Abstract interpretation

David Monniaux

CNRS / VERIMAG

May 23–27, Menlo College

Grenoble

VERIMAG

Joint lab between CNRS and Grenoble University
9 CNRS permanent researchers + 4 research engineers
23 professors

Outline

Static analysis

Establish automatically that a **program** meets a **specification**.

Specification can be:

1. Explicit, e.g. "the program sorts the integer array given as input".

   Can be expressed by e.g. temporal logics, assertions...

2. Implicit, e.g. "the program never crashes due to division by zero, array overflow, bad pointer dereference".

   Easier for the programmer (no need to write anything in addition to the code).

Impossibilities

Turing's Halting Problem / Rice's Theorem

Program analysis is impossible unless one condition is met:

1. Not fully automatic, requires user interaction.
2. Constrained enough class of programs.
3. Finite memory.
4. Finite number of program steps.
5. Analysis can answer **false positives**.
6. Analysis can answer **false negatives**.

User interaction

Example: **interactive theorem proving**.

Program analysis problems generally map to logics (e.g. Peano arithmetic) with no decision procedure.

(Actually a way to prove undecidability of such logics. . .)

Finite memory

Can enumerate **reachable states** explicitly.

Computable but costly: n bits of memory in analyzed system $\Rightarrow 2^n$ states in analyzer

Finite number of program steps

Finite number of program steps

+ program statements with semantics in logics e.g. linear arithmetic

$\Rightarrow$

Bounded model checking.

Analysis can produce false negatives

False negative = some bugs may be ignored

Examples of techniques:

- **testing**
- Coverity

(Semantically sound) static analysis

Deducing **properties** of software

- From a mathematical model of its behaviour (**semantics**).
- Examples: "no division by zero", "no assertion failure"
- valid for **all executions**
- using safe **over-approximation** of behaviors
  - **no false negatives**
  - maybe false positives (**false alarms**)

A central problem

Higher precision (fewer false alarms) vs scaling-up (low higher time/space costs)

Want to have them both?

Outline

Ariane V, maiden flight, 1996

Ariane V self-destructing

Arithmetic overflow

```
x = computation_for_Ariane4()
y = (short int) x;
```

(ok it was Ada, not C)

Arithmetic overflow

```
x = computation_for_Ariane4();
y = (short int) x;
```

(ok it was Ada, not C)

⇒PolySpace Verifier (1996–)
(Deutsch et al.; commercial tool)

Bug found by **direct automated analysis of the source code**.

A modern airplane: Airbus A380

A modern airplane: Airbus A380

⇒Astrée (2002–) (Cousot et al.)

Prove **absence of bugs**.

I was a key member of Astrée (now sold commercially).

Safety-critical embedded systems

- Airplanes (DO-178C), trains, space launchers
- Nuclear plants, electrical grid controls
- Medical devices

US Food and Drug Administration, action on **infusion pumps** (2010).

At Microsoft...

Microsoft Device Driver Verifier (from project SLAM)

CodeContracts

etc.

Outline

Large state spaces

We cannot represent the **concrete state space** X.
Four 32-bit variables: 2^{128} states.
Too large for explicit-state model-checking (need to memorize all states in memory)...
and also for implicit-state model-checking (using clever structures e.g. BDDs)

Solution

Instead of a set of states $s \subseteq X$ use another $s^\sharp$ simpler to represent.

e.g. with $X = \mathbb{Z}^2$, $s \subseteq X$ a set of pairs of integers, $s^\sharp$ a product of 2 intervals

We do not forget behaviors: since $s \subseteq s^\sharp$, cannot forget any reachable state.

Initial states + transitions

Program or machine state = values of variables, registers, memories... within state space Σ.

Examples:

- if system state = 17-bit value, then $\Sigma = \{0,1\}^{17}$;
- = 3 unbounded integers, $\Sigma = \mathbb{Z}^3$;
- if finite automaton, Σ is the set of states;
- if stack automaton, complete state = couple (finite state, stack contents), thus $\Sigma = \Sigma_S \times \Sigma_P^*$.

Transition relation $\rightarrow$ $x \rightarrow y$ = "if at x then can go to y at next time"

Safety properties

Show that a program does not reach an **undesirable state** (crash, error, out of specification). Set W of undesirable states.

