Trace Abstraction

Matthias Heizmann   Jochen Hoenicke   Andreas Podelski

University of Freiburg, Germany

Interpolant-based software model checking
for recursive programs

Software model checking

Thomas Ball, Sriram K. Rajamani:

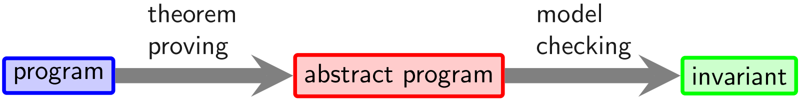
The SLAM project: debugging system software via static analysis. (POPL 2002)

Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, Grégoire Sutre

Lazy abstraction. (POPL 2002)

Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, Kenneth L. McMillan

Abstractions from proofs. (POPL 2004)



Software model checking

Thomas Ball, Sriram K. Rajamani:

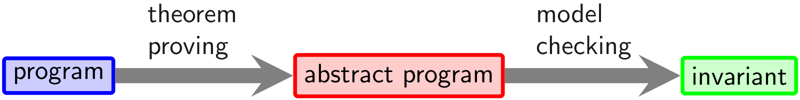
The SLAM project: debugging system software via static analysis. (POPL 2002)

Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, Grégoire Sutre

Lazy abstraction. (POPL 2002)

Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, Kenneth L. McMillan

Abstractions from proofs. (POPL 2004)



Bottleneck: Construction of abstract program

Recent approaches:
Avoid classical construction of abstract program

Franjo Ivancic, Ilya Shlyakhter, Aarti Gupta, Malay K. Ganai

Model checking C programs using F-SOFT (ICCD 2005)

Kenneth L. McMillan

Lazy abstraction with interpolants (CAV 2006)

Nels Beckman, Aditya V. Nori, Sriram K. Rajamani, Robert J. Simmons

Proofs from tests (ISSTA 2008)

Bhargav S. Gulavani, Supratik Chakraborty, Aditya V. Nori, Sriram K. Rajamani

Automatically refining abstract interpretations (TACAS 2008)

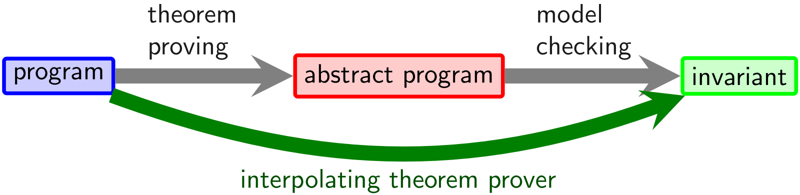
Klaus Dräger, Andrey Kupriyanov, Bernd Finkbeiner, Heike Wehrheim

SLAB: A Certifying Model Checker for Infinite-State Concurrent Systems. (TACAS 2010)

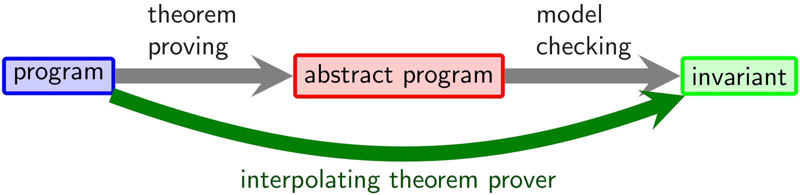
William R. Harris, Sriram Sankaranarayanan, Franjo Ivancic, Aarti Gupta

Program analysis via satisfiability modulo path programs (POPL 2010)

One idea:
Use interpolants to avoid construction of the abstract program



One idea:
Use interpolants to avoid construction of the abstract program



Ranjit Jhala, Kenneth L. McMillan

A practical and complete approach to predicate refinement (TACAS 2006)

Kenneth L. McMillan

Lazy abstraction with interpolants (CAV 2006)

Quantified invariant generation using an interpolating saturation prover (TACAS 2008)

Open: Interpolants in interprocedural analysis

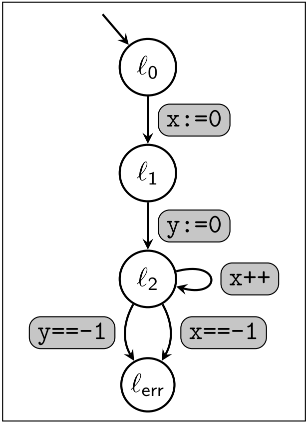
Outline

- Formal setting / Our point of view:
  A program is a language over the alphabet of statements.
- Excursion: interpolants
- Trace Abstraction with interpolants
- Trace Abstraction for recursive programs

Example – Our Model of a Verification Problem

ℓ_0: x:=0
ℓ_1: y:=0
ℓ_2: while(nondet) {x++}
assert x!= -1
assert y!= -1

Example program $\mathcal{P}$



Control flow graph of $\mathcal{P}$

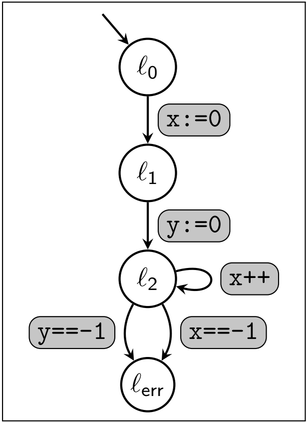
Statements

Statement

Letter of our alphabet. No further meaning.

In our example:

$\Sigma = \{$`x:=0`, `y:=0`, `x++`, `x==-1`, `y==-1`$\}$



Control flow graph of $\mathcal{P}$

Statements

Statement

Letter of our alphabet. No further meaning.

In our example:

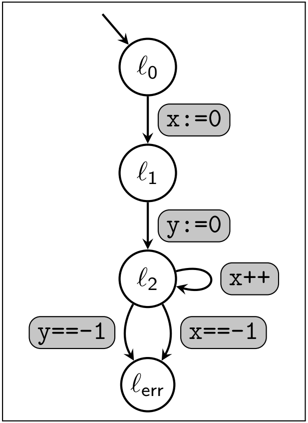
$\Sigma = \{$ `x:=0`, `y:=0`, `x++`, `x==-1`, `y==-1` $\}$

Trace

Word over the alphabet of statements.

Example:

$\pi =$ `y==-1`.`x++`.`x++`.`x:=0`.`x==-1`



Control flow graph of $\mathcal{P}$

Error Traces

Control Automaton $\mathcal{A}_\mathcal{P}$

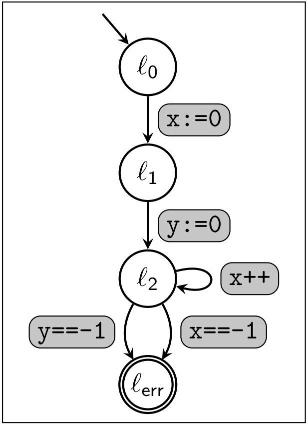
Automaton over the set of statements.
Encodes a verification problem.

$$\mathcal{A}_\mathcal{P} = \langle LOC, \delta, \{\ell_{\mathsf{init}}\}, \{\ell_{\mathsf{err}}\} \rangle$$

Error Trace of $\mathcal{P}$

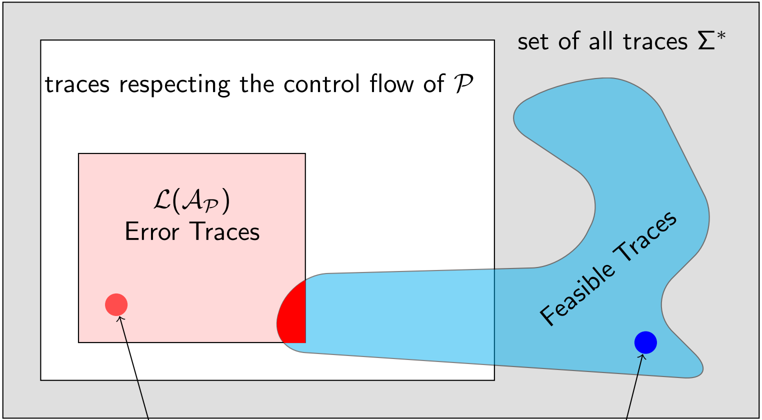
Trace accepted by $\mathcal{A}_\mathcal{P}$

In our example
$\pi =$ `x:=0`.`y:=0`.`x++`.`x==-1`
is an error trace.



