double(x) { return x+x; }
```



$x_1 = 3x_0$

$res_1 = 6x_0$

$res_1 = x_1 + x_1$

$x_1 = x_1^{\uparrow}$

$x_1^{\uparrow} = 3x_0$

$res_1 \neq 6x_0$

true

$x_1^{\uparrow} = 3x_0$

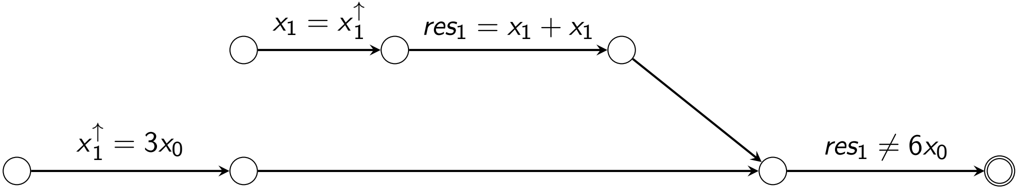
$res_1 = 6x_0$

false

Problem: Interpolants use differently scoped variables

Interpolation for Recursive Traces

```
int main(x) { assert double(3*x) == 6*x; }
int double(x) { return x+x; }
```

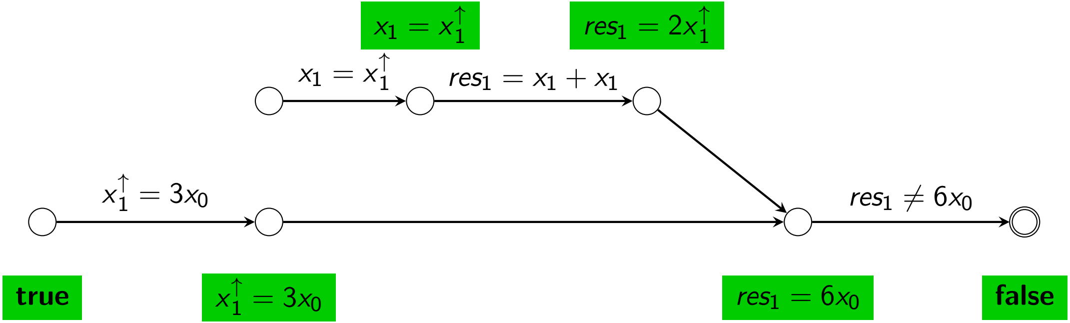


Problem: Interpolants use differently scoped variables

Solution: Tree Interpolants

Interpolation for Recursive Traces

```
int main(x) { assert double(3*x) == 6*x; }
int double(x) { return x+x; }
```

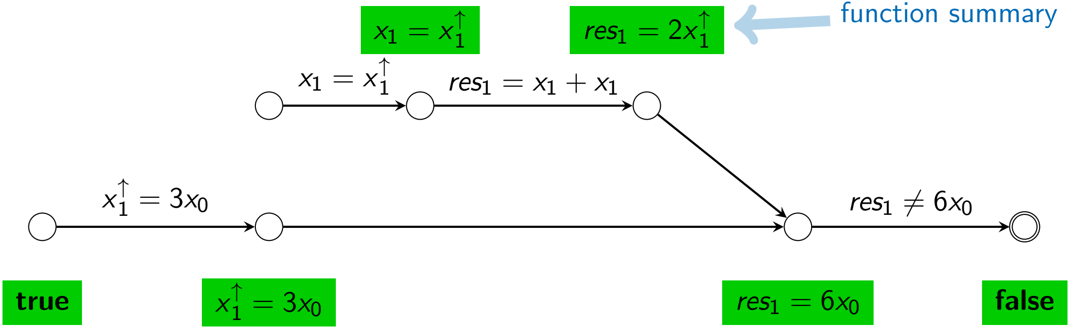


Problem: Interpolants use differently scoped variables

Solution: Tree Interpolants

Interpolation for Recursive Traces