Show that there is no $n \geq 0$ and $\sigma_0 \rightarrow \sigma_1 \rightarrow \ldots \sigma_n$ s.t. σ_0 initial state (= reset) and $\sigma_n \in W$

Otherwise said $\sigma_0 \rightarrow^* \sigma_n \in W$. $\rightarrow^*$ **transitive closure** of $\rightarrow$.

Reachable states

$\Sigma_0 \subseteq \Sigma$ set of initial states. **Reachable states** A set of states σ s.t.

$$\exists \sigma_0 \in \Sigma_0 \ \sigma_0 \to^* \sigma \tag{1}$$

Goal: show that $A \cap W = \emptyset$.

Computation

X_n set of states reachable in at most n turns of $\rightarrow$: $X_0 = \Sigma_0$, $X_1 = \Sigma_0 \cup R(\Sigma_0)$, $X_2 = \Sigma_0 \cup R(\Sigma_0) \cup R(R(\Sigma_0))$, etc.

with $R(X) = \{y \in \Sigma \mid \exists x \in X \; x \rightarrow y\}$.

The sequence X_k is ascending for $\subseteq$. Its limit (= the union of all iterates) is the **set of reachable states**.

Iterative computation

Remark $X_{n+1} = \phi(X_n)$ with $\phi(X) = \Sigma_0 \cup R(X)$.

Intuition: to reach in at most $n+1$ turns

- either in 0 turns, thus on an initial state: Σ_0
- either in $0 < k \leq n+1$ coups, otherwise said at most n turns (X_n), then another turn.

How to **compute efficiently** the X_n? And the limit?

Explicit-state model-checking

Explicit representations of X_n (list all states).

If Σ finite, X_n converges in at most $|\Sigma|$ iterations.

Reason:

- Either $X_n = X_{n+1}$, thus remains constant.
- Either $X_n \subsetneq X_{n+1}$, then $X_{n+1} \setminus X_n$ contains at least 1 state. Cannot happen more than $\|\Sigma\|$ times.

Inductive invariants

(Inductive) invariant: set X of states s.t. $\phi(X) \subseteq X$. Recall

$$\phi(X) = X_0 \cup \{y \in \Sigma \mid \exists x \in X \; x \to y\} \tag{2}$$

If X et Y two invariants, then so is $X \cap Y$.

ϕ **monotonic** for $\subseteq$ (if $X \subseteq Y$, then $\phi(X) \subseteq \phi(Y)$).

$\phi(X \cap Y) \subseteq \phi(X) \subseteq X$, same for Y, thus $\phi(X \cap Y) \subseteq X \cap Y$.

Same for intersections of infinitely many invariants.

The strongest invariant

Intersect all invariants, obtain **least invariant** / **strongest invariant**.

This invariant satisfies $\phi(X) = X$, it is the **least fixed point** of ϕ.

Outline

A system with infinite state

State = a single integer variable x

Initial state : $x = 0$
Transition: $x' = x + 1$

Reachable states: $\mathbb{N}$.

Prove that $x \geq 0$ is an invariant.

Cannot compute reachable states by iterations: infinite state space!

A finite state system

State = a single integer variable x

Initial state: $x = 0$
Transition: $x' = x + 1 \wedge x < 10^{10}$

Reachable states: $0 \leq x \leq 10^{10}$

No hope by explicit model-checking techniques (computing the 10^{10} reachable states).

Abstraction

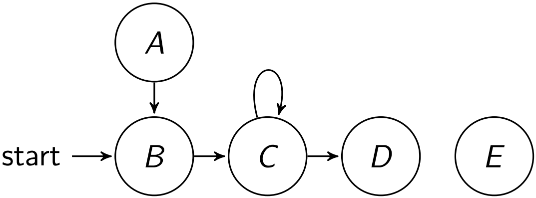
Introduce 5 "abstract states"

- A: $x < 0$
- B: $x = 0$
- C: $0 < x < 10^{10}$
- D: $x = 10^{10}$
- E: $x > 10^{10}$

Put an arrow between abstract states P and Q iff one can move from $p \in P$ to $q \in Q$.

Example: can move from A to B because $\{x = -1\} \in A$, can move to $\{x' = 0\} \in B$.

Resulting system



- A: $x < 0$
- B: $x = 0$
- C: $0 < x < 10^{10}$
- D: $x = 10^{10}$
- E: $x > 10^{10}$

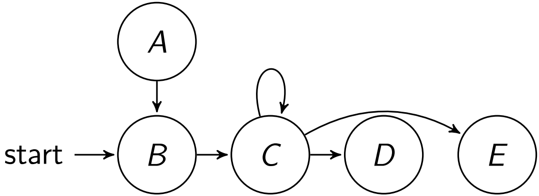
No concrete transition is forgotten and thus E is **unreachable**.

