How to obtain and prove invariants

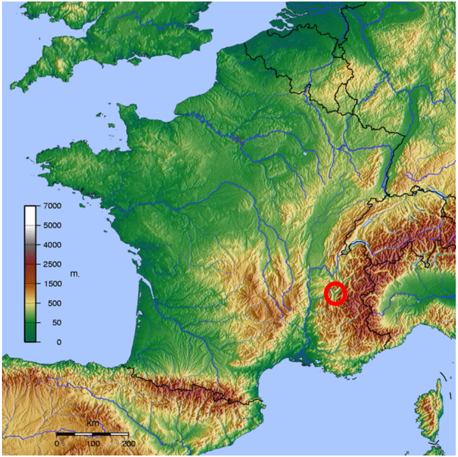
David Monniaux

CNRS / VERIMAG (Grenoble)

September 28, 2014

Project STATOR, VERIMAG, Grenoble

http://stator.imag.fr/

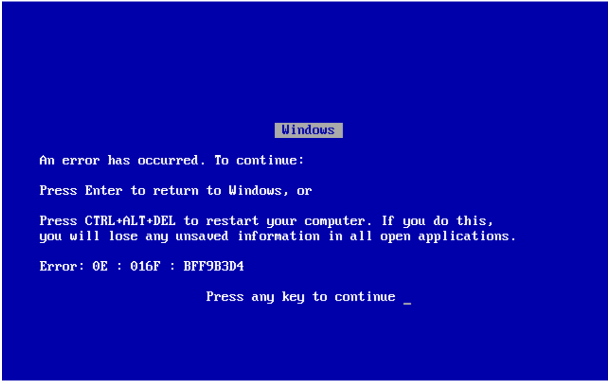


Project STATOR, VERIMAG, Grenoble

http://stator.imag.fr/

In short

Prove that a **program** does not end in the **wrong place**.



Systems we consider

Software considered as a discrete-time transition system.

Nondeterminism:

External components inputs, operating system...

Due to modeling libraries that we do not want to consider in detail (e.g. $\sin(x) \in [-1, 1]$)...

Safety properties

In this talk: properties of the form
"$\forall$ execution, $\forall$ state along execution, property P holds"

Otherwise said: "the system cannot reach $\neg P$"

Obvious use: $\neg P$ is the **error states**

- runtime errors (division by zero, null pointer, array access out of bounds, invalid cast...)
- assertion violations

Safety properties on states for input/output specifications

Often, specifications "$\forall x \in I\ f(x) = f_{\text{spec}}(x)$" (deterministic function)
Or "$\forall x \in I\ r(x, f(x))$" (nondeterministic function)

Solution Keep a copy of input state x in the current state (x, l, y) where l program location and y current program variables
$P(x, l, y)$ is $l = \text{final} \Rightarrow r(x, y)$

Safety properties on states for input/output specifications

Often, specifications "$\forall x \in I\ f(x) = f_{\text{spec}}(x)$" (deterministic function)
Or "$\forall x \in I\ r(x, f(x))$" (nondeterministic function)

Solution Keep a copy of input state x in the current state (x, l, y) where l program location and y current program variables
$P(x, l, y)$ is $l = \text{final} \Rightarrow r(x, y)$

Safety properties for termination

1. Choose an integer ranking function ρ
2. Prove that it decreases along all executions
3. Prove that it remains nonnegative

3 is directly a safety property

2 also is (record previous state along with current state)

First idea: direct proof by induction

Prove

1. for all initial state σ, $P(\sigma)$
2. **(induction step)** for all consecutive states $\sigma \rightarrow \sigma'$, $P(\sigma) \Rightarrow P(\sigma')$

Weakness?

A mathematical example

- initialization $x := 0$
- step $x \to x^2$

Prove $x < 100$.
Not inductive.
Yet $x = 0$ always! Property obviously always true!

Note: $x = 0 \Rightarrow x < 100$.
Strenghtened property is inductive.

A mathematical example

- initialization $x := 0$
- step $x \to x^2$

Prove $x < 100$.
Not inductive.
Yet $x = 0$ always! Property obviously always true!

Note: $x = 0 \Rightarrow x < 100$.
Strenghtened property is inductive.

A loopy example

```
for(int i=0; i!=100; i++) { }
```

Loop body:

- initialization $i := 0$
- step $i \to i + 1$ if $i \neq 100$

Prove $i < 200$. Inductive?

Not inductive. Yet **stronger** $i \leq 100$ inductive!

A loopy example

```
for(int i=0; i!=100; i++) { }
```

Loop body:

- initialization $i := 0$
- step $i \to i + 1$ if $i \neq 100$

Prove $i < 200$. Inductive?

Not inductive. Yet **stronger** $i \leq 100$ inductive!

Vocabulary: invariants

For some authors:

- **Invariant:** property that always holds on all traces
- **Inductive invariant:** invariant that can be proved correct by a proof of induction

For other authors or e.g. Floyd-Hoare proofs: "invariant" means "inductive invariant".

Strenghtening: executive summary

The property to prove is almost never inductive.

Replace it by a **stronger, inductive property**.
Invariant strengthening

(Basis of Floyd-Hoare rule for loops.)

Relative completeness

If a property is always true, does there necessarily exist a stronger, inductive property?

Yes, always.

But...

Relative completeness

If a property is always true, does there necessarily exist a stronger, inductive property?

Yes, always.

But...

The set of reachable states

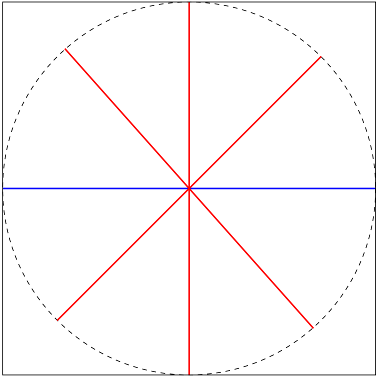
The set R of reachable states is the **least inductive invariant**:

- contains the initial states I
- stable by $\rightarrow$

Equal to $\{y \mid x \rightarrow^* y, x \in I\}$

P always holds on all traces iff $R \Rightarrow P$
(identify sets and properties)

Example: (non)inductive invariants



Initialize: $x \in [-1, 1]$, $y = 0$.
Step: rotate by 45°

Reachable states = star
Non inductive invariant: $[-1, 1] \times [-1, 1]$
Inductive invariants: octagon or disc

As a least fixed point

Initialize $S_0 = \emptyset$

Iterate $S_{k+1} \mapsto \text{step}(S_k) \cup \text{initial states}$

Ascending sequence, its limit is the set of reachable states

If finite state space (size n), converges in n
If infinite state space... quite useless!

Decidability

Assume initial state + transition relation in linear arithmetic (e.g. counter machine)

Is membership in the set of reachable states decidable?

No, since it allows deciding the **halting problem** over deterministic counter machines (test reachability of final state).

Decidability

Assume initial state + transition relation in linear arithmetic (e.g. counter machine)

Is membership in the set of reachable states decidable?

No, since it allows deciding the **halting problem** over deterministic counter machines (test reachability of final state).

Halting

In Peano arithmetic

(Cook's completeness theorem)

If

- memory state is in $\mathbb{Z}^n$
- initial states: first-order formula over n variables
- transition relation: first-order formula over $n + n$ variables

Then the set of reachable states is defined in Peano arithmetic (Gödel encoding).

(But Peano arithmetic is undecidable! Can't even show that the invariant implies the property!)

In Peano arithmetic

(Cook's completeness theorem)

If

- memory state is in $\mathbb{Z}^n$
- initial states: first-order formula over n variables
- transition relation: first-order formula over $n + n$ variables

Then the set of reachable states is defined in Peano arithmetic (Gödel encoding).

(But Peano arithmetic is undecidable! Can't even show that the invariant implies the property!)

Undecidability of arithmetic

Reversing the arrows

So far: compute inductive invariant, show no bad states inside (**forward**)

Alternative: compute inductive invariant on **reverse** transition system, starting from bad states; show no initial state in inside (**backward**)

Combinations

Our weapons

Abstraction Search for invariant in a restricted class

Extrapolation Find inductive invariant in a finite number of steps

Refinement If invariant too coarse, refine it

Tricks Cheap and not-so cheap improvements

Our goals

Depending on usage:

Guided by a property Prove a given property P

Unguided Provide "strong" invariants (best-effort)

A simple idea

Replace arbitrary sets of states (or traces etc.) by **symbolically** represented sets and **compute** on them.