Control automaton $\mathcal{A}_\mathcal{P}$

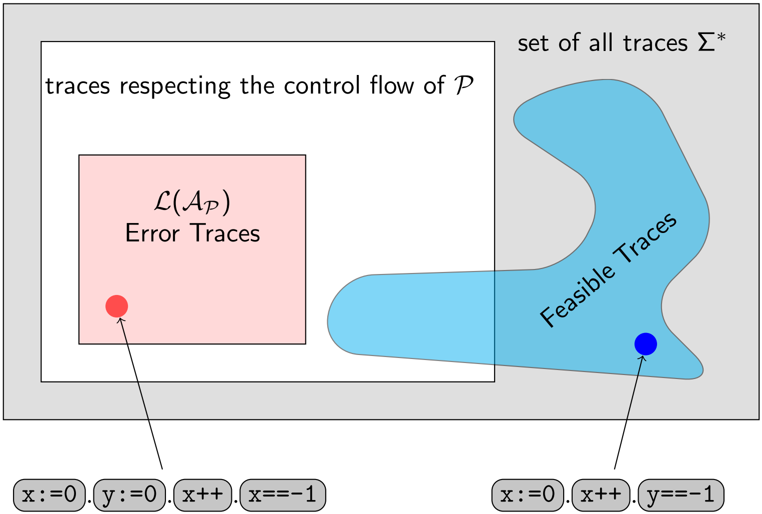
Set Theoretic View of Trace Abstraction



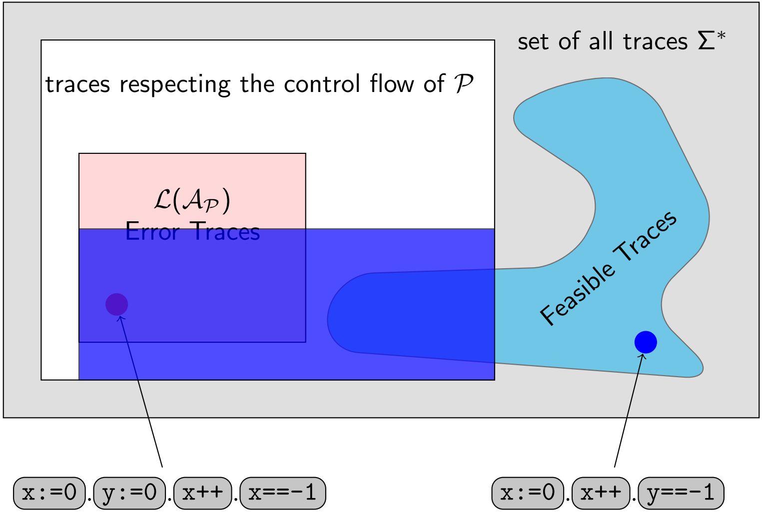
`x:=0`.`y:=0`.`x++`.`x==-1`

`x:=0`.`x++`.`y==-1`

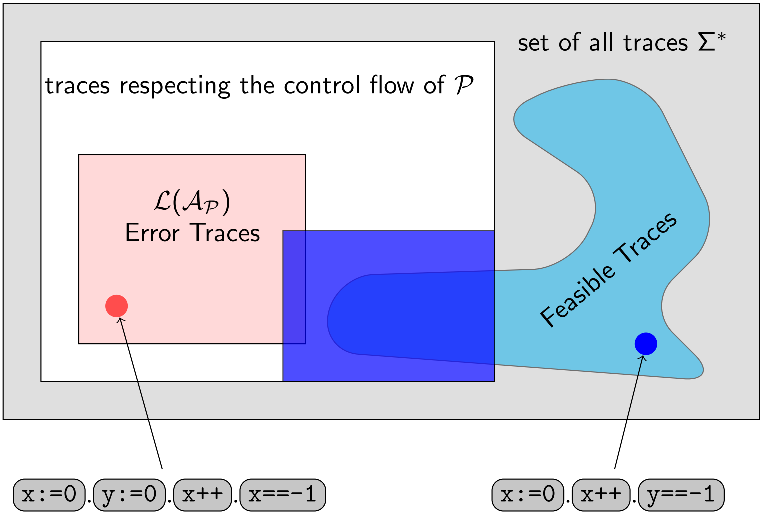
Set Theoretic View of Trace Abstraction



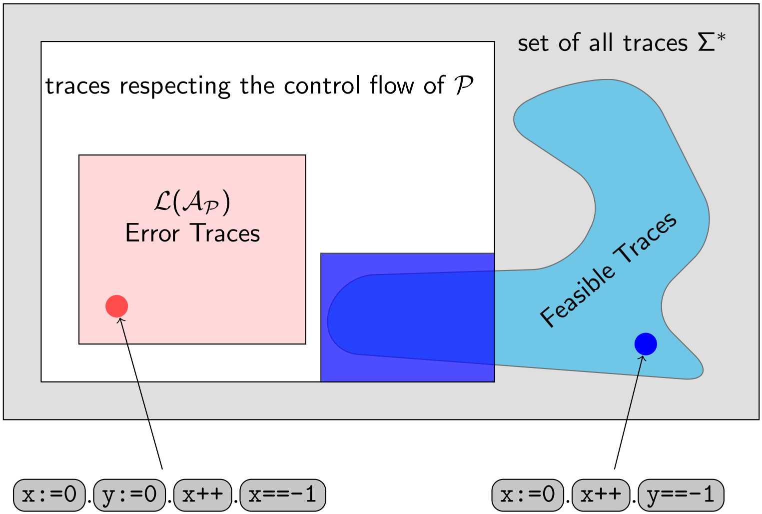
Set Theoretic View of Trace Abstraction



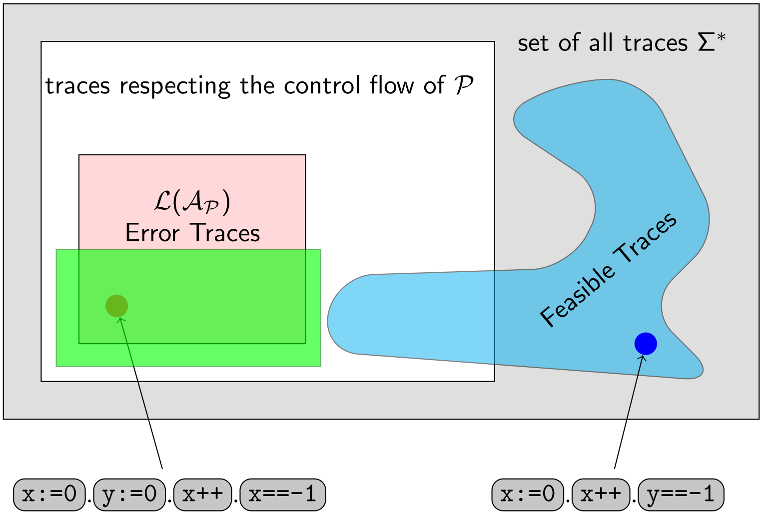
Set Theoretic View of Trace Abstraction



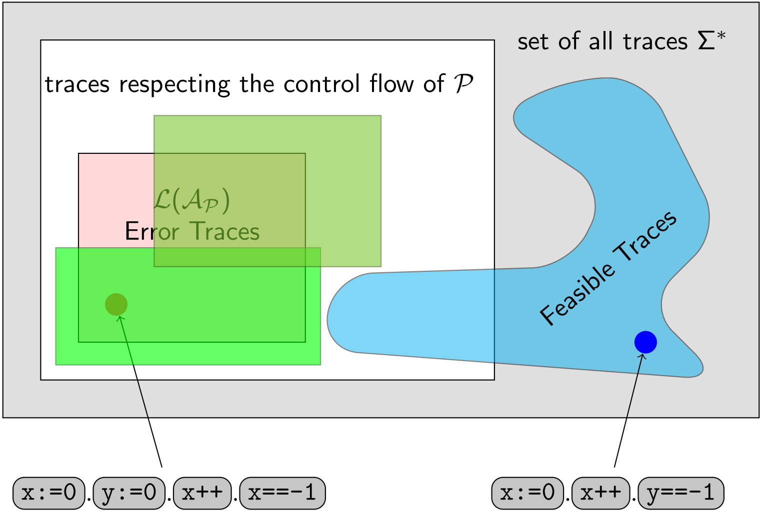
Set Theoretic View of Trace Abstraction



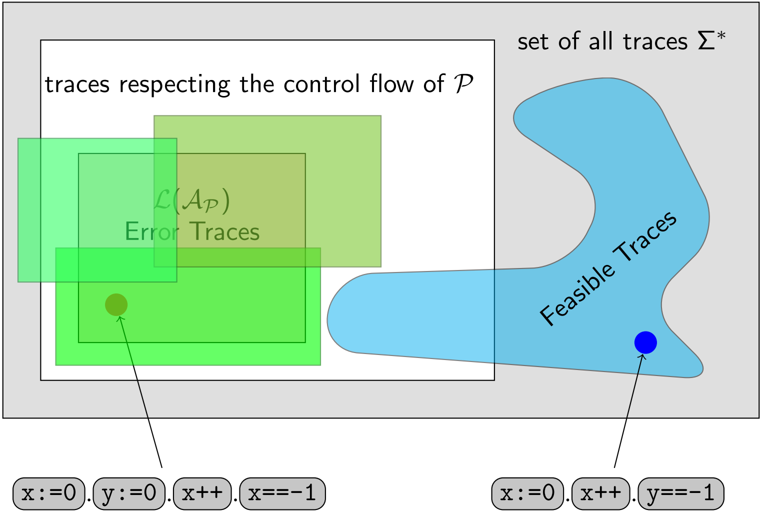
Set Theoretic View of Trace Abstraction



Set Theoretic View of Trace Abstraction



Set Theoretic View of Trace Abstraction



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)

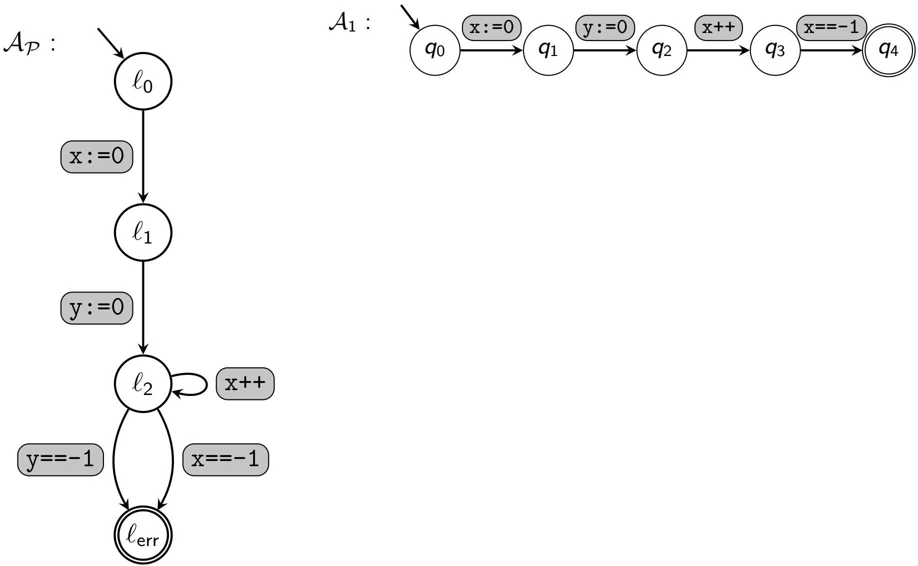
$$\mathcal{L}(\mathcal{A}_\mathcal{P} \cap \overline{\mathcal{A}_1} \cap \ldots \cap \overline{\mathcal{A}_n}) = \emptyset \qquad \Rightarrow \qquad \mathcal{P} \textit{ is correct}$$