Other example

Initial state: $x = 0$ Transition: $x' = x + 2 \wedge x \neq 10^{10}$

Reachable states: $0 \leq x < 10^{10} \wedge x \bmod 2 = 0$.

Abstract graph



- A: $x < 0$
- B: $x = 0$
- C: $0 < x < 10^{10}$
- D: $x = 10^{10}$
- E: $x > 10^{10}$

$C \rightarrow E$ since $(10^{10} - 1) \rightarrow (10^{10} + 1)$.

Over-approximation

More behaviors:

- E is concretely reachable.
- E is abstractly reachable

The analysis fails to prove the true property "E unreachable".
Incomplete method.

Remark: works with a better abstraction ($x < 10^{10} - 1$).

Principles of predicate abstraction

- A finite set of **predicates** (e.g. arithmetic constraints).
- Construct a **finite** system of abstract transitions between abstract states.
- Each abstract state labeled by predicates, e.g. ex. $x < 0$.
- Put an abstract transition from A to B iff one can move from a state $a \in A$ to a state $b \in B$.
- **Correctness** if an abstract state is unreachable, then so are the corresponding concrete states

How to construct the abstract system

Abstract states $A : x < 0$ and $C : 0 < x < 10^{10}$, transition relation $x' = x + 1 \wedge x < 10^{10}$, can we move from A to C?

Otherwise said: is there a solution to $x < 0 \wedge (x' = x + 1 \wedge x < 10^{10}) \wedge x' > 0$?

Use **satisfiability modulo theory** (SMT-solving).

Computing the graph

- Abstract states are couples (program point, set of predicates)
- Apply SMT-solving to insert or not insert arrows.
- Check if **bad states** are unreachable.
- If they are, **win!**

...and if they are reachable?

- Maybe the abstraction is badly chosen?
- Maybe the property to prove (unreachability of bad states) is false?

Outline

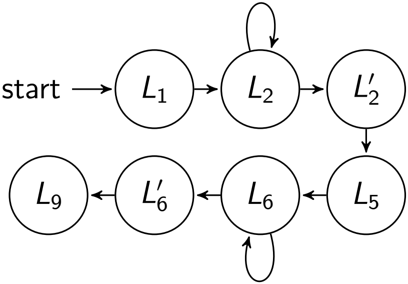
Exemple

```
x = 0;
while (x < 10) {
   x = x+1;
}
y = 0;
while (y < x) {
   y = y+1;
}
```

Try predicates $x < 0$, $x = 0$, $x > 0$, $x < 10$, $x = 10$, $x > 10$, $y < 0$, $y = 0$, $y > 0$, $y < x$, $y = x$, $y > x$.

Note: 12 predicates, so in the worst case $2^{12} = 4096$ combinations, some of which impossible (cannot have both $x < 0$ and $x > 0$ at same time).

Abstract automaton




```
1  x = 0;
2  while (x<10)
3     x = x+1;
4  }
5  y = 0;
6  while (y<x) {
7     y = y+1;
8  }
```

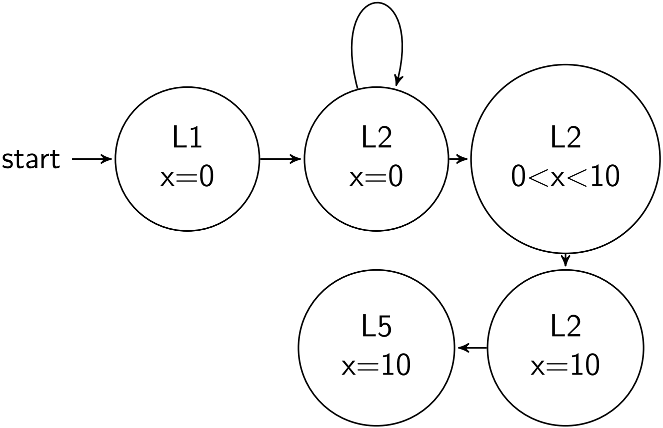
L_1: line 1, $x = 0$
L_2: line 2, $0 < x < 10$
L_2': line 2: $x = 10$
L_5: line 5: $x = 10$
L_6: line 6: $x = 10 \wedge y < x$
L_6': line 6: $x = 10 \wedge y = x$
L_9: line 9: $x = 10 \wedge y = x$

Attention

```
x = 0;
while (x != 10) {
   x = x+2;
}
```

Syntactic choice of predicates ($x < 0$, $x = 0$, $x > 0$, $x < 10$, $x = 10$, $x > 10$).