Intervals

State space $= \mathbb{Z}^d$ (d integer variables)
Compute one interval per variable per program point

```
int x, y, z; //x ∈ [−7, 5]
if (x >= 0) { //x ∈ [0, 5]
  y = x; //y ∈ [0, 5]
} else { //x ∈ [−7, −1]
  y = -x; //y ∈ [1, 7]
} //y ∈ [0, 7]
z = 2*y + x; //z ∈ [−7, 19]
```

Intervals

State space = $\mathbb{Z}^d$ (d integer variables)
Compute one interval per variable per program point

```
int x, y, z; //x ∈ [−7, 5]
if (x >= 0) { //x ∈ [0, 5]
  y = x; //y ∈ [0, 5]
} else { //x ∈ [−7, −1]
  y = -x; //y ∈ [1, 7]
} //y ∈ [0, 7]
z = 2*y + x; //z ∈ [−7, 19]
```

Intervals

State space = $\mathbb{Z}^d$ (d integer variables)
Compute one interval per variable per program point

```
int x, y, z; //x ∈ [−7, 5]
if (x >= 0) { //x ∈ [0, 5]
  y = x; //y ∈ [0, 5]
} else { //x ∈ [−7, −1]
  y = -x; //y ∈ [1, 7]
} //y ∈ [0, 7]
z = 2*y + x; //z ∈ [−7, 19]
```

Intervals

State space = $\mathbb{Z}^d$ (d integer variables)
Compute one interval per variable per program point

```
int x, y, z; //x ∈ [−7, 5]
if (x >= 0) { //x ∈ [0, 5]
  y = x; //y ∈ [0, 5]
} else { //x ∈ [−7, −1]
  y = -x; //y ∈ [1, 7]
} //y ∈ [0, 7]
z = 2*y + x; //z ∈ [−7, 19]
```

Intervals

State space $= \mathbb{Z}^d$ (d integer variables)
Compute one interval per variable per program point

```
int x, y, z; //x ∈ [−7, 5]
if (x >= 0) { //x ∈ [0, 5]
  y = x; //y ∈ [0, 5]
} else { //x ∈ [−7, −1]
  y = -x; //y ∈ [1, 7]
} //y ∈ [0, 7]
z = 2*y + x; //z ∈ [−7, 19]
```

Lack of relationality

```
int x, y, z; //x ∈ [0,1]
y = x; //y ∈ [0,1]
z = x - y; //z ∈ [−1,1]
```

No relation tracked between x and y thus z **over-approximated**.

Recovering relationality

E.g. SSA form and symbolic expressions.

```
int x, y, z; //x ∈ [0,1]
y = x; //y ∈ [0,1]
z = x - x;
```

Recovering relationality

Simplification step (see Miné)

```
int x, y, z; //x ∈ [0,1]
y = x; //y ∈ [0,1]
z = 0; //z ∈ [0,0]
```

Code motion

```
int x, y, z; //x ∈ [−7, 5]
if (x >= 0) { //x ∈ [0, 5]
  y = x; //y ∈ [0, 5]
  z = 2*y + x; //z ∈ 0, 15]
} else { //x ∈ [−7, −1]
  y = -x; //y ∈ [1, 7]
  z = 2*y + x; //z ∈ [−5, 13]
} //z ∈ [−5, 15]
```

(Recall: without motion, $z \in [-7, 19]$)

Code motion

```
int x, y, z; //x ∈ [−7, 5]
if (x >= 0) { //x ∈ [0, 5]
  y = x; //y ∈ [0, 5]
  z = 2*y + x; //z ∈ 0, 15]
} else { //x ∈ [−7, −1]
  y = -x; //y ∈ [1, 7]
  z = 2*y + x; //z ∈ [−5, 13]
} //z ∈ [−5, 15]
```

(Recall: without motion, $z \in [-7, 19]$)

Code motion

```
int x, y, z; //x ∈ [−7, 5]
if (x >= 0) { //x ∈ [0, 5]
  y = x; //y ∈ [0, 5]
  z = 2*y + x; //z ∈ 0, 15]
} else { //x ∈ [−7, −1]
  y = -x; //y ∈ [1, 7]
  z = 2*y + x; //z ∈ [−5, 13]
} //z ∈ [−5, 15]
```

(Recall: without motion, $z \in [-7, 19]$)

Code motion

```
int x, y, z; //x ∈ [−7, 5]
if (x >= 0) { //x ∈ [0, 5]
  y = x; //y ∈ [0, 5]
  z = 2*y + x; //z ∈ 0, 15]
} else { //x ∈ [−7, −1]
  y = -x; //y ∈ [1, 7]
  z = 2*y + x; //z ∈ [−5, 13]
} //z ∈ [−5, 15]
```

(Recall: without motion, $z \in [-7, 19]$)

Code motion

```
int x, y, z; //x ∈ [−7, 5]
if (x >= 0) { //x ∈ [0, 5]
  y = x; //y ∈ [0, 5]
  z = 2*y + x; //z ∈ 0, 15]
} else { //x ∈ [−7, −1]
  y = -x; //y ∈ [1, 7]
  z = 2*y + x; //z ∈ [−5, 13]
} //z ∈ [−5, 15]
```

(Recall: without motion, $z \in [-7, 19]$)

Code motion and symbolic expressions

```
int x, y, z;
if (x >= 0)
  y = x;
  z = 2*x + x;
} else {
  y = -x;
  z = 2*(-x) + x;
}
```

Code motion and symbolic expressions

```
int x, y, z; //x ∈ [−7, 5]
if (x >= 0) { //x ∈ [0, 5]
  y = x; //y ∈ [0, 5]
  z = 3*x; //z ∈ [0, 15]
} else { //x ∈ [−7, −1]
  y = -x; //y ∈ [1, 7]
  z = -x; //z ∈ [1, 7]
} //z ∈ [0, 15]
```

Code motion and symbolic expressions

```
int x, y, z; //x ∈ [−7, 5]
if (x >= 0) { //x ∈ [0, 5]
  y = x; //y ∈ [0, 5]
  z = 3*x; //z ∈ [0, 15]
} else { //x ∈ [−7, −1]
  y = -x; //y ∈ [1, 7]
  z = -x; //z ∈ [1, 7]
} //z ∈ [0, 15]
```

Code motion and symbolic expressions

```
int x, y, z; //x ∈ [−7, 5]
if (x >= 0) { //x ∈ [0, 5]
  y = x; //y ∈ [0, 5]
  z = 3*x; //z ∈ [0, 15]
} else { //x ∈ [−7, −1]
  y = -x; //y ∈ [1, 7]
  z = -x; //z ∈ [1, 7]
} //z ∈ [0, 15]
```

Implementation note: exploit sparsity

```
/* 10000 variables initialized */
if (x > 0) {
  y = 1;
}
```

The merge operation at the end of if-then-else should **not** perform 9999 useless interval merges.

Intervals: summary

- $+$ Easy to implement
- $+$ Low complexity
- $+$ Handles $+$, $-$, $\times$, $/$, $\sqrt{}$...
- \- No relations
- \- Enforces convexity
- $=$ Some relations may be recovered by symbolic rewriting

Forward analysis can be insufficient

```
double x, z;
if (abs(x) > 0.1) {
  assert(z != 0);
  z = 1.0 / x;
  ...
}
```

```
double x, y, z;
if (x >= 0) { //x ≥ 0
  y = x; //  y ≥ 0
} else { //  x < 0
  y = -x; //  y > 0
} //y ≥ 0
if (y > 0.1) { //y > 0.1
  assert(x != 0); //FAIL
  z = 1.0 / x;
}
```

Forward analysis can be insufficient

```
double x, z;
if (abs(x) > 0.1) {
  assert(z != 0);
  z = 1.0 / x;
  ...
}
```

```
double x, y, z;
if (x >= 0) { //x ≥ 0
  y = x; //   y ≥ 0
} else { //   x < 0
  y = -x; //   y > 0
} //y ≥ 0
if (y > 0.1) { //y > 0.1
  assert(x != 0); //FAIL
  z = 1.0 / x;
}
```

Forward analysis can be insufficient

```
double x, z;
if (abs(x) > 0.1) {
  assert(z != 0);
  z = 1.0 / x;
  ...
}
```

```
double x, y, z;
if (x >= 0) { //x ≥ 0
  y = x; //  y ≥ 0
} else { //  x < 0
  y = -x; //  y > 0
} //y ≥ 0
if (y > 0.1) { //y > 0.1
  assert(x != 0); //FAIL
  z = 1.0 / x;
}
```

Forward analysis can be insufficient

```
double x, z;
if (abs(x) > 0.1) {
  assert(z != 0);
  z = 1.0 / x;
  ...
}
```

```
double x, y, z;
if (x >= 0) { //x ≥ 0
  y = x; //  y ≥ 0
} else { //  x < 0
  y = -x; //  y > 0
} //y ≥ 0
if (y > 0.1) { //y > 0.1
  assert(x != 0); //FAIL
  z = 1.0 / x;
}
```

Forward analysis can be insufficient

```
double x, z;
if (abs(x) > 0.1) {
  assert(z != 0);
  z = 1.0 / x;
  ...
}
```

```
double x, y, z;
if (x >= 0) { //x ≥ 0
  y = x; //  y ≥ 0
} else { //  x < 0
  y = -x; //  y > 0
} //y ≥ 0
if (y > 0.1) { //y > 0.1
  assert(x != 0); //FAIL
  z = 1.0 / x;
}
```

Forward analysis can be insufficient

```
double x, z;
if (abs(x) > 0.1) {
  assert(z != 0);
  z = 1.0 / x;
  ...
}
```

```
double x, y, z;
if (x >= 0) { //x ≥ 0
  y = x; //  y ≥ 0
} else { //  x < 0
  y = -x; //  y > 0
} //y ≥ 0
if (y > 0.1) { //y > 0.1
  assert(x != 0); //FAIL
  z = 1.0 / x;
}
```

Code motion

```
double x, y, z;
if (x >= 0) { //x ≥ 0
  y = x; //y ≥ 0
  if (y > 0.1) { //y > 0.1
    assert(x != 0); //FAIL
    z = 1.0 / x;
  }
} else { //x < 0
  y = -x; //y > 0
  if (y > 0.1) { //y > 0.1
    assert(x != 0); //FAIL
    z = 1.0 / x;
  }
}
```

Code motion

```
double x, y, z;
if (x >= 0) { //x ≥ 0
  y = x; //y ≥ 0
  if (y > 0.1) { //y > 0.1
    assert(x != 0); //FAIL
    z = 1.0 / x;
  }
} else { //x < 0
  y = -x; //y > 0
  if (y > 0.1) { //y > 0.1
    assert(x != 0); //FAIL
    z = 1.0 / x;
  }
}
```