Theorem (Completeness)

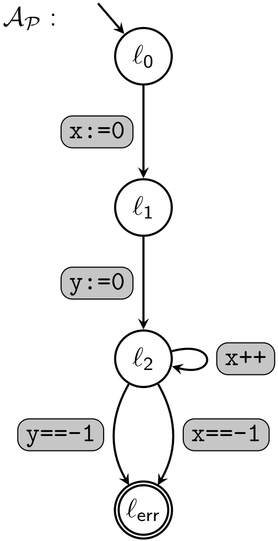
If $\mathcal{P}$ *is correct, there is a trace abstraction* $(\mathcal{A}_1, \ldots, \mathcal{A}_n)$ *such that*

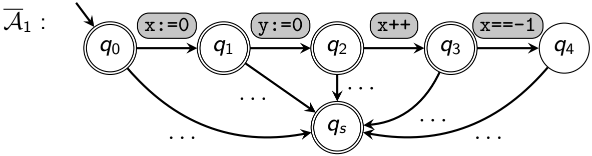
$$\mathcal{L}(\mathcal{A}_\mathcal{P} \cap \overline{\mathcal{A}_1} \cap \ldots \cap \overline{\mathcal{A}_n}) = \emptyset$$

Example – Exclude an Infeasible Trace

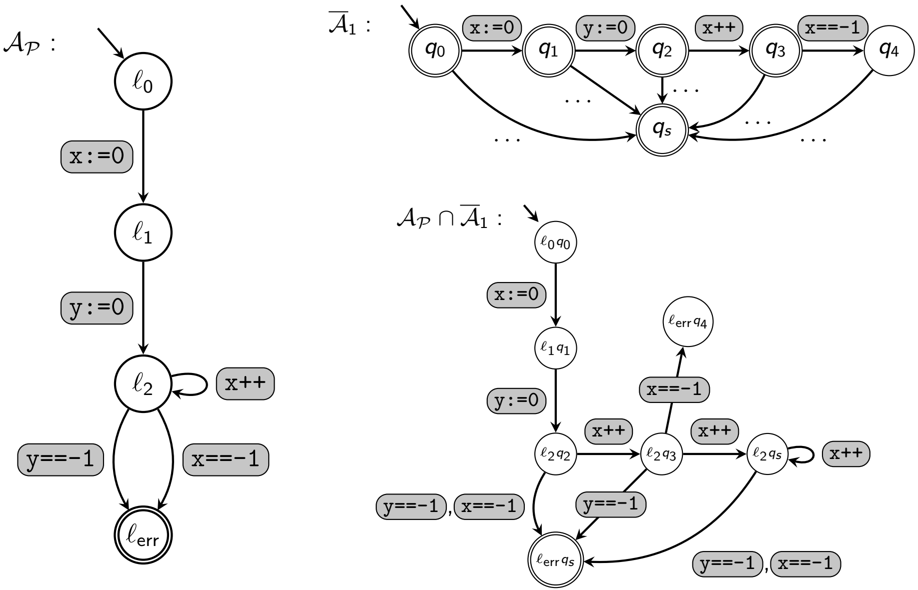


Example – Exclude an Infeasible Trace





Example – Exclude an Infeasible Trace



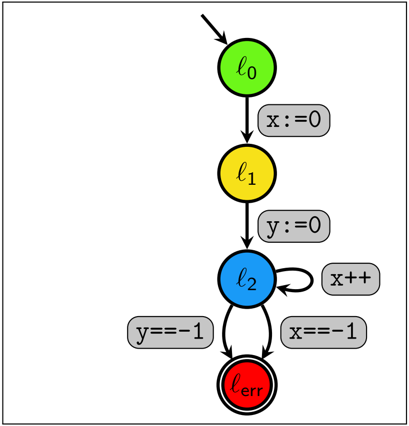
Control flow as finite automaton

ℓ_0: `x:=0`

ℓ_1: `y:=0`

ℓ_2: `while(nondet) {x++}`
`assert x!= -1`
`assert y!= -1`

Example program $\mathcal{P}$

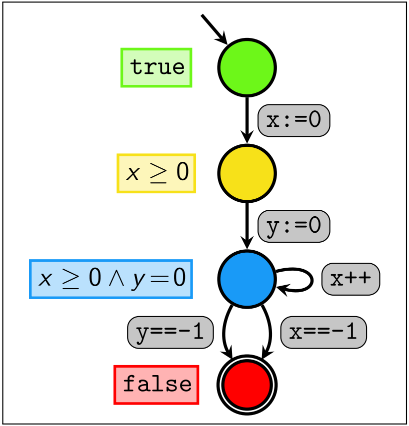


Control flow graph of $\mathcal{P}$

Floyd-Hoare proof as finite automaton

```
      {⊤}
ℓ0:   x:=0
      {x ≥ 0}
ℓ1:   y:=0
      {x ≥ 0 ∧ y = 0}
ℓ2:   while(nondet) {x++}
      assert x!= -1
      assert y!= -1
```