```
int main(x) { assert double(3*x) == 6*x; }
int double(x) { return x+x; }
```



$x_1 = x_1^{\uparrow}$

$res_1 = 2x_1^{\uparrow}$

function summary

$x_1 = x_1^{\uparrow}$

$res_1 = x_1 + x_1$

$x_1^{\uparrow} = 3x_0$

$res_1 \neq 6x_0$

true

$x_1^{\uparrow} = 3x_0$

$res_1 = 6x_0$

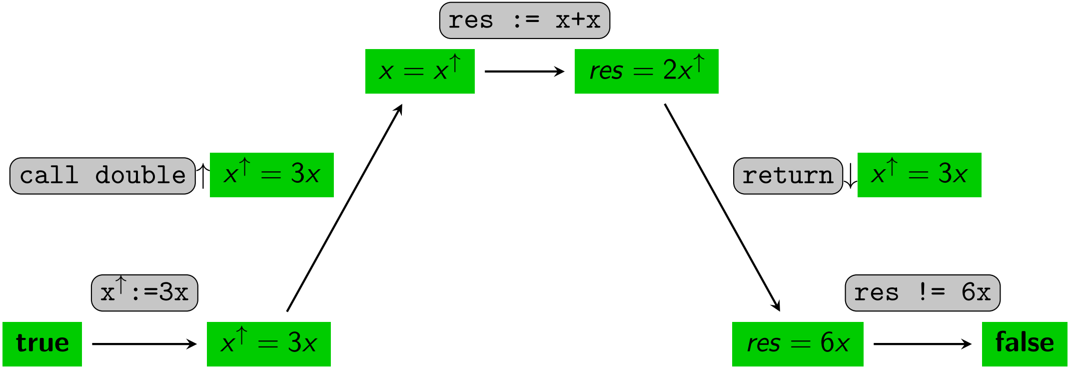
false

Problem: Interpolants use differently scoped variables

Solution: Tree Interpolants

Interpolation for Recursive Traces

```
int main(x) { assert double(3*x) == 6*x; }
int double(x) { return x+x; }
```



`res := x+x`

$x = x^{\uparrow}$ $\longrightarrow$ $res = 2x^{\uparrow}$

`call double`$\uparrow$ $x^{\uparrow} = 3x$

`return`$\downarrow$ $x^{\uparrow} = 3x$

`x`$^{\uparrow}$`:=3x`

`res != 6x`

true $\longrightarrow$ $x^{\uparrow} = 3x$

$res = 6x$ $\longrightarrow$ **false**

Problem: Interpolants use differently scoped variables

Solution: Tree Interpolants

Interpolant Automata

Given: Sequence of interpolants $\mathcal{I} = I_0, I_1, \ldots, I_n$

Definition (Interpolant Automaton $\mathcal{A}_\mathcal{I}$)

$$\mathcal{A}_\mathcal{I} = \langle Q_\mathcal{I}, \delta_\mathcal{I}, Q^{\text{init}}_\mathcal{I}, Q^{\text{fin}}_\mathcal{I} \rangle \qquad\qquad Q_\mathcal{I} = \mathcal{I}$$

$$\begin{aligned}
(I_i, st, I_j) \in \delta_\mathcal{I} \quad & \text{iff} \quad \{I_i\}\ st\ \{I_j\} \text{ holds} \\
(I_i, \boxed{\texttt{call f}}\uparrow I_i, I_j) \quad & \text{iff} \quad x^{\uparrow} = x \Rightarrow I_j \\
(I_i, \boxed{\texttt{return}}\downarrow I_k, I_j) \quad & \text{iff} \quad I_i \wedge I_k \Rightarrow I_j \\
q_0 := \mathbf{true} \in Q_\mathcal{I} & \\
Q^{\text{fin}} := \{\mathbf{false}\} \subseteq Q_\mathcal{I} &
\end{aligned}$$

Interpolant Automata

Given: Sequence of interpolants $\mathcal{I} = I_0, I_1, \ldots, I_n$

Definition (Interpolant Automaton $\mathcal{A}_\mathcal{I}$)

$$\mathcal{A}_\mathcal{I} = \langle Q_\mathcal{I}, \delta_\mathcal{I}, Q^{\text{init}}_\mathcal{I}, Q^{\text{fin}}_\mathcal{I} \rangle \qquad Q_\mathcal{I} = \mathcal{I}$$

$$\begin{aligned}
(I_i, st, I_j) \in \delta_\mathcal{I} \quad & \text{iff} \quad \{I_i\}\ st\ \{I_j\} \text{ holds} \\
(I_i, \boxed{\texttt{call f}}\uparrow I_i, I_j) \quad & \text{iff} \quad x^\uparrow = x \Rightarrow I_j \\
(I_i, \boxed{\texttt{return}}\downarrow I_k, I_j) \quad & \text{iff} \quad I_i \wedge I_k \Rightarrow I_j \\
q_0 := \textbf{true} \in Q_\mathcal{I} & \\
Q^{\text{fin}} := \{\textbf{false}\} \subseteq Q_\mathcal{I} &
\end{aligned}$$

Theorem

An interpolant automaton $\mathcal{A}_\mathcal{I}$ recognizes a subset of infeasible traces.

$$\mathcal{L}(\mathcal{A}_\mathcal{I}) \subseteq \mathit{Infeasible}$$

Computing Tree Interpolants with SMTInterpol

$x^{\uparrow}$:=3x	call double	res := x+x	return	res != 6x