Code motion

```
double x, y, z;
if (x >= 0) { //x ≥ 0
  y = x; //y ≥ 0
  if (y > 0.1) { //y > 0.1
    assert(x != 0); //FAIL
    z = 1.0 / x;
  }
} else { //x < 0
  y = -x; //y > 0
  if (y > 0.1) { //y > 0.1
    assert(x != 0); //FAIL
    z = 1.0 / x;
  }
}
```

Code motion

```
double x, y, z;
if (x >= 0) { //x ≥ 0
  y = x; //y ≥ 0
  if (y > 0.1) { //y > 0.1
    assert(x != 0); //FAIL
    z = 1.0 / x;
  }
} else { //x < 0
  y = -x; //y > 0
  if (y > 0.1) { //y > 0.1
    assert(x != 0); //FAIL
    z = 1.0 / x;
  }
}
```

Code motion

```
double x, y, z;
if (x >= 0) { //x ≥ 0
  y = x; //y ≥ 0
  if (y > 0.1) { //y > 0.1
    assert(x != 0); //FAIL
    z = 1.0 / x;
  }
} else { //x < 0
  y = -x; //y > 0
  if (y > 0.1) { //y > 0.1
    assert(x != 0); //FAIL
    z = 1.0 / x;
  }
}
```

Forward, then backward

```
double x, y, z;
if (x >= 0) { //x ≥ 0
  y = x; //y ≥ 0
} else { //x < 0
  y = -x; //y > 0
}
if (y > 0.1) { //y > 0.1
  assert(x != 0);
  z = 1.0 / x;
}
```

```
double x, y, z; //x = 0
if (x >= 0) { //x ≥ 0
  y = x; //y ≥ 0 ∧ x = 0
} else { //
  y = -x; //y > 0 ∧ x = 0
}
if (y > 0.1) { //y > 0.1 ∧ x = 0
  if (x == 0) {
    fail(); //  x = 0
  }
  z = 1.0 / x;
}
```

Forward, then backward

```
double x, y, z;
if (x >= 0) { //x ≥ 0
  y = x; //y ≥ 0
} else { //x < 0
  y = -x; //y > 0
}
if (y > 0.1) { //y > 0.1
  assert(x != 0);
  z = 1.0 / x;
}
```

```
double x, y, z; //x = 0
if (x >= 0) { //x ≥ 0
  y = x; //y ≥ 0 ∧ x = 0
} else { //
  y = -x; //y > 0 ∧ x = 0
}
if (y > 0.1) { //y > 0.1 ∧ x = 0
  if (x == 0) {
    fail(); //  x = 0
  }
  z = 1.0 / x;
}
```

Forward, then backward

```
double x, y, z;
if (x >= 0) { //x ≥ 0
  y = x; //y ≥ 0
} else { //x < 0
  y = -x; //y > 0
}
if (y > 0.1) { //y > 0.1
  assert(x != 0);
  z = 1.0 / x;
}
```

```
double x, y, z; //x = 0
if (x >= 0) { //x ≥ 0
  y = x; //y ≥ 0 ∧ x = 0
} else { //x < 0
  y = -x; //y > 0 ∧ x = 0
}
if (y > 0.1) { //y > 0.1 ∧ x = 0
  if (x == 0) {
    fail(); //  x = 0
  }
  z = 1.0 / x;
}
```

Forward, then backward

```
double x, y, z;
if (x >= 0) { //x ≥ 0
  y = x; //y ≥ 0
} else { //x < 0
  y = -x; //y > 0
}
if (y > 0.1) { //y > 0.1
  assert(x != 0);
  z = 1.0 / x;
}
```

```
double x, y, z; //x = 0
if (x >= 0) { //x = 0
  y = x; //y ≥ 0 ∧ x = 0
} else { //⊥
  y = -x; //y > 0 ∧ x = 0
}
if (y > 0.1) { //y > 0.1 ∧ x = 0
  if (x == 0) {
    fail(); //  x = 0
  }
  z = 1.0 / x;
}
```

Forward, then backward

```
double x, y, z;
if (x >= 0) { //x ≥ 0
  y = x; //y ≥ 0
} else { //x < 0
  y = -x; //y > 0
}
if (y > 0.1) { //y > 0.1
  assert(x != 0);
  z = 1.0 / x;
}
```

```
double x, y, z; //x = 0
if (x >= 0) { //x = 0
  y = x; //y ≥ 0 ∧ x = 0
} else { //⊥
  y = -x; //y > 0 ∧ x = 0
}
if (y > 0.1) { //y > 0.1 ∧ x = 0
  if (x == 0) {
    fail(); //  x = 0
  }
  z = 1.0 / x;
}
```

Forward, then backward, then forward

```
double x, y, z; //x = 0
if (x >= 0) { //x0
  y = x; //y ≥ 0 ∧ x = 0
} else { //
  y = -x; //y > 0 ∧ x = 0
}
if (y > 0.1) { //y > 0.1 ∧ x = 0
  if (x == 0) {
    fail(); //   x = 0
  }
  z = 1.0 / x;
}
```

```
double x, y, z; //x = 0
if (x >= 0) { //x = 0
  y = x; // y = 0
} else { // ⊥
  y = -x; // ⊥
} // y = 0
if (y > 0.1) {  // ⊥
  if (x == 0) {  // ⊥
    fail();  // ⊥
  }
  z = 1.0 / x;
}
```

Forward, then backward, then forward

```
double x, y, z; //x = 0
if (x >= 0) { //x0
  y = x; //y ≥ 0 ∧ x = 0
} else { //
  y = -x; //y > 0 ∧ x = 0
}
if (y > 0.1) { //y > 0.1 ∧ x = 0
  if (x == 0) {
    fail(); //   x = 0
  }
  z = 1.0 / x;
}
```

```
double x, y, z; //x = 0
if (x >= 0) { //x = 0
  y = x; // y = 0
} else { // ⊥
  y = -x; // ⊥
} // y = 0
if (y > 0.1) {  // ⊥
  if (x == 0) {  // ⊥
    fail();  // ⊥
  }
  z = 1.0 / x;
}
```

Forward, then backward, then forward

```
double x, y, z; //x = 0
if (x >= 0) { //x ≥ 0
  y = x; //y ≥ 0 ∧ x = 0
} else { //x < 0
  y = -x; //y > 0 ∧ x = 0
}
if (y > 0.1) { //y > 0.1 ∧ x = 0
  if (x == 0) {
    fail(); //  x = 0
  }
  z = 1.0 / x;
}
```

```
double x, y, z; //x = 0
if (x >= 0) { //x = 0
  y = x; // y = 0
} else { // ⊥
  y = -x; // ⊥
} // y = 0
if (y > 0.1) {  // ⊥
  if (x == 0) {  // ⊥
    fail();  // ⊥
  }
  z = 1.0 / x;
}
```

Forward, then backward, then forward

```
double x, y, z; //x = 0
if (x >= 0) { //x = 0
  y = x; //y ≥ 0 ∧ x = 0
} else { //⊥
  y = -x; //y > 0 ∧ x = 0
}
if (y > 0.1) { //y > 0.1 ∧ x = 0
  if (x == 0) {
    fail(); //  x = 0
  }
  z = 1.0 / x;
}
```

```
double x, y, z; //x = 0
if (x >= 0) { //x = 0
  y = x; // y = 0
} else { // ⊥
  y = -x; // ⊥
} // y = 0
if (y > 0.1) {  // ⊥
  if (x == 0) {  // ⊥
    fail();  // ⊥
  }
  z = 1.0 / x;
}
```

Forward, then backward, then forward

```
double x, y, z; //x = 0
if (x >= 0) { //x = 0
  y = x; //y ≥ 0 ∧ x = 0
} else { //⊥
  y = -x; //y > 0 ∧ x = 0
}
if (y > 0.1) { //y > 0.1 ∧ x = 0
  if (x == 0) {
    fail(); //  x = 0
  }
  z = 1.0 / x;
}
```

```
double x, y, z; //x = 0
if (x >= 0) { //x = 0
  y = x; // y = 0
} else { // ⊥
  y = -x; // ⊥
} // y = 0
if (y > 0.1) {  // ⊥
  if (x == 0) {  // ⊥
    fail();  // ⊥
  }
  z = 1.0 / x;
}
```

Forward, backward, forward with Interproc

http://pop-art.inrialpes.fr/interproc/concurinterprocweb.cgi

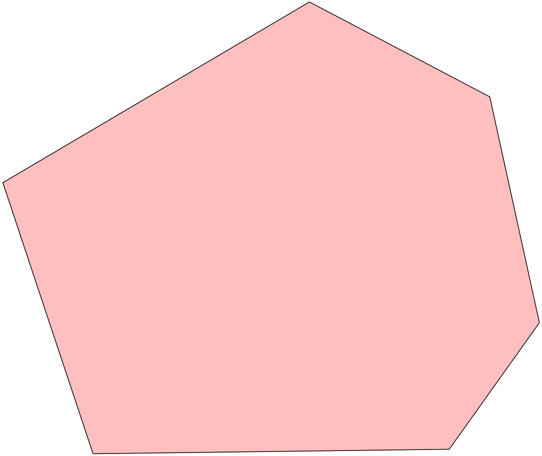
```
var x,y,z: real;

begin
  if x >= 0 then
    y = x;
  else
    y = -x;
  endif;
  if 10 * y >= 1 then
    if x == 0 then
      fail;
    endif;
  endif;
end
```