Example program $\mathcal{P}$



Control flow graph of $\mathcal{P}$

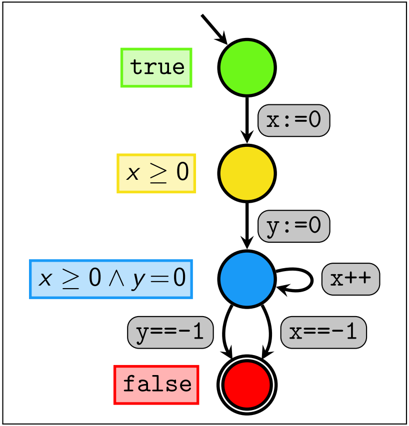
Floyd-Hoare proof as finite automaton

ℓ_0: `x:=0`

ℓ_1: `y:=0`

ℓ_2: `while(nondet) {x++}`
`assert x!= -1`
`assert y!= -1`

Example program $\mathcal{P}$



Control flow graph of $\mathcal{P}$

Observation: Every transition is related to a Hoare triple!

e.g. (, `y:=0` ,) $\in \delta$ $\quad$ $post(x \geq 0, \texttt{y:=0}) \subseteq x \geq 0 \wedge y = 0$

Interpolant Automata

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

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

$$\mathcal{A}_\mathcal{I} = \langle Q_\mathcal{I}, \delta_\mathcal{I}, Q_\mathcal{I}^{\text{init}}, Q_\mathcal{I}^{\text{fin}} \rangle \qquad Q_\mathcal{I} = \{q_0, \ldots, q_n\}$$

$$\begin{array}{rcl} (q_i, st, q_j) \in \delta_\mathcal{I} & \text{implies} & post(st, I_i) \subseteq I_j \\ q_i \in Q^{\text{init}} & \text{implies} & I_i = true \\ q_i \in Q^{\text{fin}} & \text{implies} & I_i = false \end{array}$$

Interpolant Automata

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

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

$$\mathcal{A}_\mathcal{I} = \langle Q_\mathcal{I}, \delta_\mathcal{I}, Q_\mathcal{I}^{\mathsf{init}}, Q_\mathcal{I}^{\mathsf{fin}} \rangle \qquad Q_\mathcal{I} = \{q_0, \ldots, q_n\}$$

$$\begin{aligned} (q_i, st, q_j) \in \delta_\mathcal{I} \quad &\text{implies} \quad post(st, I_i) \subseteq I_j \\ q_i \in Q^{\mathsf{init}} \quad &\text{implies} \quad I_i = true \\ q_i \in Q^{\mathsf{fin}} \quad &\text{implies} \quad I_i = false \end{aligned}$$

Theorem

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

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

Outline

- Formal setting / Our point of view:
  A program is a language over the alphabet of statements.
- Excursion: interpolants
- Trace Abstraction with interpolants
- Trace Abstraction for recursive programs

Craig interpolants

Craig interpolant - logical formulas

Given: Unsatisfiable conjuction A $\wedge$ B

Interpolant is a formula I such that:

- A implies I and I $\wedge$ B unsatisfiable
- I contains only common symbols of A and B

William Craig

Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory (

Journal of Sybolic Logic (1957))

Craig interpolants

Craig interpolant - logical formulas

Given: Unsatisfiable conjuction `A` $\wedge$ `B`

Interpolant is a formula `I` such that:

- `A` implies `I` and `I` $\wedge$ `B` unsatisfiable
- `I` contains only common symbols of A and B

Example (propositional logic)

unsatisfiable conjuction: $p \wedge q \quad \wedge \quad \neg p \wedge r$

possible Craig interpolant: p

Example (SMT)

unsatisfiable conjuction: $f(x_1) = y \wedge x_1 = x_2 \quad \wedge \quad x_2 = x_3 \wedge f(x_3) \neq y$

possible Craig interpolant: $y = f(x_2)$

Interpolants

Interpolant - execution traces

Given: Infeasible trace $st_1 \ldots st_i\, st_{i+1} \ldots st_n$

Interpolant is assertion `I` such that:

- $post($ `true` $, st_1 \ldots st_i) \subseteq$ `I` $\subseteq wp($ `false` $, st_{i+1} \ldots st_n)$
- `I` contains only program variables occuring in both, $st_1 \ldots st_i$ and $st_{i+1} \ldots st_n$

Kenneth L. McMillan

Interpolation and SAT-Based Model Checking (CAV 2003)

Interpolants

Interpolant - execution traces

Given: Infeasible trace $st_1 \ldots st_i\, st_{i+1} \ldots st_n$

Interpolant is assertion `I` such that:

- $post($ `true` $, st_1 \ldots st_i) \subseteq$ `I` $\subseteq wp($ `false` $, st_{i+1} \ldots st_n)$
- `I` contains only program variables occuring in both, $st_1 \ldots st_i$ and $st_{i+1} \ldots st_n$

Example

infeasible trace: `x:=0` `y:=0` `x++` `x==-1`

possible interpolant: $x \geq 0$

Inductive interpolants

Inductive sequence of interpolants

Given: Infeasible trace $st_1 \dots st_n$

There exists sequence of assertions $I_0 \dots I_n$ such that:

- $post(I_i, st_i) \subseteq I_{i+1}$
- $I_0 = \texttt{true}$ and $I_n = \texttt{false}$
- I_i contains only variables occuring in both, $st_1 \dots st_i$ and $st_{i+1} \dots st_n$

Ranjit Jhala, Kenneth L. McMillan

A Practical and Complete Approach to Predicate Refinement (TACAS 2006)

Inductive interpolants

Inductive sequence of interpolants

Given: Infeasible trace $st_1 \ldots st_n$

There exists sequence of assertions $I_0 \ldots I_n$ such that:

- $post(I_i, st_i) \subseteq I_{i+1}$
- $I_0 =$ `true` and $I_n =$ `false`
- I_i contains only variables occuring in both, $st_1 \ldots st_i$ and $st_{i+1} \ldots st_n$

Example

`true` — `x:=0` — $x \geq 0$ — `y:=0` — $x \geq 0$ — `x++` — $x \geq 0$ — `x==-1` — `false`

Computation of interpolants - Example

infeasible trace

`x:=0` `y:=0` `x++` `x==-1`

Computation of interpolants - Example

infeasible trace

`x:=0` `y:=0` `x++` `x==-1`

single static assignment form

$$x_0 = 0 \quad \wedge \quad y_1 = 0 \quad \wedge \quad x_2 = x_0 + 1 \quad \wedge \quad x_2 = -1$$

Computation of interpolants - Example

infeasible trace

`x:=0` `y:=0` `x++` `x==-1`

single static assignment form

$$x_0 = 0 \quad \wedge \quad y_1 = 0 \quad \wedge \quad x_2 = x_0 + 1 \quad \wedge \quad x_2 = -1$$

inductive sequence of Craig interpolants

`true`

Computation of interpolants - Example

infeasible trace

`x:=0` `y:=0` `x++` `x==-1`

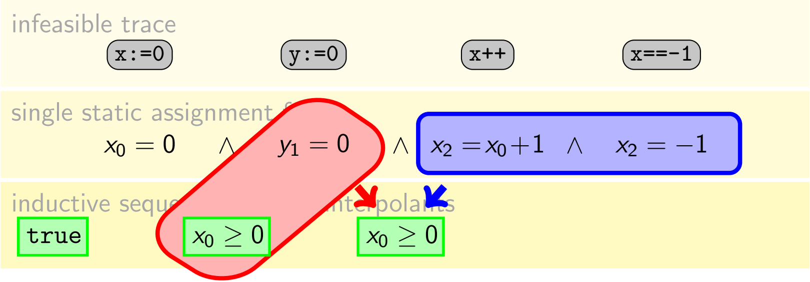
single static assignment form

$x_0 = 0 \quad \wedge \quad y_1 = 0 \quad \wedge \quad x_2 = x_0 + 1 \quad \wedge \quad x_2 = -1$