Some solution?



Why is this solution wrong?

This solution is **sound** since it collects all behaviors of the program.

But you realize this only because you already know (in your head) the set of reachable states! (This is cheating.)

This solution is not **inductive**: it is possible to move from a state represented in the graph to one that isn't!

Attention

```
1 x = 0;
2 while (x != 10) {
3   x = x+2;
4 }
```

At line 2, abstraction says $0 < x < 10$, thus $x = 9$ for instance.

$x = 9$ is inaccessible *in the concrete systems*! You know it only because you computed the set of reachable states $\{0, 2, 4, 6, 8\}$.

Need a transition from $0 < x < 10$ $(x = 9)$ to a new state $x > 10$ $(x = 11)$.

Human intuition vs automated computation

The human sees the simple program and computes the set of reachable states $\{0, 2, 4, 6, 8\}$ knowing x should be even.

Then projects onto predicates, and $x > 10$ unreachable.

Automated computation does not see that x is even because it was not given the predicate $x \bmod 2 = 0$.

Not convinced?

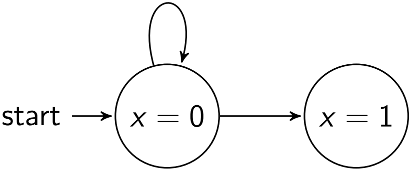
Let P be a program where Boolean x is not mentioned. Consider:
$x := 0;\ P;\ x := 1$

Use predicates $x = 0$ et $x = 1$. Give a finite automaton for the behaviors of the program wrt x...

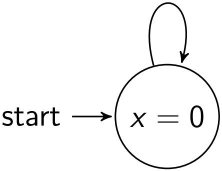
Automaton with two states $x = 0$, $x = 1$. Simple, hey?

A minimal automaton (not inductive)

If P **terminates**:

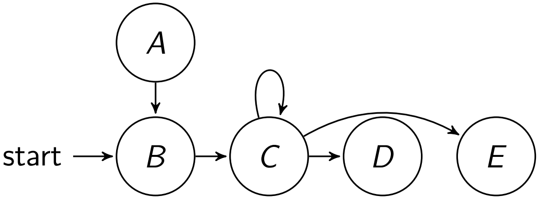


If P **does not terminate**:



Outline

Abstraction refinement



$x' = x + 2 \wedge x \neq 10^{10}$

E is reachable in the abstract and not in the concrete.
Would have been prevented using predicate $x < 10^{10} - 1$. Can this be made automatic?

Yes: compute weakest precondition $wp(\neg E)$ for one step:
$x \leq 10^{10} \wedge x + 2 \leq 10^{10} \equiv x \leq 10^{10} - 2$.
Add $x \leq 10^{10} - 2$ as predicate and voil.

CEGAR

Counterexample guided abstraction refinement

Generalize the idea compute weakest precondition and add predicates .

Some tools

- Bounded model checking on C programs: CBMC
- Predicate abstraction on C programs: Microsoft Device Driver Verifier [SLAM], BLAST
- SMT-solvers: Yices (SRI), Z3 (Microsoft)

Recall the idea

Try to compute an interval for each variable at each program point using **interval arithmetic** :

```
assume(x >= 0 && x<= 1);
assume(y >= 2 && y= 3);
assume(z >= 3 && z= 4);
t = (x+y) * z;
```

Interval for z?

Recall the idea

Try to compute an interval for each variable at each program point using **interval arithmetic** :

```
assume(x >= 0 && x<= 1);
assume(y >= 2 && y= 3);
assume(z >= 3 && z= 4);
t = (x+y) * z;
```

Interval for z? $[6, 16]$

Why is this interesting?

Let $t(0..10)$ an array.
Program writes to $t(i)$.

We must know whether $0 \leq i \leq 10$, thus know an **interval** over i.

Again...

```
assume(x >= 0 && x <= 1);
y = x;
z = x-y;
```

- The human (intelligent) sees $z = 0$ thus interval $[0, 0]$, taking into account $y = x$.
- Interval arithmetic does not see $z = 0$ because it does not take $y = x$ into account.