```
(set-option :produce-interpolants true)
(set-logic QF_LIA)
(declare-const x0 Int)
(declare-const x1 Int)
(declare-const x^1 Int)
(declare-const res1 Int)

(assert (! (= x^1 (* 3 x0)) :named st1))
(assert (! (= x1 x^1) :named st2))
(assert (! (= res1 (+ x1 x1)) :named st3))
(assert (! true :named st4))
(assert (! (not (= res1 (* 6 x0))) :named st5))

(check-sat)
(set-option :print-terms-cse false)
(get-interpolants st1 (st2 st3) st4 st5)
```

Running Example: McCarthy 91

$$\pi = \begin{array}{l}
\boxed{\texttt{x}^{\uparrow}\texttt{<=101}} \\
\quad \boxed{\texttt{call f}}\boxed{\texttt{x<=100}}\boxed{\texttt{x}^{\uparrow}\texttt{:=x+11}} \\
\qquad \boxed{\texttt{call f}}\boxed{\texttt{x>100}}\boxed{\texttt{res:=x-10}}\boxed{\texttt{return}} \\
\quad \boxed{\texttt{x}^{\uparrow}\texttt{:=res}} \\
\qquad \boxed{\texttt{call f}}\boxed{\texttt{x>100}}\boxed{\texttt{res:=x-10}}\boxed{\texttt{return}} \\
\quad \boxed{\texttt{return}} \\
\boxed{\texttt{tmp!=91}}
\end{array}$$

Running Example: McCarthy 91

$\pi =$

```
x↑<=101
  call f  x<=100  x↑:=x+11
    call f  x>100  res:=x-10  return
  x↑:=res
    call f  x>100  res:=x-10  return
  return
tmp!=91
```

$$
SSA(\pi) = \begin{array}{l}
x_0^{\uparrow} \leq 101 \wedge \\
\quad x_1 = x_0^{\uparrow} \wedge x_1 \leq 100 \wedge x_1^{\uparrow} = x_1 + 11 \wedge \\
\qquad x_2 = x_1^{\uparrow} \wedge x_2 > 100 \wedge res_1 = x_2 - 10 \wedge \textbf{true} \wedge \\
\quad x_2^{\uparrow} = res_1 \wedge \\
\qquad x_3 = x_2^{\uparrow} \wedge x_3 > 100 \wedge res_2 = x_3 - 10 \wedge \textbf{true} \wedge \\
\quad \textbf{true} \wedge \\
res_2 \neq 91
\end{array}
$$

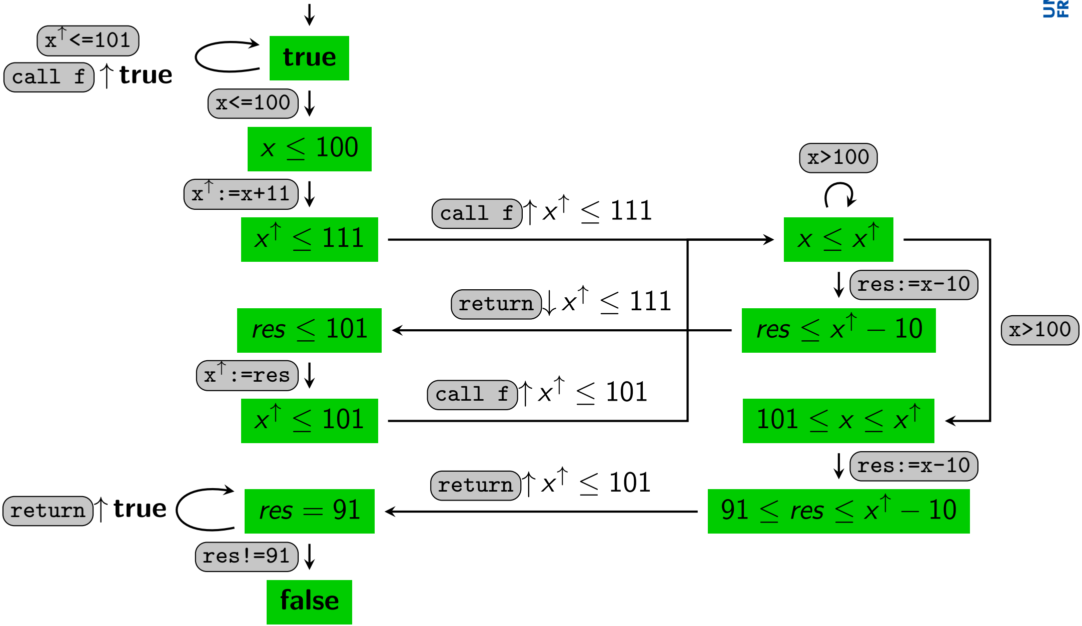
Computing Interpolants

```
(set-option :produce-interpolants true)
(set-logic QF_LIA)
(declare-const x0 Int)
(declare-const x1 Int)
(declare-const x2 Int)
(declare-const x3 Int)
(declare-const x^0 Int)
...