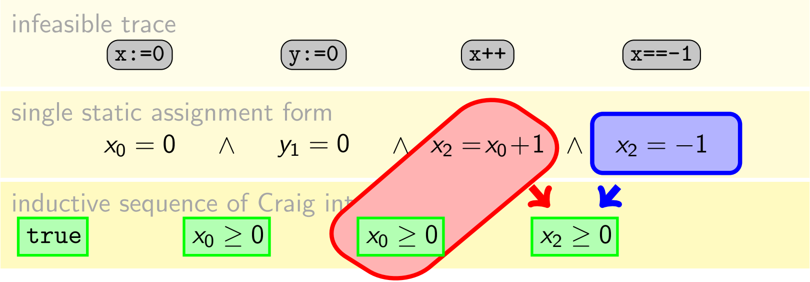
inductive sequence of Craig interpolants

`true` $x_0 \geq 0$

Computation of interpolants - Example



Computation of interpolants - Example



Computation of interpolants - Example

infeasible trace

`x:=0` `y:=0` `x++` `x==-1`

single static assignment form

$x_0 = 0 \quad \wedge \quad y_1 = 0 \quad \wedge \quad x_2 = x_0 + 1 \quad \wedge \quad x_2 = -1$

inductive sequence of Craig interpolants

`true` $x_0 \geq 0$ $x_0 \geq 0$ $x_2 \geq 0$ `false`

Computation of interpolants - Example

infeasible trace

`x:=0`	`y:=0`	`x++`	`x==-1`

single static assignment form

$x_0 = 0 \quad \wedge \quad y_1 = 0 \quad \wedge \quad x_2 = x_0 + 1 \quad \wedge \quad x_2 = -1$

inductive sequence of Craig interpolants

`true`	$x_0 \geq 0$	$x_0 \geq 0$	$x_2 \geq 0$	`false`

inductive sequence of interpolants

`true`	$x \geq 0$	$x \geq 0$	$x \geq 0$	`false`

SmtInterpol

- SMT-Solver
  Computes sequences of Craig interpolants for the quantifier free combined theory of uninterpreted functions and linear arithmetic over rationals and integers.
- Developed by

Jürgen Christ

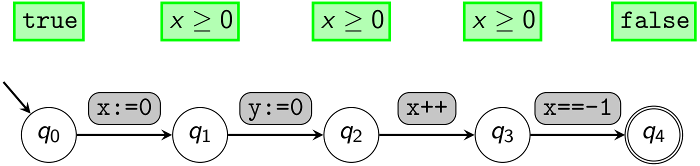
Jochen Hoenicke

- http://swt.informatik.uni-freiburg.de/research/tools/smtinterpol

Outline

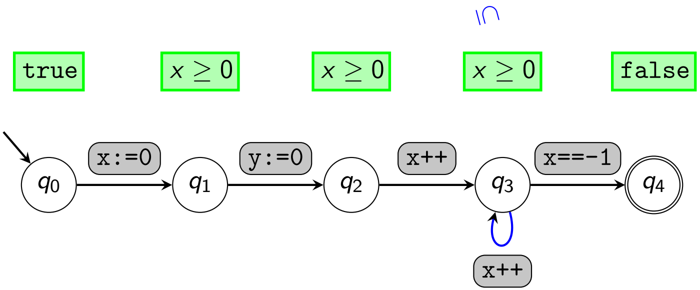
- Formal setting / Our point of view:
  A program is a language over the alphabet of statements.
- Excursion: interpolants
- Trace Abstraction with interpolants
- Trace Abstraction for recursive programs

Example – Use Interpolants to Generalize Infeasible Traces



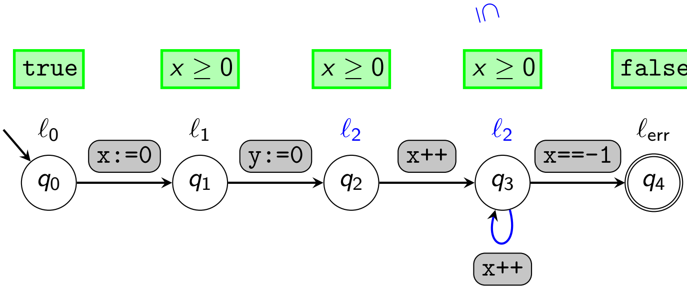
Example – Use Interpolants to Generalize Infeasible Traces

$$\text{post}(x \geq 0, \texttt{x++}) = x \geq 1$$

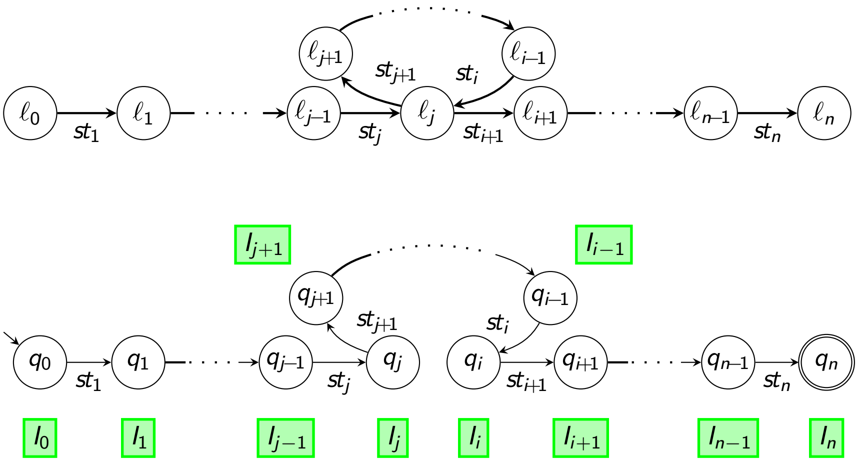


Example – Use Interpolants to Generalize Infeasible Traces

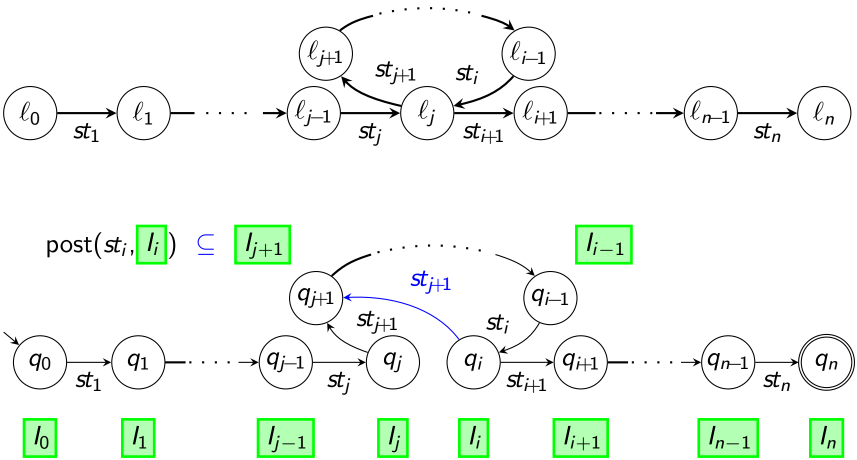
$$\text{post}(x \geq 0, \texttt{x++}) = x \geq 1$$



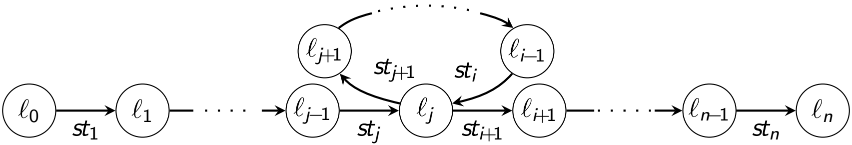
Schematic Example – Use Interpolants for Generalization