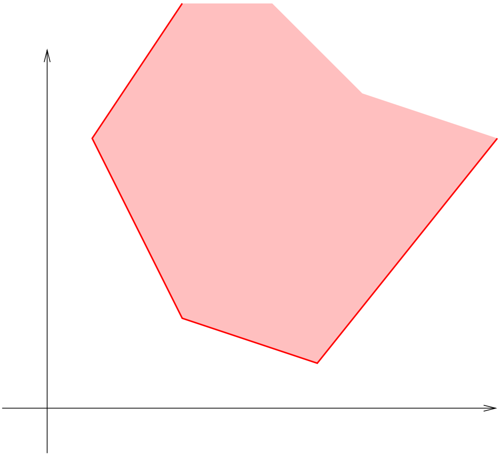
Forward: gets to “fail”

Forward — backward — forward: “fail” unreachable

Convex polyhedra



Unbounded convex polyhedra



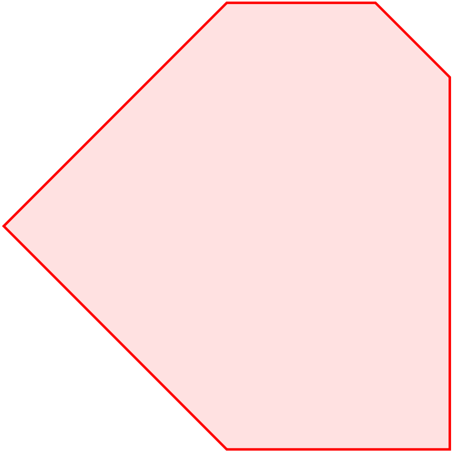
Example of an inductive polyhedron

```
int i, n;
assume(n >= 0);
i = 0;
while(i < n) {
  i = i+1;
}
```

```
int i, n;
assume(n >= 0);
i = 0;
LOOP: // 0 ≤ i ≤ n
  if (i >= n) goto EXIT;
  i = i+1;
  goto LOOP;
EXIT:
```

Octagons

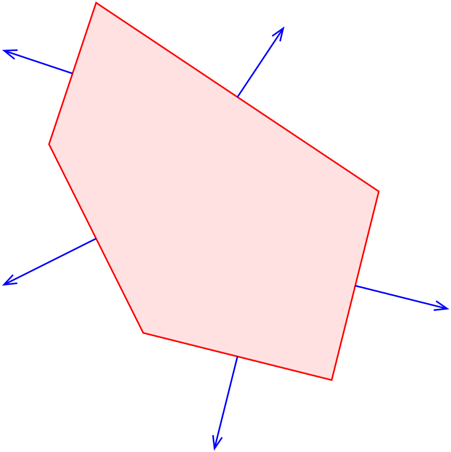
$\pm x \pm y \leq C$ for all variables x and y



(Algorithmics: variants of Floyd-Warshall)

Linear templates

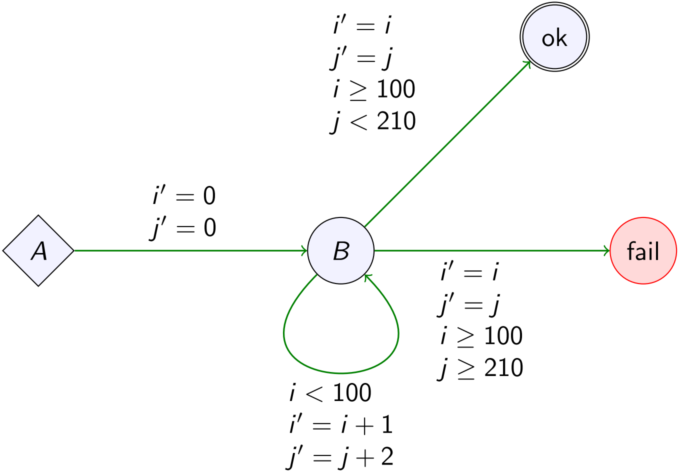
$A\vec{x} \leq B$, A set matrix, $\vec{x}$ program variables



(Algorithmics: linear programming)

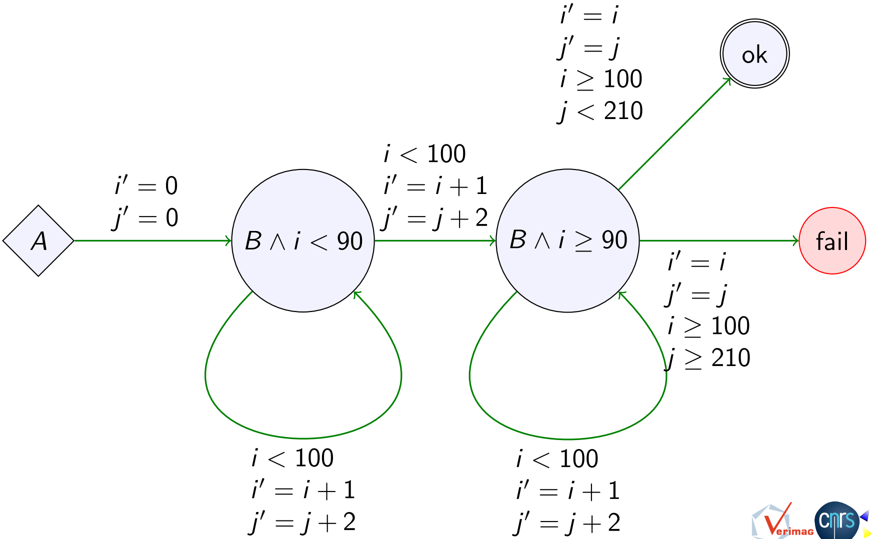
A simple example

```
for(int i=0; i<100; i++) {
  j = j+2;
}
assert(j < 210);
```