(assert (! (...) :named st1))
...

(check-sat)
(get-interpolants st1 (st2 ...) st12)
```

Computing Interpolants

```
(set-option :produce-interpolants true)
(set-logic QF_LIA)
(declare-const x0 Int)
(declare-const x1 Int)
(declare-const x2 Int)
(declare-const x3 Int)
(declare-const x^0 Int)
...

(assert (! (...) :named st1))
...

(check-sat)
(get-interpolants st1 (st2 ...) st12)

;;(true true (<= x1 100) (<= x^1 111) (<= x2 x^1) (<= x2 x^1)
;;(<= res1 (- x^1 10)) (<= x^2 101) (<= x3 x^2) (<= 101 x3 x^2))
;;(or (<= 91 res2 (- x^2 10)) (= res2 91)) (= res2 91))
```

Interpolant Automaton



$x^{\uparrow}$<=101

call f $\uparrow$ **true**

true

x<=100 $\downarrow$

$x \leq 100$

$x^{\uparrow}$:=x+11 $\downarrow$

$x^{\uparrow} \leq 111$

call f $\uparrow x^{\uparrow} \leq 111$

x>100

$x \leq x^{\uparrow}$

$\downarrow$ res:=x-10

return $\downarrow x^{\uparrow} \leq 111$

$res \leq 101$

$res \leq x^{\uparrow} - 10$

x>100

$x^{\uparrow}$:=res $\downarrow$

$x^{\uparrow} \leq 101$

call f $\uparrow x^{\uparrow} \leq 101$

$101 \leq x \leq x^{\uparrow}$

$\downarrow$ res:=x-10