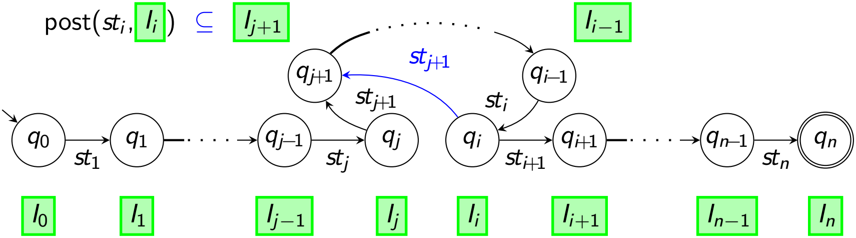


Schematic Example – Use Interpolants for Generalization

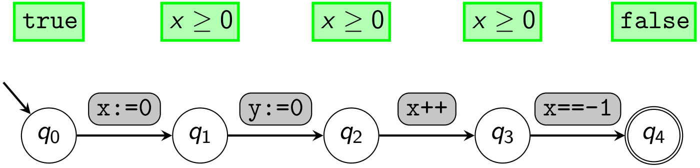


Schematic Example – Use Interpolants for Generalization

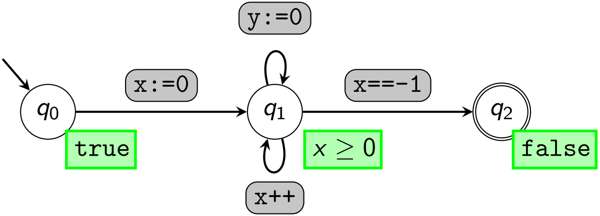




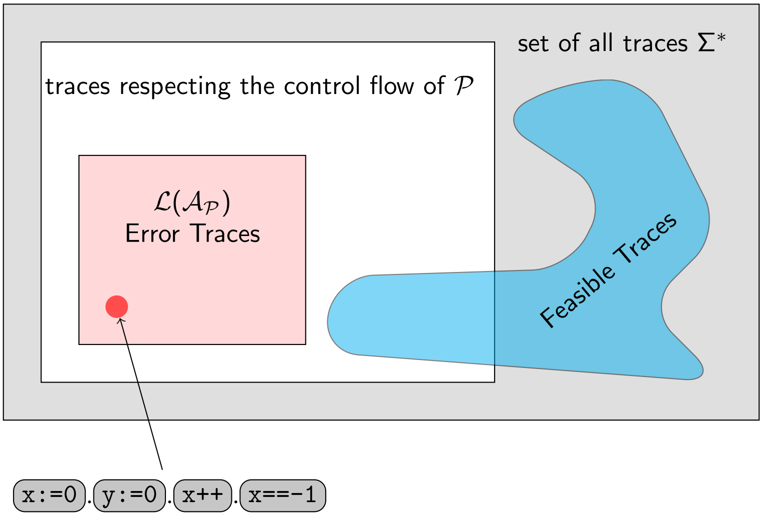
Example – Use Interpolants to Generalize Infeasible Traces



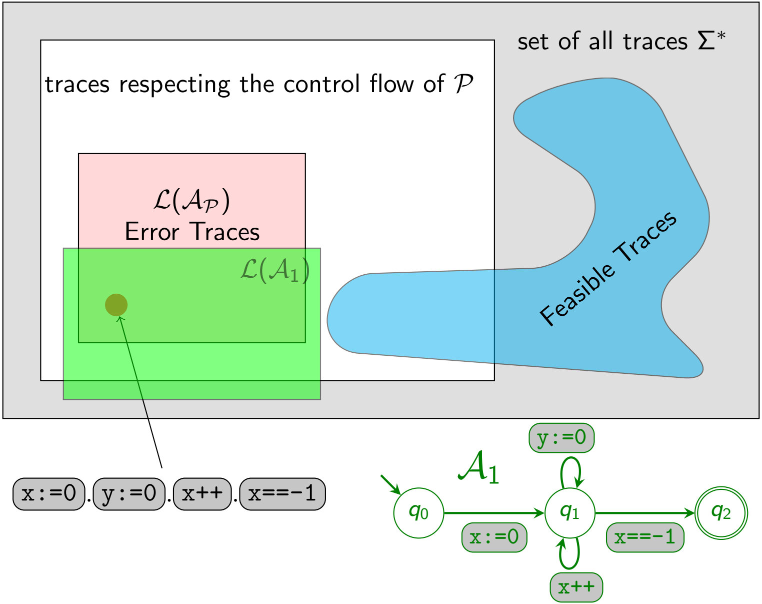
Interpolant automaton
obtained by merging all states labelled with same interpolant



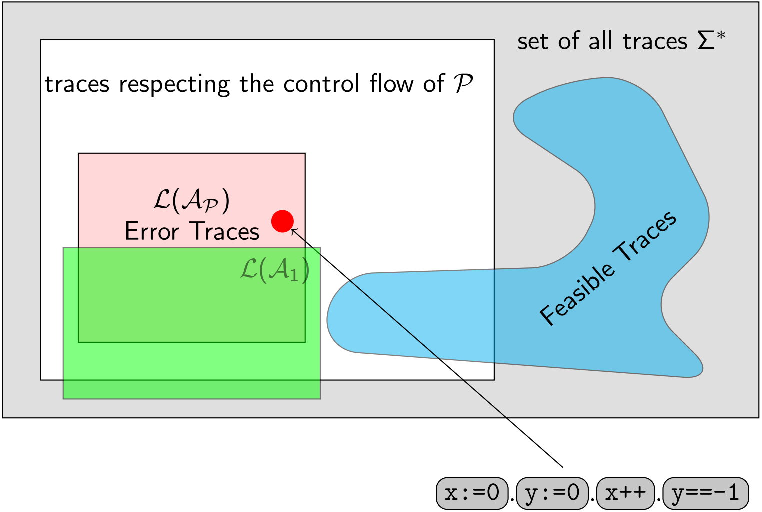
Example – Refinement Using Interpolant Automata



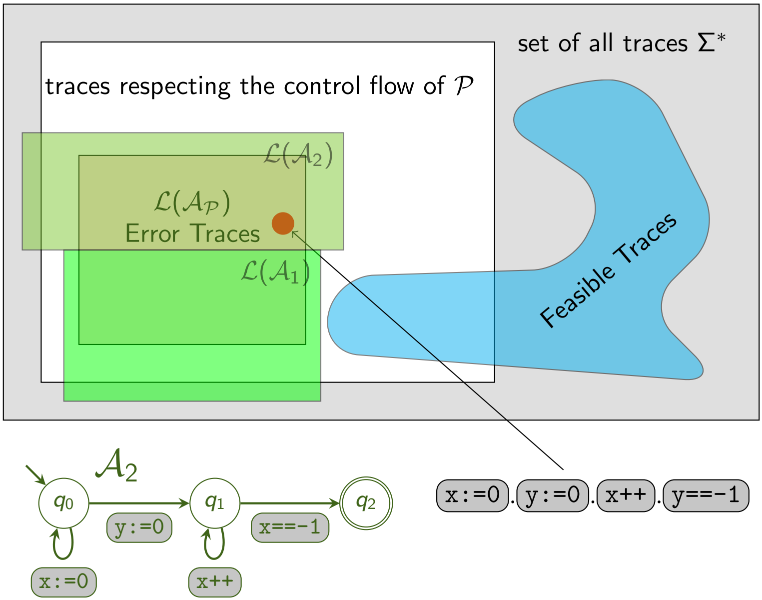
Example – Refinement Using Interpolant Automata



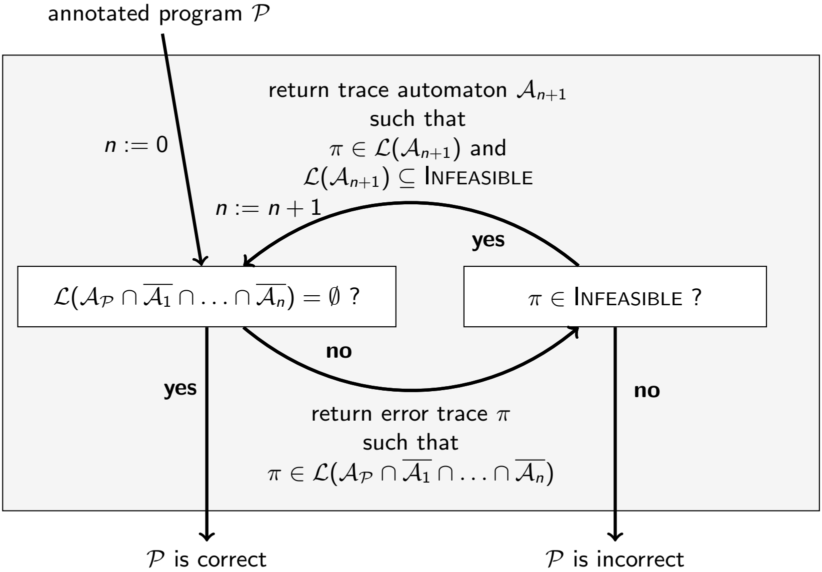
Example – Refinement Using Interpolant Automata