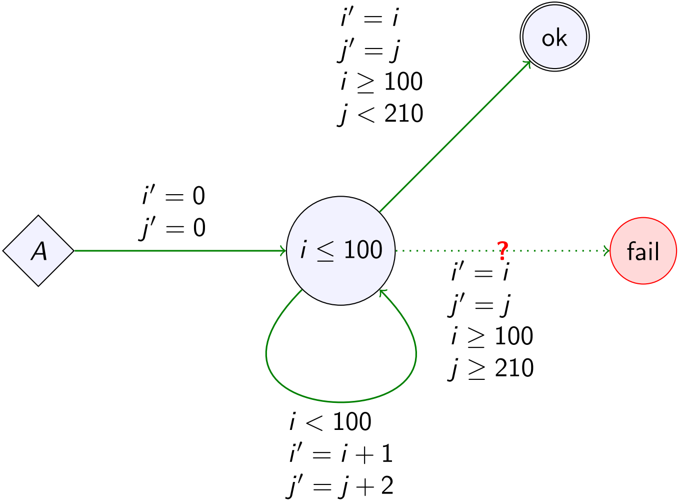
Unsuccessful predicate abstraction

Split control node B according to $i < 90$



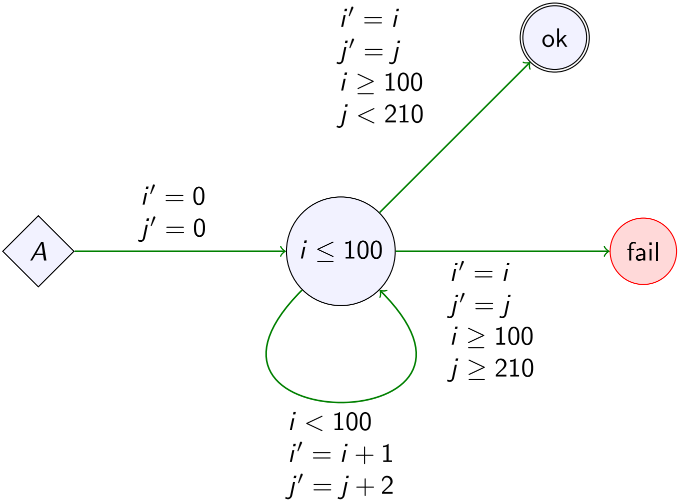
Unsuccessful predicate abstraction

Split control node B according to $i \leq 100$



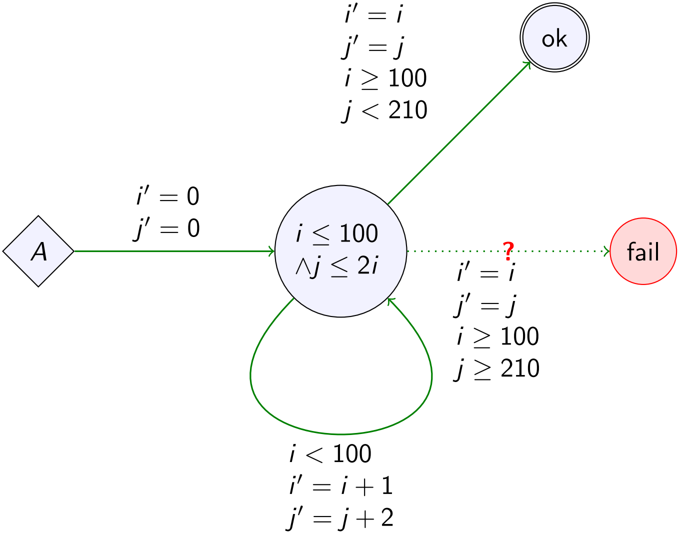
Unsuccessful predicate abstraction

Split control node B according to $i \leq 100$



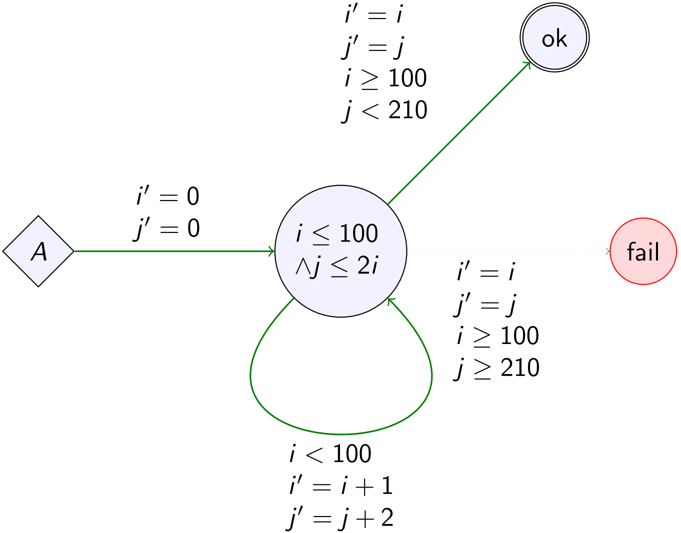
Successful predicate abstraction

Split control node B according to $i \leq 100$ and $j \leq 2i$



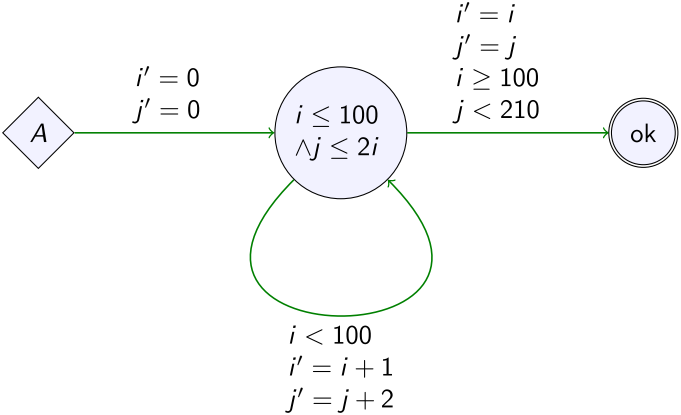
Successful predicate abstraction

Split control node B according to $i \leq 100$ and $j \leq 2i$



Pruning by construction

Do not generate unreachable states



Summary

Select a finite set of predicates (possibly depending on control location)

Split each control state according to predicates (use subsumption)

Draw feasible transitions

Procedure in finite time: only finitely many abstract states

Some decision problem

Given

- initial state
- transition relation (in fixed class)
- final state

answer whether there exist inductive invariant in fixed class.

Vary class of relations and class of invariants.

Proved on template polyhedra:

- ???? for intervals for linear transitions (related to mean-payoff?)
- Σ_2^P-completeness (even single $x \leq B$ interval bound) for transitions with $\vee, \wedge, \exists$-first-order linear arithmetic
- EXPTIME-hardness, in NEXPTIME, for implicit CFG (in the Papadimitriou-Yannakakis sense)

Concrete acceleration

Given $\rightarrow$, compute transitive closure $\rightarrow^{+}$

example: $x \rightarrow x' \iff x < n \wedge x' = x + 1$
transitive closure: $y \rightarrow^{+} y' \iff y < y' \leq n$

Possible for certain subclasses of formulas.
e.g. conjunction of "octagonal" inequalities $\pm x \pm y \leq C$ (Bozga, Iosif, Konečný)

Disjunctions

Let a, b... be accelerable relations.

```
if (*) {
  a
} else {
  b
}
```

$(a|b)^+ = a^+|b^+|a^+b^+|b^+a^+|\ldots$

Replace normal Kleene iterations by partial acceleration.

(Bozga, Iosif, Konečný) FLATA tool

Disjunctions

Let a, b... be accelerable relations.

```
if (*) {
  a
} else {
  b
}
```

$(a|b)^+ = a^+|b^+|a^+b^+|b^+a^+|\ldots$
Replace normal Kleene iterations by partial acceleration.

(Bozga, Iosif, Konečný) FLATA tool

Abstract acceleration

Given τ, compute $\alpha(\tau^+)$ where $\alpha(X)$ is "strongest abstraction" in a certain domain.

(Gonnord & Halbwachs; Jeannet, Schrammel & Sankaranarayanan)

Finite automata

Saturation in finite time:

- Finite concrete state (e.g. n Boolean variables): compute exact reachable states
- Predicate abstraction with fixed set of predicates: compute reachable abstract states

Terminates (can be slow)

Intervals

```
int i=0;
while(i < 1000000) { //
  i++;
}
```

Follow edges, merge etc. until stabilization.

Intervals

```
int i=0;
while(i < 1000000) { // i ∈ [0,0]
  i++;
}
```

Follow edges, merge etc. until stabilization.

Intervals

```
int i=0;
while(i < 1000000) { // i ∈ [0,1]
  i++;
}
```

Follow edges, merge etc. until stabilization.

Intervals

```
int i=0;
while(i < 1000000) { // i ∈ [0,2]
  i++;
}
```

Follow edges, merge etc. until stabilization.

Intervals

```
int i=0;
while(i < 1000000) { // i ∈ [0,3]
  i++;
}
```

Follow edges, merge etc. until stabilization.

Intervals

```
int i=0;
while(i < 1000000) { // i ∈ [0,4]
  i++;
}
```

Follow edges, merge etc. until stabilization.

Intervals

```
int i=0;
while(i < 1000000) { // i ∈ [0,5]
  i++;
}
```

Follow edges, merge etc. until stabilization.

Intervals

```
int i=0;
while(i < 1000000) { // i ∈ [0,6]
  i++;
}
```

Follow edges, merge etc. until stabilization.

Intervals

```
int i=0;
while(i < 1000000) { // i ∈ [0,7]
  i++;
}
```

Follow edges, merge etc. until stabilization.

Intervals

```
int i=0;
while(i < 1000000) { // i ∈ [0,8]
  i++;
}
```

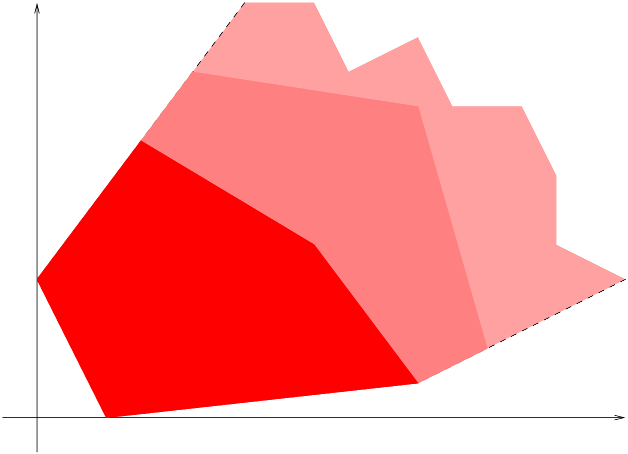
Follow edges, merge etc. until stabilization.

Interval widening

```
int i=0;
while(i < 1000000) { // i ∈ [0, +∞)
  i++;
}
```

Idea: unstable bounds go to $+\infty$

Widening on polyhedra



(Halbwachs)

Interval widening "up to"

Syntactic tracking/extraction of bounds in the program; here $i \to 1000000$

```
int i=0;
while(i < 1000000) { // i ∈ [0, 1000000]
  i++;
}
```

Resembles predicate abstraction!
Can be generalized to polyhedra (see Jeannet).

Upward "Kleene" iterations

- Iterate (upward, using merges) until a fixed point is reached.
- If possibility of infinite ascending sequence (e.g. intervals, polyhedra), apply **widening** instead of normal merge.

Where do we need to apply widening?

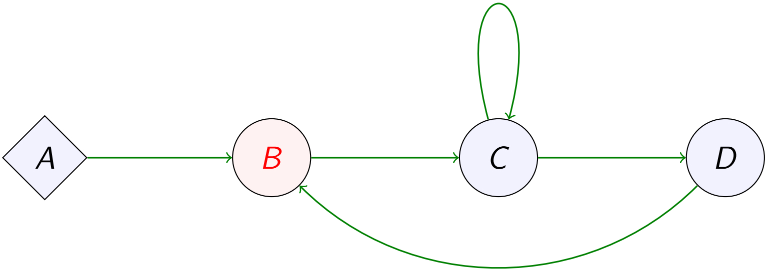
Need to apply widening only at "cut set" of control states.

Upward “Kleene” iterations

- Iterate (upward, using merges) until a fixed point is reached.
- If possibility of infinite ascending sequence (e.g. intervals, polyhedra), apply **widening** instead of normal merge.

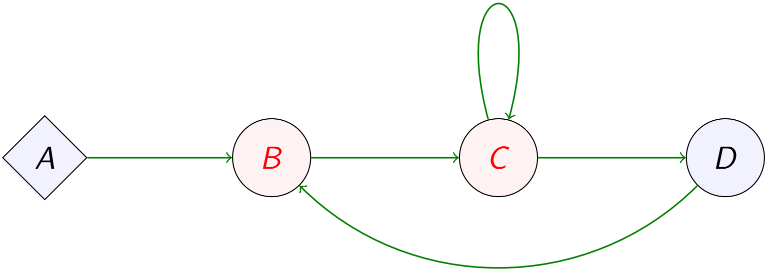
Where do we need to apply widening?
Need to apply widening only at “cut set” of control states.