return $\uparrow x^{\uparrow} \leq 101$

return $\uparrow$ **true**

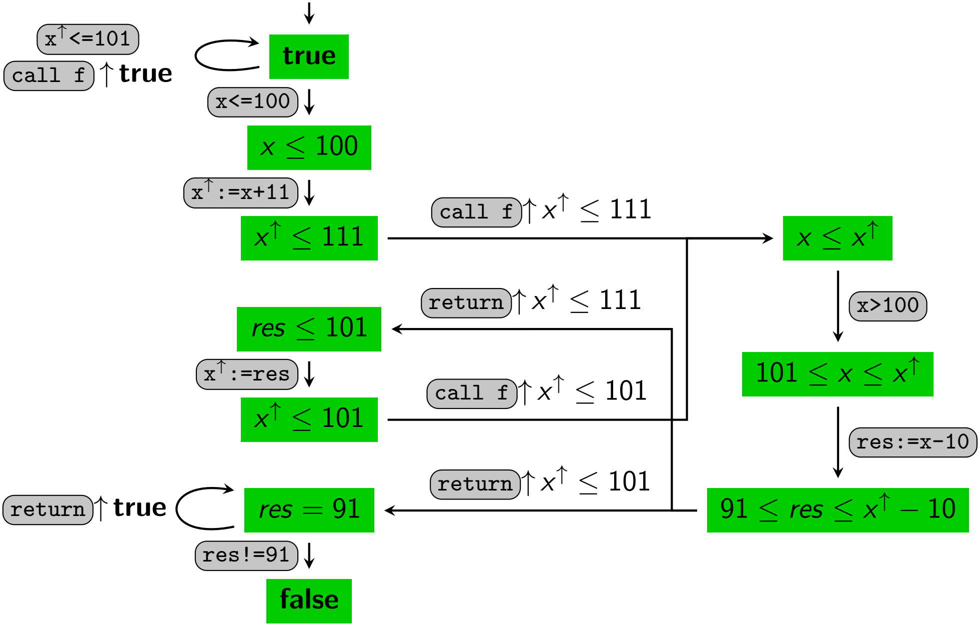
$res = 91$

$91 \leq res \leq x^{\uparrow} - 10$

res!=91 $\downarrow$

false

Interpolant Automaton



$\mathtt{x}^{\uparrow}\mathtt{<=101}$

$\mathtt{call\ f}\uparrow$ **true**

true

`x<=100`

$x \leq 100$

$\mathtt{x}^{\uparrow}\mathtt{:=x+11}$

$x^{\uparrow} \leq 111$

$\mathtt{call\ f}\uparrow x^{\uparrow} \leq 111$

$x \leq x^{\uparrow}$

`x>100`

$101 \leq x \leq x^{\uparrow}$

$\mathtt{return}\uparrow x^{\uparrow} \leq 111$

$res \leq 101$

$\mathtt{x}^{\uparrow}\mathtt{:=res}$

$x^{\uparrow} \leq 101$

$\mathtt{call\ f}\uparrow x^{\uparrow} \leq 101$

`res:=x-10`

$\mathtt{return}\uparrow x^{\uparrow} \leq 101$

$\mathtt{return}\uparrow$ **true**

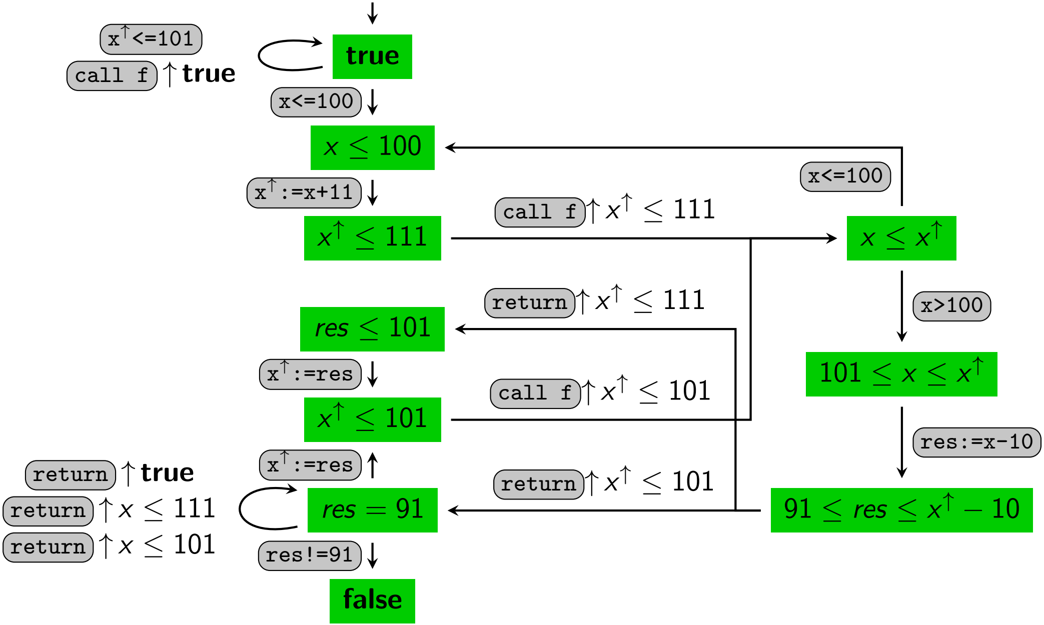
$res = 91$

$91 \leq res \leq x^{\uparrow} - 10$

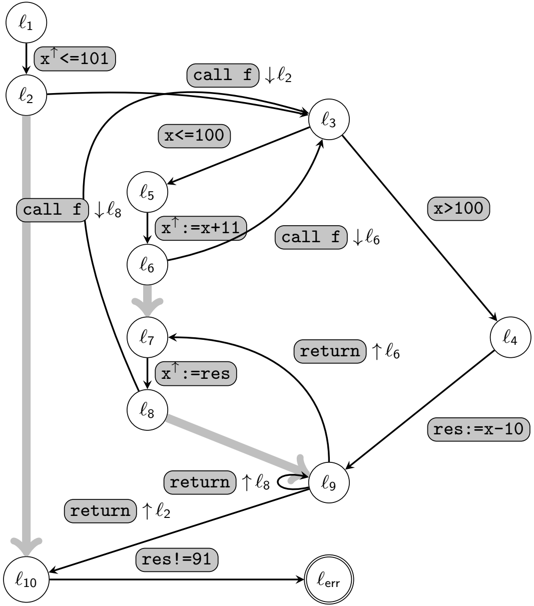
`res!=91`

false

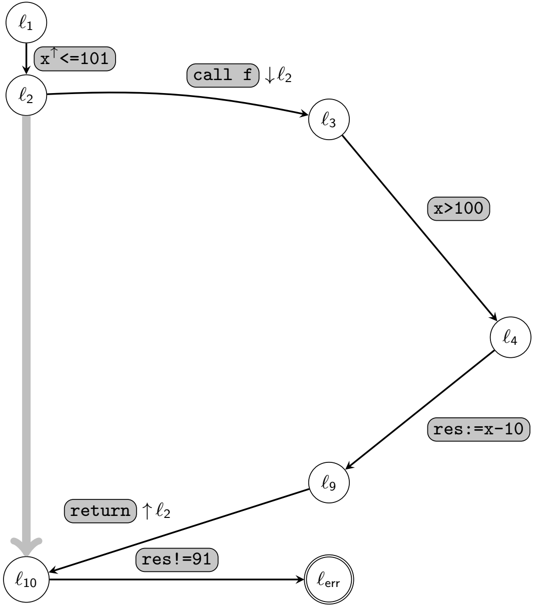
Interpolant Automaton



Subtracting from Program Automaton



Subtracting from Program Automaton



Termination

Example: Termination Problem