Example – Refinement Using Interpolant Automata



CEGAR for Trace Abstraction



Outline

- Formal setting / Our point of view:
  A program is a language over the alphabet of statements.
- Excursion: interpolants
- Trace Abstraction with interpolants
- Trace Abstraction for recursive programs

Recursive programs - challange 1: control flow

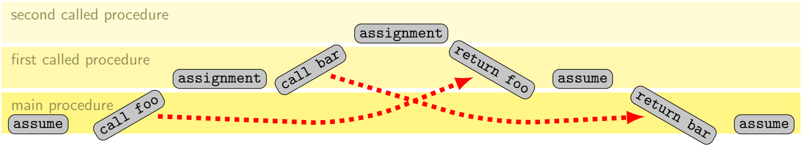
Problem:
Sequence of statements that does not respect call-return-discipline

`assume` `call foo` `assignment` `call bar` `assignment` `return foo` `assume` `return bar` `assume`

Regular languages / finite automata not suitable to model control flow of recursive program

Recursive programs - challange 1: control flow

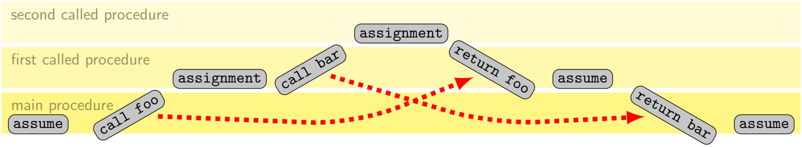
Problem:
Sequence of statements that does not respect call-return-discipline



Regular languages / finite automata not suitable to model control flow of recursive program

Recursive programs - challange 1: control flow

Problem:
Sequence of statements that does not respect call-return-discipline



Regular languages / finite automata not suitable to model control flow of recursive program

Idea: Use context free languages / pushdown automata

Context free languages are not closed under intersection

Solution 1

Visibly pushdown languages / visibly pushdown automata.

Partition symbols. Type of symbol determines stack behaviour

- Call symbol Must push one element on stack.
- Internal symbol Must not alter stack.
- Return symbol Must pop one element from stack.

Rajeev Alur, P. Madhusudan

Visibly pushdown languages (STOC 2004)

Modelling control flow

- Partition statements

  `assume` `call foo` `assignment` `call bar` `assignment` `return foo` `assume` `return bar` `assume`

- Store return address on stack

Solution 2

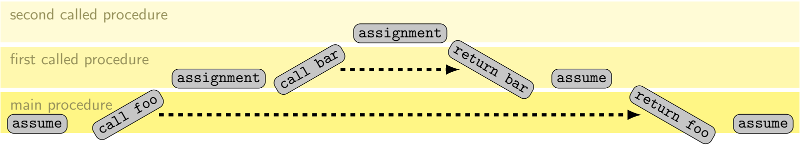
Nested word languages / nested word automata.

Add call-return dependency explicitly to the word

Nested word = word + nesting relation

Rajeev Alur, P. Madhusudan

Adding nesting structure to words (DLT 2006, J. ACM 56(3) 2009)



Solution 2

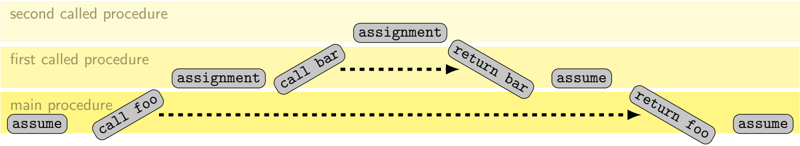
Nested word languages / nested word automata.

Add call-return dependency explicitly to the word

Nested word = word + nesting relation

Rajeev Alur, P. Madhusudan

Adding nesting structure to words (DLT 2006, J. ACM 56(3) 2009)



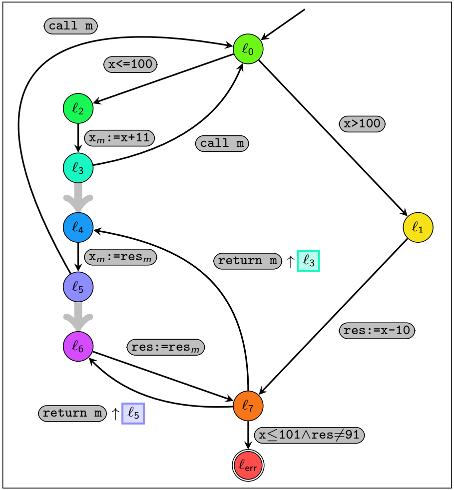
visibly pushdown vs. nested word

	input	device
visibly pushdown automata	simple	complex (stack)
nested word automata	complex (nesting relation)	simple

Example - control flow as nested word automata

```
procedure m(x) returns (res)

ℓ0: if x>100

ℓ1:   res:=x-10
    else

ℓ2:   x_m := x+11

ℓ3:   call m

ℓ4:   x_m := res_m

ℓ5:   call m

ℓ6:   res := res_m

ℓ7: assert (x<=101 -> res=91)
    return m
```

McCarthy 91 function



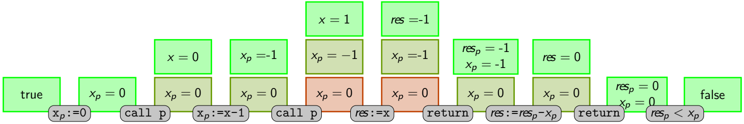
nested word automaton

nested word automaton has 4-ary return relations e.g. $(\ell_7, \ell_5, \texttt{return m}, \ell_6) \in \delta_{return}$

Recursive programs - challange 2: interpolants

What is an interpolant for an interprocedural execution?

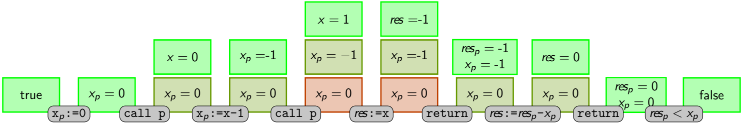
- state with a stack?
  ⇝ locality of interpolant is lost



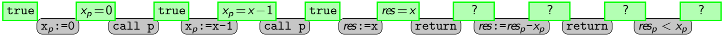
Recursive programs - challange 2: interpolants

What is an interpolant for an interprocedural execution?

- state with a stack?
  ⇝ locality of interpolant is lost



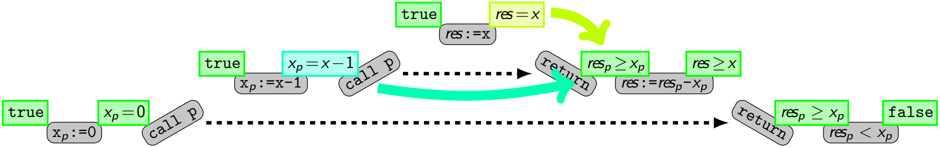
- only local valuations?
  ⇝ call/return dependency lost,
  ⇝ sequence of interpolants is not a proof



Recursive programs - challange 2: interpolants

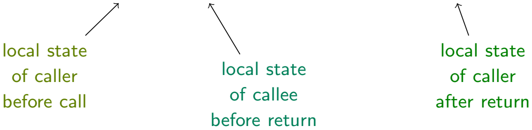
What is an interpolant for an interprocedural execution?

Idea: "Nested Interpolants"
Define sequence of interpolants with respect to nested trace.