Cut sets



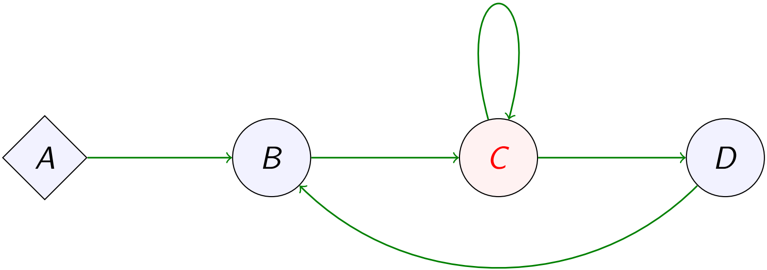
Insufficient: uncut loop at C

Cut sets



Loop heads

Cut sets



Minimal

How to compute a cut set

- Simple but not optimal: target of back edges in depth-first search
- Least cardinality: NP-complete in general
- . . . but linear time for structured programs (reducible cfg) [Shamir '79]

How to compute a cut set

- Simple but not optimal: target of back edges in depth-first search
- Least cardinality: NP-complete in general
- ...but linear time for structured programs (reducible cfg) [Shamir '79]

Non monotonicity

```
int i=0;
while(i != 1000000) { //
  i++;
}
```

Non monotonicity

```
int i=0;
while(i != 1000000) { // i ∈ [0,0]
  i++;
}
```

Non monotonicity

```
int i=0;
while(i != 1000000) { // i ∈ [0,1]
  i++;
}
```

Non monotonicity

```
int i=0;
while(i != 1000000) { // i ∈ [0,2]
  i++;
}
```

Non monotonicity

```
int i=0;
while(i != 1000000) { // i ∈ [0,3]
  i++;
}
```

Non monotonicity

```
int i=0;
while(i != 1000000) { // i ∈ [0,4]
  i++;
}
```

Non monotonicity

```
int i=0;
while(i != 1000000) { // i ∈ [0,5]
  i++;
}
```

Non monotonicity

```
int i=0;
while(i != 1000000) { // i ∈ [0,6]
  i++;
}
```

Non monotonicity

```
int i=0;
while(i != 1000000) { // i ∈ [0, +∞)
  i++;
}
```

Non monotonicity

```
int i=choose(0, 1000000); // i ∈ [0,1000000]
while(i != 1000000) { //
  i++;
}
```

Non monotonicity

```
int i=choose(0, 1000000); // i ∈ [0,1000000]
while(i != 1000000) { // i ∈ [0,1000000]
  i++;
}
```

A very simple loop

```
i=0;
while (i < 100) {
  i=i+1;
}
```

Find an inductive loop invariant as an interval $[-l, h]$:

- $[-l, h]$ must contain the initial state: $l \geq 0,\ h \geq 0$
- $[-l, h]$ must be stable by "pushing the interval through the loop"
  - test maps $[-l, h]$ to $[-l, \min(h, 99)]$
  - then $i = i + 1$ maps $[-l, \min(h, 99)]$ to $[-(l-1), \min(h, 99) + 1]$

  Thus inclusion: $l \geq l - 1$ and $h \geq \min(h, 99) + 1$

Thus the least solution satisfies

- $l = \max(0, l - 1)$
- $h = \max(0, \min(h, 99) + 1)$

How to solve min-max equations

We end with equations with “min”, “max”, and monotone affine-linear expressions

$$h = \max(0, \min(h, 99) + 1)$$

How to solve them?

Naive approach:

- Enumerate all argument choices for “min” and “max”
- For each choice, compute solution of linear equation system
- Discard if not a solution of the original problem (wrong choices of arguments of “min” and “max”)
- Take the least one

How to solve min-max equations

We end with equations with “min”, “max”, and monotone affine-linear expressions

$$h = \max(0, \min(h, 99) + 1)$$

How to solve them?

Naive approach:

- Enumerate all argument choices for “min” and “max”
- For each choice, compute solution of linear equation system
- Discard if not a solution of the original problem (wrong choices of arguments of “min” and “max”)
- Take the least one

Solving the naive way

$$h = \max(0, \min(h, 99) + 1) \quad (1)$$

Turned into 3 different equations:

- $h = \max(\underline{0}, \min(h, 99) + 1) \rightsquigarrow h = 0$ (left-arg to "max"), solution $h = 0$, but not solution of (1): $\max(0, \min(0, 99) + 1)$, the right argument of "max" is greater $\Rightarrow$ discarded
- $h = \max(0, \underline{\min(\underline{h}, 99) + 1}) \rightsquigarrow h = h + 1$ (right-arg to "max", left-arg to "min"), solution $h = +\infty$, but not solution of (1): $\min(+\infty, 99)$, the argument of "min" is smaller $\Rightarrow$ discarded
- $h = \max(0, \underline{\min(h, \underline{99}) + 1}) \rightsquigarrow h = 99 + 1 = 100$ (right-arg to "max", right-arg to "min"), **solution** of the original problem.

But **exponential blowup**.

Max-policy iteration

(Developed by H. Seidl, T. Gawlitza)

$$h = \max(-\infty, 0, \min(h, 99) + 1)$$

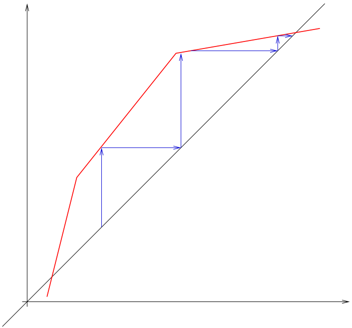
Pick an argument for "max":

- Initial value for $h = -\infty$
- $h = \max(\underline{-\infty}, 0, \min(h, 99) + 1)$; $h = -\infty$; replace: $\max(-\infty, \underline{0}, -\infty)$, found higher argument $h = 0$
- $h = \max(-\infty, \underline{0}, \min(h, 99) + 1)$; $h = 0$; replace: $\max(-\infty, 0, \underline{1})$, found higher argument $h = 1$
- $h = \max(-\infty, 0, \underline{\min(h, 99) + 1})$; solve $h = \min(h, 99) + 1$ for solution $h \geq 1$:
  Solve $h \leq h + 1 \wedge h \leq 99 + 1$ for maximal finite h: $h = 100$.

High level view

Transforms the original problem (with "max") into a sequence of problems (without "max") with increasing "value".

Intuition: solution is maximum of "order-concave" functions



It's like solving $h = F(h)$ by infinite ascending sequence
$-\infty, F(-\infty), F \circ F(-\infty), F \circ F \circ F(-\infty)\dots$
but taking "big strides"!

Max policy iteration: executive summary

- Works for linear templates (and some nonlinear ones, but much more difficult)
- For linear transitions (and...)
- "Policy": for each control node, and each bound in the template, pick an incoming edge ($-\infty$ = unreachable, $+\infty$ = unbounded)
- Solve resulting system for greatest finite fixed point
- Change policy if unstable

Replaces blind widening and widening "up to" by a kind of acceleration of ω Kleene iteration steps.

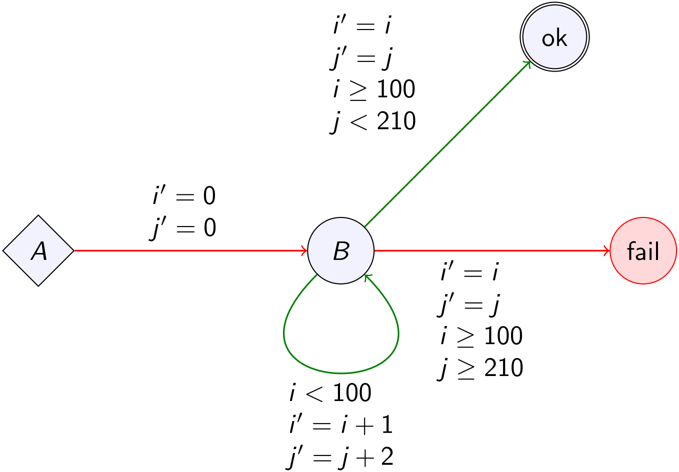
Refinement

From a property or invariant, get a stronger one.

- **Unguided** Get a stronger one.
- **Guided** Get a stronger one that "kills" some counterexample.

In predicate abstraction

```
for(int i=0; i<100; i++) {
  j = j+2;
}
assert(j < 210);
```



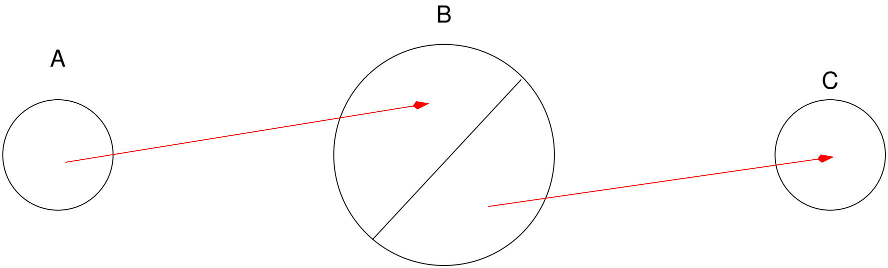
A bad counterexample

Try to find values for the red path:

$i_1 = 0 \wedge j_1 = 0 \wedge i_2 = i_1 \wedge j_2 = j_1 \wedge i \geq 100 \wedge j \geq 210$

UNSAT

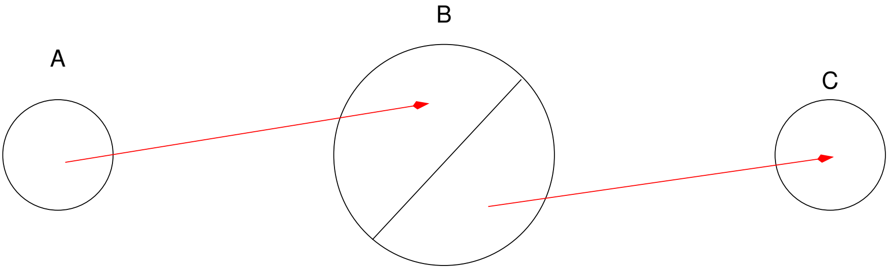
Why wrong? Can move from one state in A to one state in B, from one state in B to one in "fail". But states in B not the same.