```
void main() {
  while (x > 0 && y > 0) {
    if (*) {
      x := x - 1;
    } else {
      x := *;
      y := y - 1;
    }
  }
}
```

Does this program terminate?

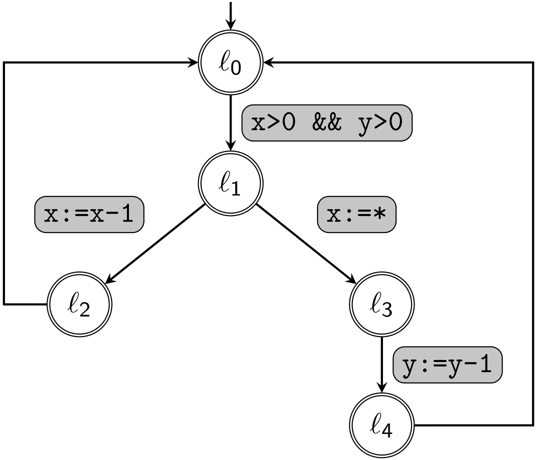
Example: Termination Problem

```
void main() {
  while (x > 0 && y > 0) {
    if (*) {
      x := x - 1;
    } else {
      x := *;
      y := y - 1;
    }
  }
}
```

Does this program terminate?
How can we prove it?

Program Büchi Automata

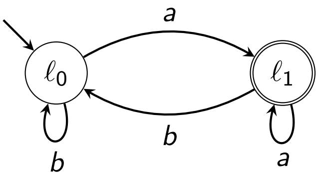
```
void main() {
  while (x > 0 && y > 0) {
    if (*) {
      x := x - 1;
    } else {
      x := *;
      y := y - 1;
    }
  }
}
```



A Büchi automaton accepts infinite words that infinitely often visit accepting states.

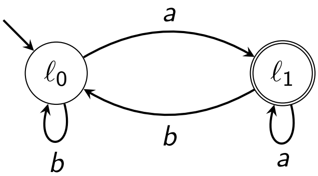
Büchi Automata

$\mathcal{A}_1$:



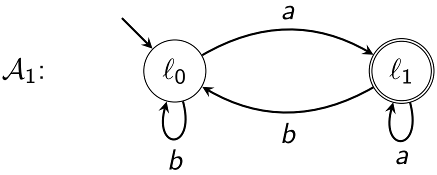
Büchi Automata

$\mathcal{A}_1$:

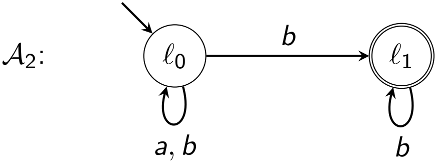


$$\mathcal{L}(\mathcal{A}_1) = \{(a+b)^\omega \mid \text{infinitely many } a\}$$

Büchi Automata

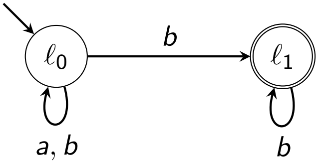


$$\mathcal{L}(\mathcal{A}_1) = \{(a+b)^\omega \mid \text{infinitely many } a\}$$



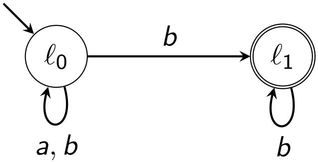
$$\mathcal{L}(\mathcal{A}_2) = \{(a+b)^\omega \mid \text{finitely many } a\}$$