How to track relations

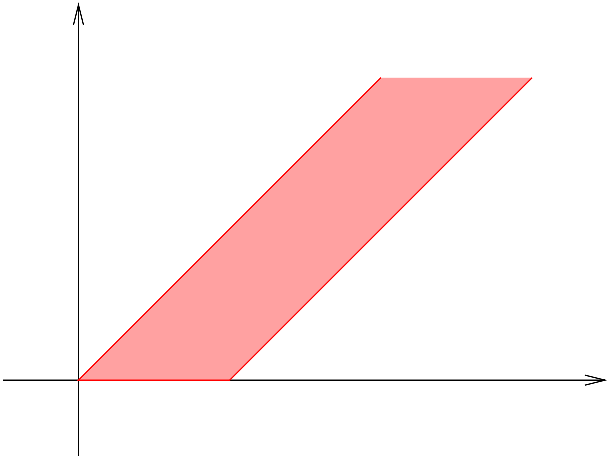
Using **relational domains**.

E.g.: keep

- for each variable an interval
- for each pair of variables (x, y) an information $x - y \leq C$.
- (One obtains $x = y$ by $x - y \leq 0$ and $y - x \leq 0$.)

How to **compute** on that?

Bounds on differences



Practical example

Suppose $x - y \leq 4$, computation is $z = x + 3$, then we know $z - y \leq 7$.

Suppose $x - z \leq 20$, that $x - y \leq 4$ and that $y - z \leq 6$, then we know $x - z \leq 10$.

We know how to **compute** on these relations (transitive closure / shortest path).
On our example, obtain $z = 0$.

Why this is useful

Let $t(0..\mathbf{n})$ an array in the program.
The program writes $t(i)$.

Need to know whether $0 \leq i \leq n$, otherwise said find bounds on i and on $n - i$...

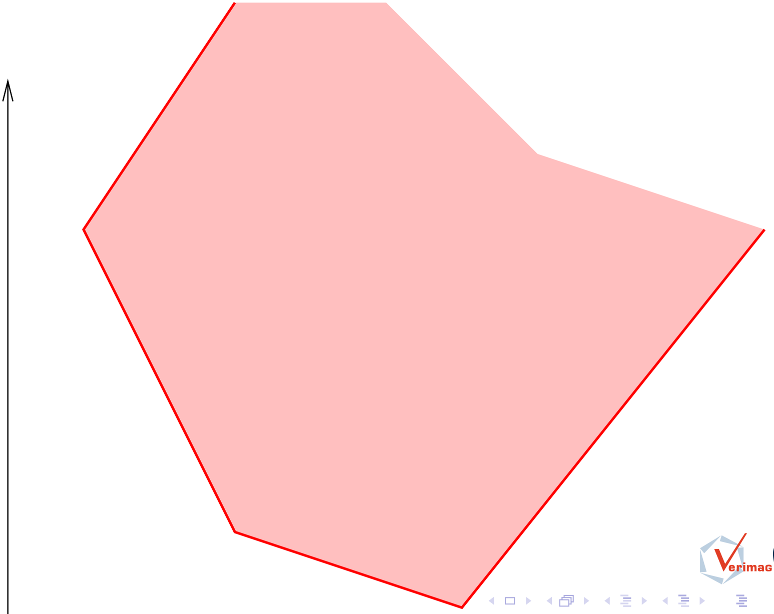
Can we do better?

How about tracking relations such as $2x + 3y \leq 6$?

At a given program point, a set of **linear inequalities**.

In other words, a **convex polyhedron**.

Example of polyhedron



Caveat

(In general) The more precise we are, the higher the costs.
For each line of code:

- Intervals: algorithms $O(n)$, n number of variables.
- Differences $x - y \leq C$: algorithms $O(n^3)$
- Polyhedra: algorithms often $O(2^n)$.

On short examples with few variables, ok... But in general?