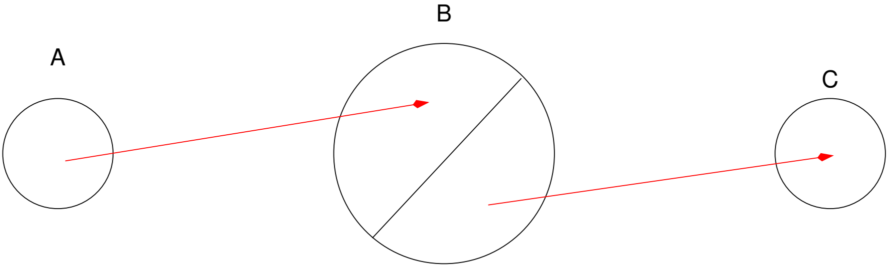
A bad counterexample

Try to find values for the red path:
$i_1 = 0 \wedge j_1 = 0 \wedge i_2 = i_1 \wedge j_2 = j_1 \wedge i \geq 100 \wedge j \geq 210$
UNSAT

Why wrong? Can move from one state in A to one state in B, from one state in B to one in "fail". But states in B not the same.



A refinement

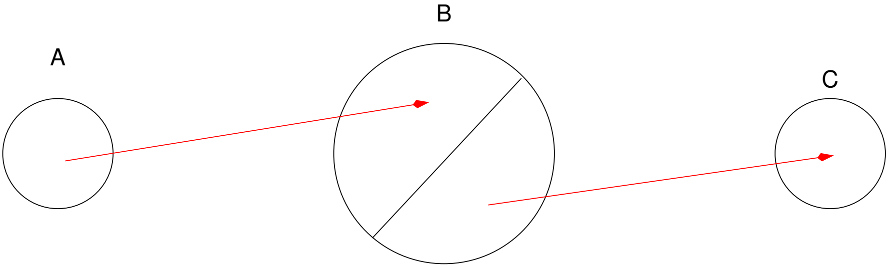


Two step explanation for infeasible path:

- $i_1 = 0 \wedge j_1 = 0 \wedge i_2 = i_1 \wedge j_2 = j_1 \Rightarrow j_2 = 2i_2 \wedge i \geq 100$
- $j_2 = 2i_2 \wedge i_2 \geq 100 \Rightarrow j_2 < 210$

This is a **Craig interpolant**.

A refinement



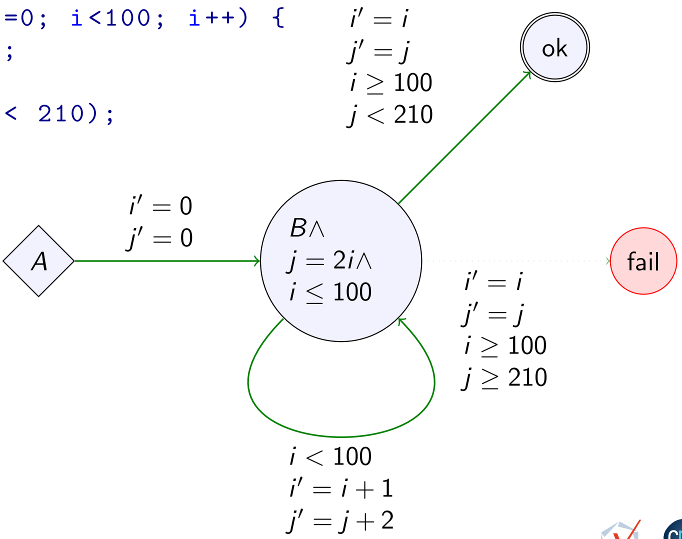
Two step explanation for infeasible path:

- $i_1 = 0 \wedge j_1 = 0 \wedge i_2 = i_1 \wedge j_2 = j_1 \Rightarrow j_2 = 2i_2 \wedge i \geq 100$
- $j_2 = 2i_2 \wedge i_2 \geq 100 \Rightarrow j_2 < 210$

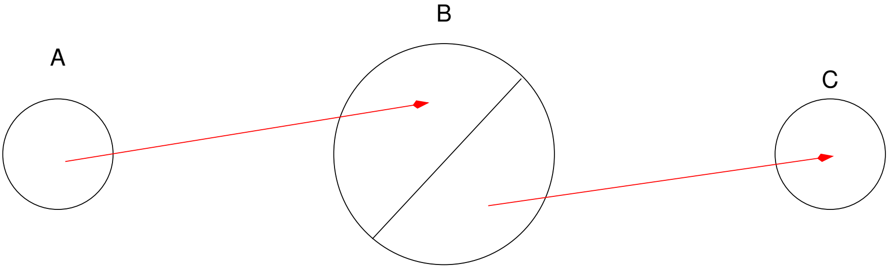
This is a **Craig interpolant**.

A good refinement

```
for(int i=0; i<100; i++) {
  j = j+2;
}
assert(j < 210);
```



Another refinement



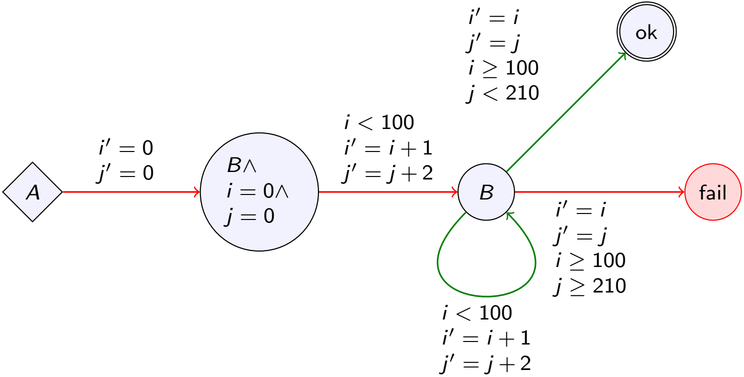
Two step explanation for infeasible path:

- $i_1 = 0 \wedge j_1 = 0 \wedge i_2 = i_1 \wedge j_2 = j_1 \Rightarrow i_2 = 0 \wedge j_2 = 0$
- $i_2 = 0 \wedge j_2 = 0 \Rightarrow j_2 < 210$

This is another **Craig interpolant**.

Another refinement

```
for(int i=0; i<100; i++) {
  j = j+2;
}
assert(j < 210);
```



Further refinement

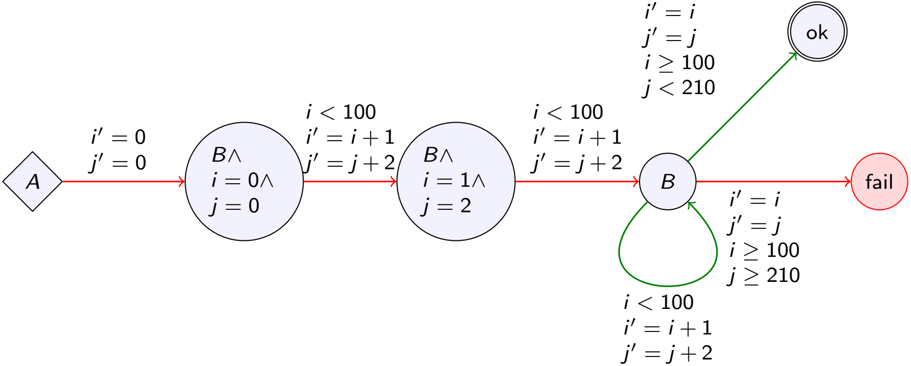
Two step explanation for infeasible path:

- $i_1 = 0 \land j_1 = 0 \land i_2 = i_1 \land j_2 = j_1 \Rightarrow i_2 = 0 \land j_2 = 0$
- $i_2 = 0 \land j_2 = 0 \land i_3 = i_2 + 1 \land j_3 = j_2 + 2 \Rightarrow i_3 = 1 \land j_3 = 2$
- $i_3 = 1 \land j_3 = 2 \Rightarrow j_2 < 210$

This is another **Craig interpolant**.

Further refinement

```
for(int i=0; i<100; i++) {
  j = j+2;
}
assert(j < 210);
```



Overfitting and convergence

- Interpolant $j = 2i \wedge i \leq 100$ (polyhedral inductive invariant) proves the property.
- Interpolants $i = 0 \wedge j = 0$, $i = 1 \wedge j = 2$, $i = 2 \wedge j = 4$ ... (exact post-conditions) lead to non-termination.

Challenge: find "good" interpolants "likely" to become inductive
Problem similar to widening
McMillan: find "short" interpolants using few "magic" constants?

When simple induction is insufficient

```
int index = 0, tab[8];
while(true) {
  out = tab[index];
  tab[index] = input(-10, 10);
  index++;
  if (index==8) index=0;
}
```

Cannot prove $out \in [-10, 10]$ by simple induction.