Traces and Lassos



How to represent infinite words?

Traces and Lassos

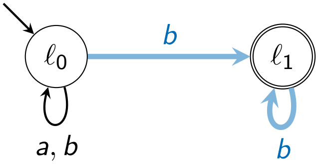


How to represent infinite words?

Fact

Every non-empty Büchi-Automaton accepts an ultimately periodic word.

Traces and Lassos



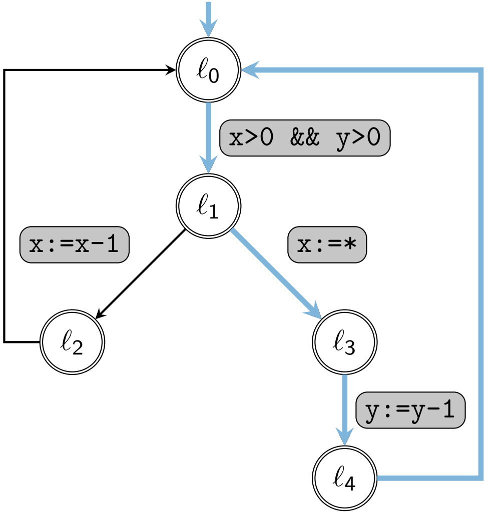
How to represent infinite words?

Fact

Every non-empty Büchi-Automaton accepts an ultimately periodic word.

$$b(b)^\omega \in \mathcal{L}(\mathcal{A}_2)$$

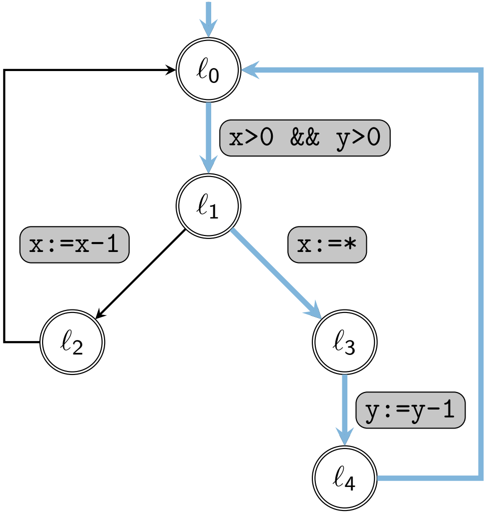
Program Büchi Automata



Error Trace:

`x>0 && y>0` (`x:=*` `y:=y-1` `x>0 && y>0`)$^\omega$

Program Büchi Automata

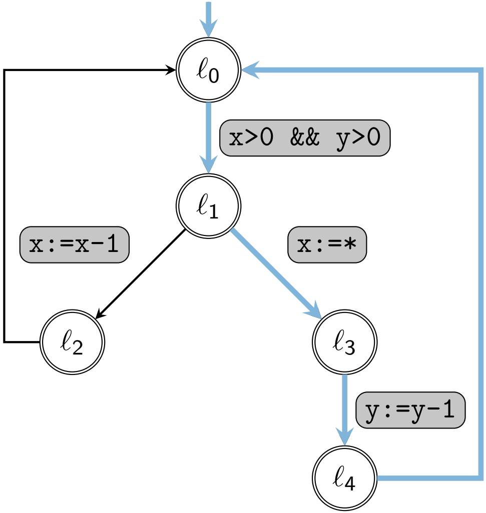


Error Trace:

$$\boxed{\texttt{x>0 \&\& y>0}}\left(\boxed{\texttt{x:=*}}\boxed{\texttt{y:=y-1}}\boxed{\texttt{x>0 \&\& y>0}}\right)^{\omega}$$

Infeasible: Ranking function y.

Program Büchi Automata

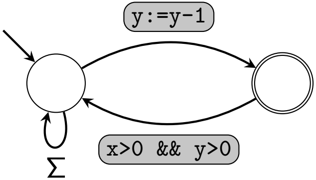


Error Trace:

`x>0 && y>0` $($ `x:=*` `y:=y-1` `x>0 && y>0` $)^\omega$

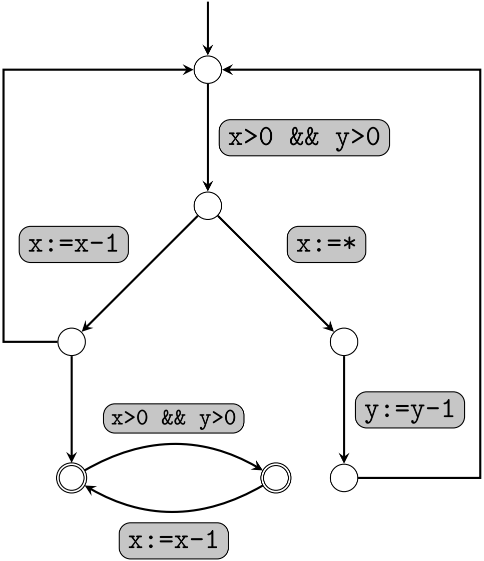
Infeasible: Ranking function y.

Generalization:



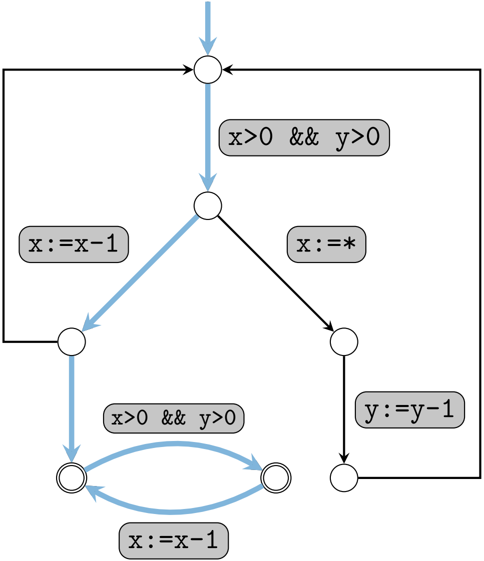
Remaining Program Büchi Automata

∃ algorithms for intersection/complementation of Büchi Automata!



Remaining Program Büchi Automata