Even linear may not be fast enough

Fly-by-wire control code from Airbus:

- Main control loop
- Number of tests linear in length n of code
- Number of variables linear in length n of code (global state)
- Complexity of naive convex hull on products of intervals linear in number of variables

Even linear may not be fast enough

Fly-by-wire control code from Airbus:

- Main control loop
- Number of tests linear in length n of code
- Number of variables linear in length n of code (global state)
- Complexity of naive convex hull on products of intervals linear in number of variables

$\Rightarrow$Cost per iteration in n^2

Absolute value

```
y = abs(x);   /* valeur absolue */
if (y >= 1) {
  assert(x != 0);
}
```

Interval expansion

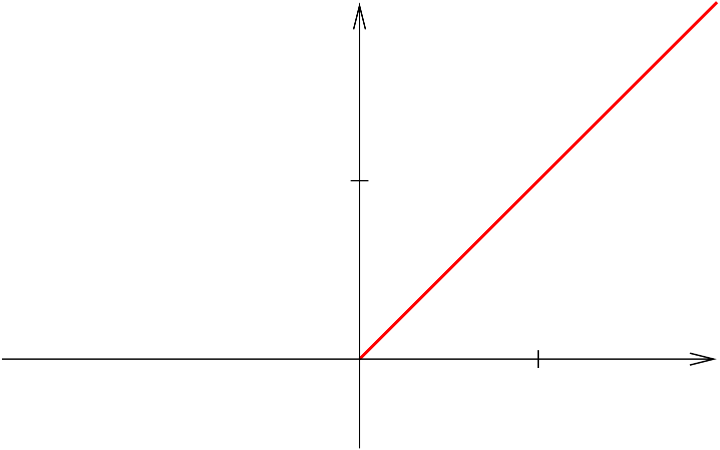
Intervals:

```
/* -1000 <= x <= 2000 */
if (x < 0) y = -x; /* 0 <= y <= 1000 */
else y = x; /* 0 <= y <= 2000 */

if (y >= 1) { /* 1 <= y <= 2000 */
  assert(x != 0); /* -1000 <= x <= 2000 !!! */
}
```

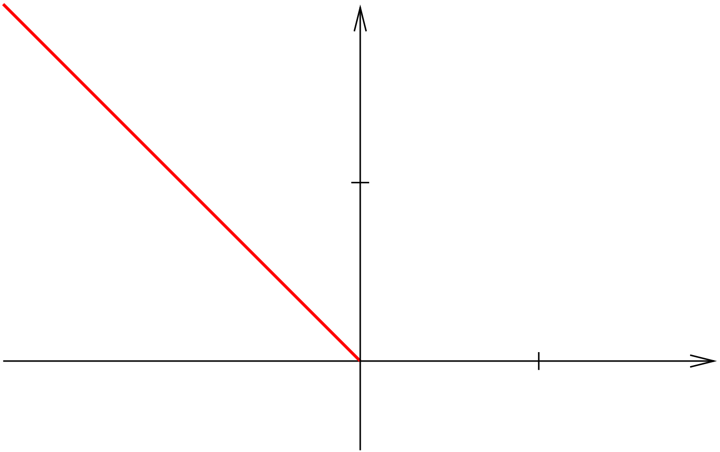
Polyhedra

Branch $x \geq 0$



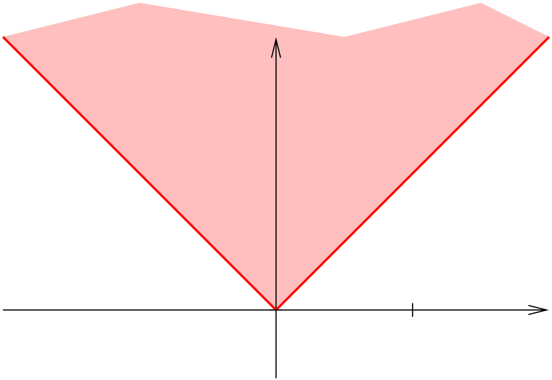
Autre branche du test

Branch $x < 0$



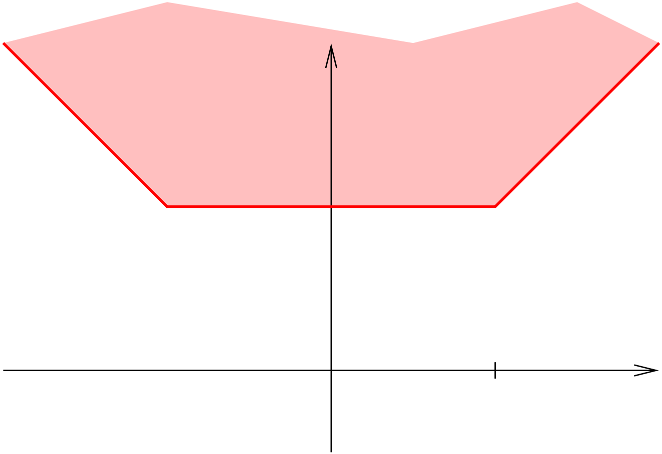
After first test

$y = |x|$ = union of the two red lines. Not a convex.
Convex hull = pink polyhedron



At second test

Note: includes $(x, y) = (0, 1)$.