Otherwise said

```
while(true) {
  int t0, t1, t2, t3,
      t4, t5, t6, t7;
  out = t7;
  t7=t6; t6=t5;
  t5=t4; t4=t3;
  t3=t2; t2=t1;
  t1=t0;
  t0 = input(-10, 10);
}
```

Or in Lustre

```
node top
  (in : int)
returns
  (ok : bool);

var
  t0 : int;
  t1 : int;
  t2 : int;
  t3 : int;
  t4 : int;
  t5 : int;
  t6 : int;
  t7 : int;
  out : int;

let
```

```
  ok = out >= -10
   and out <= 10;
  out = 0 -> pre t7;
  t7 = 0 -> pre t6;
  t6 = 0 -> pre t5;
  t5 = 0 -> pre t4;
  t4 = 0 -> pre t3;
  t3 = 0 -> pre t2;
  t2 = 0 -> pre t1;
  t1 = 0 -> pre t0;
  t0 = if in >= -10
       and in <= 10
       then in else 0;

--%PROPERTY ok;

tel
```

Kind

`http://clc.cs.uiowa.edu/Kind/`
Proves the above program correct by 9-induction.

Descending sequence in abstract interpretation

```
int i=0, n;
assume(n >= 0);
while(i < n) {
  i = i+1;
}
```

Invariant inferred after widening: $i \geq 0$

Reasoning: "in order to be at loop head, either coming from initialization, either from preceding iteration"
Preceding iteration:

```
assume(i >= 0);
assume(i < n);
i = i+1;
```

Thus: $i = 0 \vee 1 \leq i \leq n$, thus $0 \leq i \leq n$ better invariant.

Executive summary: k-induction and descending sequence

k-induction Given P to prove:

- initialization Prove that P holds on k first steps of execution
- induction If P holds on k steps $x_0 \to x_1 \to \cdots \to x_{k-1}$ and $x_{k-1} \to x_k$ then P holds on x_k

Descending sequence Given any invariant I:

- initialization Keep states in k first steps of execution in I'
- induction Assume I holds on k steps $x_0 \to x_1 \to \cdots \to x_{k-1}$ and $x_{k-1} \to x_k$, add x_k states to I'.

Intuition

(E. Goubault's group)

If precondition $\rightarrow$ program $\rightarrow$ $x \leq 10$ provable in, say, octagons

a proof of $x \leq 10$ can be written using only a subset

- of the guards (tests) and other constraints (operations)
- of possible reasoning steps (e.g. "if $y - x \leq 3$ and $z - y \leq 5$ then $z - x \leq 8$")

Iterative refinement: if another subset is "obviously" better, choose it.

A very simple loop

```
i=0;
while (i < 100) {
  i=i+1;
}
```

Find an inductive loop invariant as an interval $[-l, h]$:

- $[-l, h]$ must contain the initial state: $l \geq 0,\ h \geq 0$
- $[-l, h]$ must be stable by "pushing the interval through the loop"
  - test maps $[-l, h]$ to $[-l, \min(h, 99)]$
  - then $i = i + 1$ maps $[-l, \min(h, 99)]$ to $[-(l-1), \min(h, 99) + 1]$

  Thus inclusion: $l \geq l - 1$ and $h \geq \min(h, 99) + 1$

Thus the least solution satisfies

- $l = \max(0, l - 1)$
- $h = \max(0, \min(h, 99) + 1)$

Min-policy iteration

Only choose for "min":

- $h = \max(0, \min(\underline{h}, 99) + 1) \rightsquigarrow h = \max(0, h + 1) \rightsquigarrow$ find least solution of $h \geq 0 \wedge h \geq h + 1$ (linear programming) $\rightsquigarrow h = +\infty$ $\min(+\infty, 99) = 99$, so flip to right argument of "min"
- $h = \max(0, \min(h, \underline{99}) + 1) \rightsquigarrow h = \max(0, 100) \rightsquigarrow$ find least solution of $h \geq 0 \wedge h \geq 100$ (linear programming) $\rightsquigarrow h = 100$

Solution: $h = 100$

Always the least one?

In general, the min-policy iteration process may stop on a solution of the system of min-max equation that is not the least one.

Min-policy iteration

Only choose for "min":

- $h = \max(0, \min(\underline{h}, 99) + 1) \rightsquigarrow h = \max(0, h + 1) \rightsquigarrow$ find least solution of $h \geq 0 \wedge h \geq h + 1$ (linear programming) $\rightsquigarrow h = +\infty$ $\min(+\infty, 99) = 99$, so flip to right argument of "min"
- $h = \max(0, \min(h, \underline{99}) + 1) \rightsquigarrow h = \max(0, 100) \rightsquigarrow$ find least solution of $h \geq 0 \wedge h \geq 100$ (linear programming) $\rightsquigarrow h = 100$

Solution: $h = 100$
Always the least one?

In general, the min-policy iteration process may stop on a solution of the system of min-max equation that is not the least one.

Min-policy iteration, executive summary

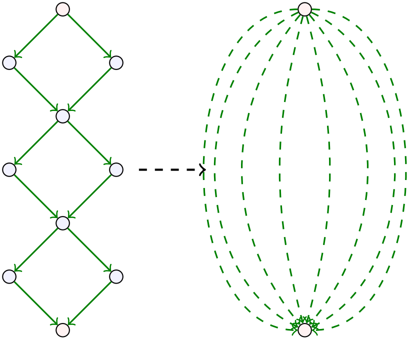
- Replaces original system by an over-approximation: leave out some guards and other constraints on the reachable states
- Solve the resulting, simple, system
- If the system solves the property, terminate
- Check whether some "obvious" improvement exists, loop

Code motion or trace partitioning

```
double x, y, z;
if (x >= 0) {
  y = x;
  if (y > 0.1) {
    assert(x != 0);
    z = 1.0 / x;
  }
} else {
  y = -x;
  if (y > 0.1) {
    assert(x != 0);
    z = 1.0 / x;
  }
}
```

In general: exponential cost

Considering individual paths



Use of SMT-solving

(Satisfiability modulo theory: given a first-order formula in a theory, say "unsat" or give a solution)

Instead of explicitly considering all 2^n paths
consider them **implicitly**, as needed

Solve successive SMT problems "is my current candidate invariant inductive"?

Applied to

- Kleene iterations + widening [Monniaux & Gonnord, C. Tinelli]
- max-policy iteration [Gawlitza & Monniaux, Monniaux & Schrammel]

Floating-point

Most abstract domains: ideal mathematics ($\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$)
Intervals: handle floating-point by directed rounding
But relational domains?

$$x \oplus y = x + y + \epsilon,\ |\epsilon| \leq \epsilon_r |x + y|$$

$$x \otimes y = x \times y + \epsilon,\ |\epsilon| \leq \min(\epsilon_a, \epsilon_r |x + y|)$$

ϵ_r "error at last bit of precision"
ϵ_a least positive floating-point value

Floating-point

Most abstract domains: ideal mathematics ($\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$)
Intervals: handle floating-point by directed rounding
But relational domains?

$$x \oplus y = x + y + \epsilon,\ |\epsilon| \leq \epsilon_r |x + y|$$

$$x \otimes y = x \times y + \epsilon,\ |\epsilon| \leq \min(\epsilon_a, \epsilon_r |x + y|)$$

ϵ_r "error at last bit of precision"
ϵ_a least positive floating-point value

Data structures

- Point-to graph (may/must)
- Abstract into a single variable
  - all data allocated at single location? (but beware of malloc-like functions)
  - all fields with same field identifier (e.g. in Java)
- Recursive decomposition of memory
- Separation logic?

Warning

List of tools **certainly not exhaustive**.

General framework

(not all tools exactly this way)

compiler front-end (Java bytecode, LLVM, CIL, Eclipse CDT...)
$\downarrow$
iterator
$\downarrow$
data structure abstractions
$\downarrow$
numeric abstractions

Polyspace

(commercial) now Mathworks

Astrée

Cousot et al. (commercial)
`http://www.astree.ens.fr/`
`http://www.absint.com/astree/index.htm`

- home-made front-end
- only abstract interpretation
- intervals and octagons
- specialized abstractions for numerical filters
- limited memory abstractions
- now support for parallel programs

Astrée applications

Large plane, large fly-by-wire code

CPAChecker

`http://cpachecker.sosy-lab.org/`

- Eclipse CDT front-end
- mostly predicate interpretation
- intervals
- limited support for octagons and polyhedra

Verasco

Proved correct in Coq

- Compcert front-end
- (floating point) intervals
- polyhedra
- memory
- in the future: filters

No public release yet

`http://verasco.imag.fr/`

APRON & Interproc

Jeannet & Miné

APRON: abstract domains
(Concur)Interproc: demonstrator analyzer

`http://apron.cri.ensmp.fr/library/`
`http://pop-art.inrialpes.fr/interproc/concurinterprocweb.cgi`

PAGAI

http://pagai.forge.imag.fr/ (Julien Henry)

- LLVM front-end
- uses APRON abstract domains
- path-focusing for SMT-solving
- extra applications to worst case execution time (WCET) analysis

Demo

PAGAI

`http://pagai.forge.imag.fr/` (Julien Henry)

- LLVM front-end
- uses APRON abstract domains
- path-focusing for SMT-solving
- extra applications to worst case execution time (WCET) analysis

Demo

Questions?

http://stator.imag.fr/

European Research Council

Established by the European Commission