∃ algorithms for intersection/complementation of Büchi Automata!

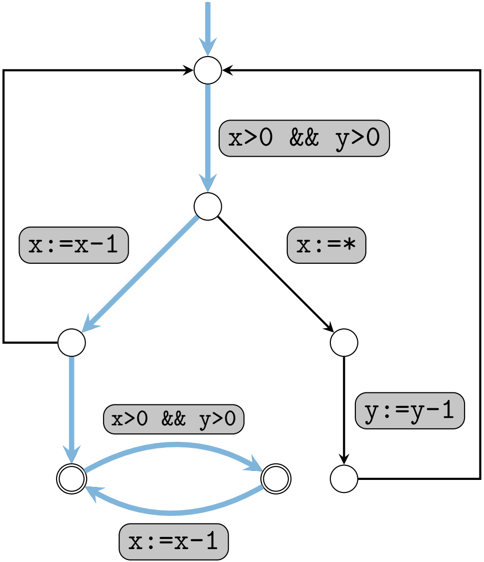


Error Trace:

`x>0 && y>0` `x:=x-1` $\big($`x>0 && y>0` `x:=x-1`$\big)^\omega$

Remaining Program Büchi Automata

∃ algorithms for intersection/complementation of Büchi Automata!



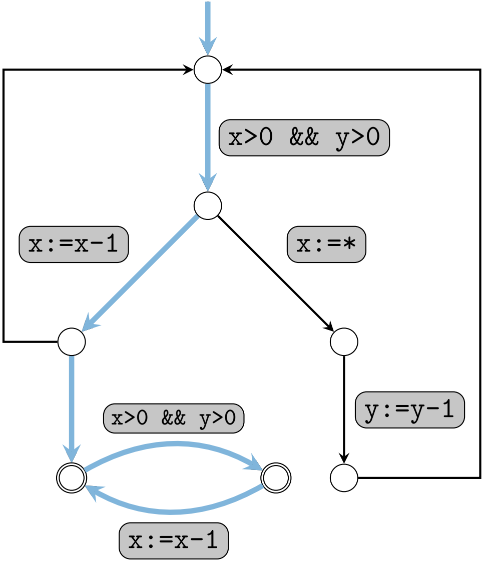
Error Trace:

`x>0 && y>0` `x:=x-1` (`x>0 && y>0` `x:=x-1`)$^\omega$

Infeasible: Ranking function x.

Remaining Program Büchi Automata

∃ algorithms for intersection/complementation of Büchi Automata!

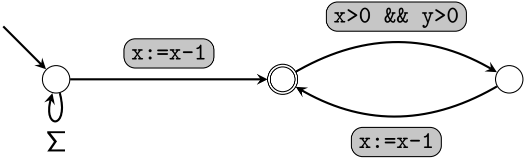


Error Trace:

`x>0 && y>0` `x:=x-1` (`x>0 && y>0` `x:=x-1`)$^{\omega}$

Infeasible: Ranking function x.

Generalization:



How to find Ranking Functions?

We have a tool ULTIMATE RANKFINDER.

- General pattern $\mathit{rank} := c_1 x_1 + \cdots + c_n x_n$
  where $x_1, \ldots, x_n$ are the variables, $c_1 \ldots c_n \in \mathbb{Z}$.
- For a lasso $\mathit{stem}(\mathit{loop})^\omega$, encode an SMT problem:

$$\forall x_1 \ldots x_n, x_1', \ldots, x_n'. \mathit{loop} \Rightarrow \mathit{rank} \geq 0 \wedge \mathit{rank}' \leq \mathit{rank} - 1$$

  Quantifiers can be eliminated with Farkas' Lemma.
- Ask SMT solver for solution for $c_1, \ldots, c_n$ (non-linear arithmetic required).