Disjunction

Possible if we do a union of two polyhedra:

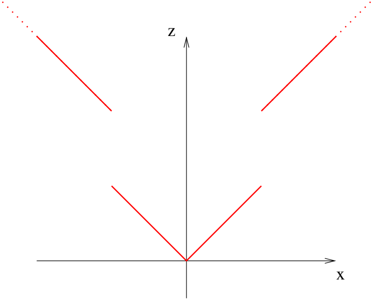
- $x \geq 0 \wedge y = x$
- $x < 0 \wedge y = -x$

But with *n* tests?

Two tests

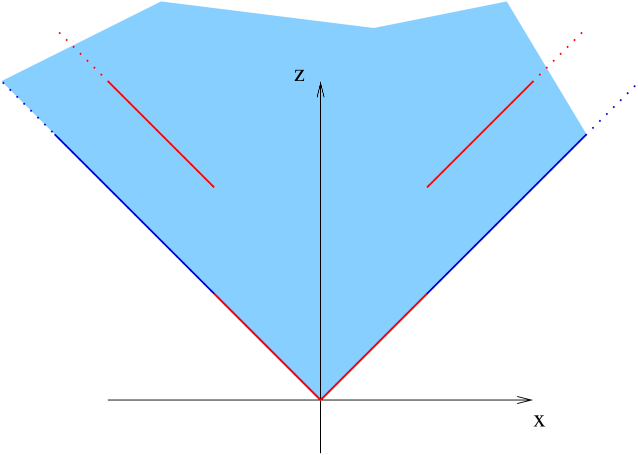
```
if (x >= 0) y=x; else y= -x;
if (y >= 1) z=y+1; else z=y;
```

4 polyhedra = costly computations



Two tests, convex hull

More imprecise:



Sources of imprecision

- Need to distinguish **each path** and compute one polyhedron for each.
- But 2^n paths.
- **Too costly** if done naively.
- In current tools, not implemented.
- $\Rightarrow$explains some imprecisions.

Current research

In the last few years articles propose methods distinguishing paths.

Use of SMT-solving techniques to cut the exponential cost:

Only look at "useful" paths.

Loops?

Push intervals / polyhedra forward…

```
int x=0;
while (x<1000) {
  x=x+1;
}
```

Loop iterations $[0, 0]$, $[0, 1]$, $[0, 2]$, $[0, 3]$,…

How? $\phi(X) = \text{tat initial} \sqcup \text{post}(X)$, thus
$\phi([a, b]) = \{0\} \sqcup [a + 1, \min(b, 999) + 1]$

When do we stop? Wait 1000 iterations? No.

One solution...

Extrapolation!

$[0,0]$, $[0,1]$, $[0,2]$, $[0,3] \to [0,+\infty)$

Push interval:

```
int x=0; /* [0, 0] */
while /* [0, +infty) (x<1000) {
  /* [0, 999] */
  x=x+1;
  /* [1, 1000] */
}
```

Yes! $[0,\infty[$ is stable!

Mediocre results

Expected: $[0, 999]$.
Obtained $[0, +\infty)$.

Mediocre results

Expected: $[0, 999]$.
Obtained $[0, +\infty)$.

Run one more iteration of the loop:

```
[0, +infty) (x<1000)
   /* [0, 999] */
   x=x+1;
   /* [1, 1000] */
```

Obtain $\{0\} \sqcup [1, 1000] = [0, 1000]$.

Narrowing

```
int x=0; /* [0, 0] */
while /* [0,1000] (x<1000) {
  /* [0, 999] */
  x=x+1;
  /* [1, 1000] */
}
```

Yes! $[0, 1000]$ is an inductive invariant!

Stabilization

Look for a set (polyhedron, intervals)

- Containing initial values for the loop.
- **Inductive**: if valid at one iteration, valid at the next.

Look for X such that $\phi(X) \subseteq X$ with $\phi(X) = \text{tats initiaux} \cup \text{post}(X)$
$\text{post}(X)$ = states reachable from X in one loop iteration

Any inductive invariant. (Not necessarily the least one.)