Define ternary post operator for return statements

$$post(res = x, x_p = x - 1, \texttt{return p}) \subseteq res_p \geq x_p$$



Control flow as nested word automata

`procedure m(x) returns (res)`

ℓ_0: `if x>100`

ℓ_1: `  res:=x-10`
`else`

ℓ_2: `  x`$_m$ `:= x+11`

ℓ_3: `  call m`

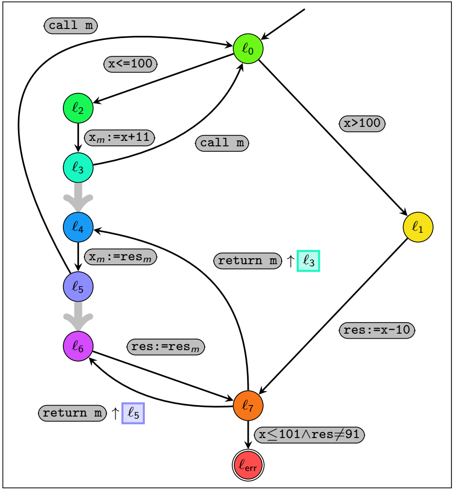
ℓ_4: `  x`$_m$ `:= res`$_m$

ℓ_5: `  call m`

ℓ_6: `  res := res`$_m$

ℓ_7: `assert (x<=101 -> res=91)`
`return m`

McCarthy 91 function

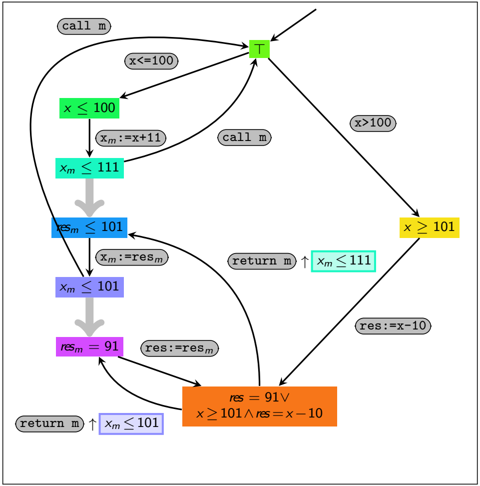


nested word automaton

Floyd-Hoare proof as nested word automata

```
procedure m(x) returns (res)
        {⊤}
ℓ0: if x>100
        {x ≥ 101}
ℓ1:   res:=x-10
    else
        {x ≤ 100}
ℓ2:   x_m := x+11
        {x_m ≤ 111}
ℓ3:   call m
        {res_m ≤ 101}
ℓ4:   x_m := res_m
        {x_m ≤ 101}
ℓ5:   call m
        {res_m = 91}
ℓ6:   res := res_m
        {res = 91 ∨ (x ≥ 101 ∧ res = x − 10)}
ℓ7: assert (x<=101 -> res=91)
    return m
```

McCarthy 91 function



nested word automaton

Floyd-Hoare proof as nested word automata

procedure m(x) returns (res)

$\{\top\}$

ℓ_0: if x>100

$\{x \geq 101\}$

ℓ_1: res:=x-10

else

$\{x \leq 100\}$

ℓ_2: x_m := x+11

$\{x_m \leq 111\}$

ℓ_3: call m

$\{res_m \leq 101\}$

ℓ_4: x_m := res_m

$\{x_m \leq 101\}$

ℓ_5: call m

$\{res_m = 91\}$

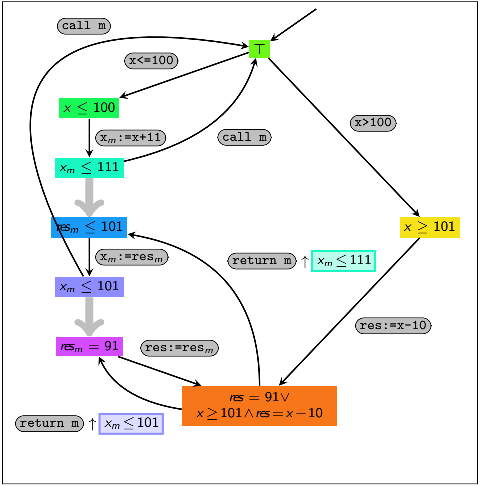
ℓ_6: res := res_m

$\{res = 91 \ \vee \ (x \geq 101 \wedge res = x - 10)\}$

ℓ_7: assert (x<=101 -> res=91)

return m

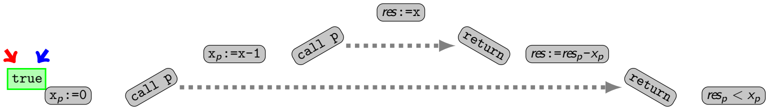
McCarthy 91 function



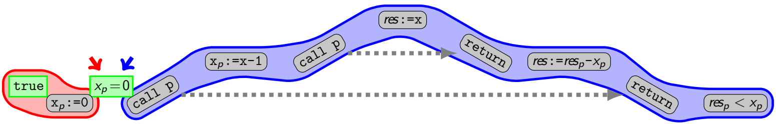
nested word automaton

e.g. $post\big(x \leq 100, \ x_m\text{:=x+11}\big) \subseteq x_m \leq 111$

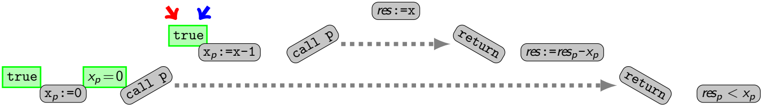
Computation of nested interpolants - Example



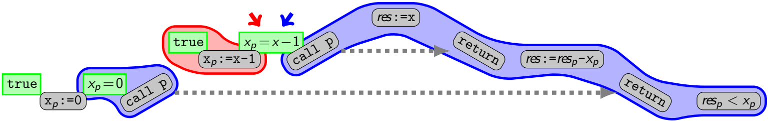
Computation of nested interpolants - Example



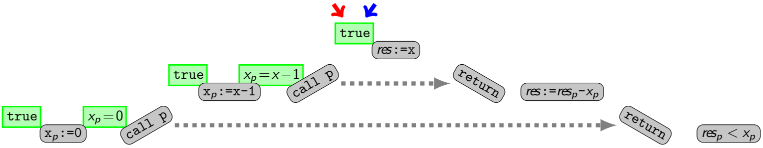
Computation of nested interpolants - Example



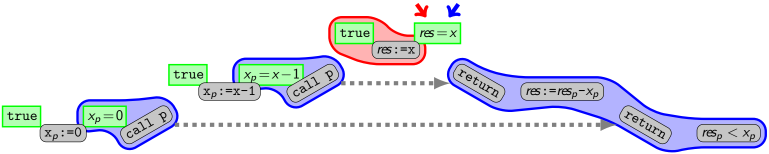
Computation of nested interpolants - Example



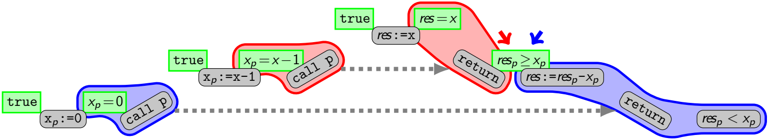
Computation of nested interpolants - Example



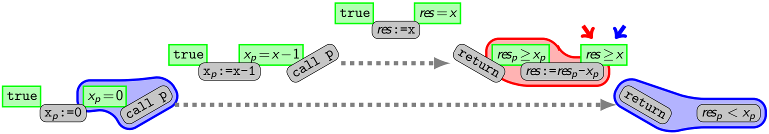
Computation of nested interpolants - Example



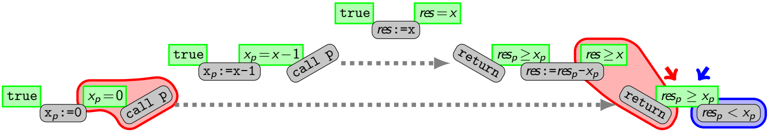
Computation of nested interpolants - Example



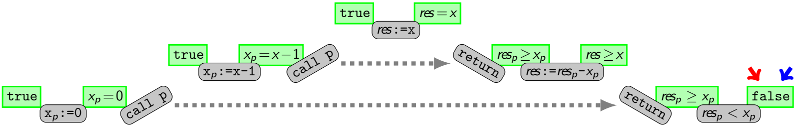
Computation of nested interpolants - Example



Computation of nested interpolants - Example



Computation of nested interpolants - Example



Conclusion

Trace Abstraction

- Refine abstraction by using independent underapproximations of infeasible traces.
- Use interpolants directly to create a component of the abstraction. Economic use of theorem prover.
- Use nested words to define inductive sequence of interpolants for recursive programs.

Future Work

- Liveness properties
- Concurrent Programs
- Caching infeasibility: reuse abstractions from one program to another.
- Guided generation of interpolants (strength of interpolants)