Computing the inductive invariant

We don't know how to compute $\text{post}(P)$ with P interval / polyhedron in general.
(The loop body may be complex, with tests...)

Replace computation by simpler over-approximation $\text{post}(X) \subseteq \text{post}^\sharp(X)$.

Cannot do $\cup$ over polyhedra, do $\sqcup$ (convex hull)
Thus computation: $\phi^\sharp(X) = \text{initial states} \sqcup \text{post}^\sharp(X)$
Instead of $\phi(X) \subseteq X$ with $\phi(X) = \text{initial states} \cup \text{post}(X)$

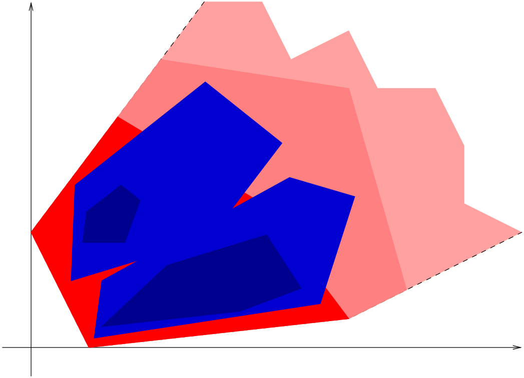
All the time, over-approximation

$\phi(X) \subseteq \phi^\sharp(X)$ so $\operatorname{lfp} \phi \subseteq \operatorname{lfp} \phi^\sharp$
(work out the math, using $\operatorname{lfp} \psi = \inf\{X \mid \psi(X) \subseteq X\}$)

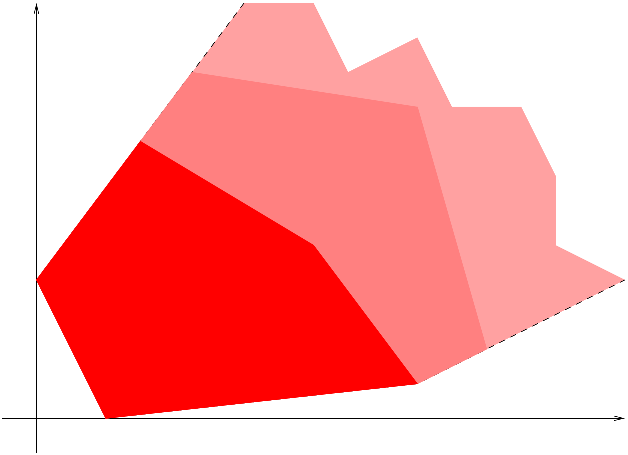
In the end, **over-approximation** of the least fixed point of ϕ.

Graphical vision

Dark blue = concrete reachable states after ≤ 1 loop iteration
Light blue = concrete reachable states after ≤ 2 loop iterations
Dark red = over-approximated states after ≤ 1 loop iteration
Light red = over-approximated states after ≤ 2 loop iterations



Extrapolation



Consequences

- Over-approximate during computations (even without loops).
- Over-approximation during widening.
- Thus obtain **super-**set of reachable states.
- This super-set is an **inductive invariant** (cannot exit from it).

Practical consequences

- Cannot prove that a problem truly happens.
  Example: interval $i \in [0, 20]$ for access $t(0..10)$, is the interval exact?
- Yet sure that all potential problems are detected (over-approximation of problems).
- Let B be the set of bad states. $X^\sharp \cap B \neq \emptyset$: "ORANGE"
- If $X^\sharp \subseteq B$, "RED".
- What do orange vs red mean?

Outside of numerics

Pointers, arrays, memory threads. . .

E.g. representing tree / graphs using automata
Widening = limitation in the number of states when computing bisimulation (Myhill-Nerode minimization of DFA)

Important points

- The computer is stupid, it does not "see" why a program works.
- Normal, everything important is **undecidable algorithmically** (or of **high complexity**).
- Look for inductive invariants that can be **proved automatically** (e.g. by propagation of intervals or polyhedra).
- They over-approximate the reachable states, thus the safety violations.

Success stories

- **Microsoft SLAM** / Device driver verifier — predicate abstraction, checks the respect of Windows API in device drivers
- **PolySpace Verifier**
- **Astre**, with specific control numerical relations — A340, A380 (Airbus), ATV (EADS Astrium / ESA), etc.
- **Absint**, worst case execution time (WCET) with cache and pipelines