

# Abstract Interpretation

SAVE 2016,  
Changsha, 10 December 2016

**Patrick Cousot**  
[pcousot@cs.nyu.edu](mailto:pcousot@cs.nyu.edu) [cs.nyu.edu/~pcousot](http://cs.nyu.edu/~pcousot)

# This is an abstract interpretation

**Colloquium d'Informatique de l'UPMC Sorbonne Universités**

**Abstract interpretation**

**Patrick Cousot**  
*New York University*

**Amphi 15**

4, place Jussieu  
75005 Paris  
Metro Jussieu

**29 Septembre 2016 à 18h00**

The complexity of large programs grows faster than the intellectual ability of programmers in charge of their development and maintenance. The direct consequence is a lot of errors and bugs in programs mostly entangled by their end-users. Programmers are not responsible for these bugs. They are not required to produce provably safe and secure programs. This is because programmers are only required to copy state-of-the-art techniques, that is trading on safety many cases. This state of the art is changing rapidly and so will irresponsibility, as in other manufacturing chapters.

Scalable and cost-effective tools have appeared recently that can avoid bugs with possible dramatic consequences for example in transportation, safety, privacy of social networks, etc. Entirely automatic, they are able to capture all bugs involving the violation of software healthiness rules such as the use of operations with arguments for which they are undefined.

These tools are formally founded on abstract interpretation. They are based on a definition of the semantics of programming languages specifying all possible executions of the programs of a language. Program properties of interest are abstractions of these semantics abstracting away all aspects of the semantics not relevant for a particular response or program. This yields good methods.

Full automation is more difficult because of undecidability: programs cannot always prove programs correct in finite time and memory. Further abstractions are therefore necessary for automation, which introduce approximations. Bugs may be signalled that are impossible in any execution but not more significant. This has an economic cost, much like this testing. Moreover, the best static analysis tools are able to reduce these false alarms to almost zero. A time-consuming and error-prone task which is too difficult, if not impossible for programmers, without tools.

Patrick Cousot received the Doctor Engineer degree in Computer Science and the Doctor in Sciences degree in Mathematics from the University Joseph Fourier of Grenoble (France). He was a Research Scientist at the French National Center for Scientific Research at the University Joseph Fourier of Grenoble, France, then professor at the University of Metz, the Ecole Polytechnique, the Ecole Normale Supérieure, Paris, France. He is Silver Professor of Computer Science at the Courant Institute of Mathematical Sciences, New York University, USA. Patrick Cousot is the inventor, with Radhia Cousot, of Abstract Interpretation.

contact : [colloquium@lip.fr](mailto:colloquium@lip.fr)  
<http://www.lip.fr/colloquium/>  
Video disponible sur le site



# Scientific research

# Scientific research

- In **Mathematics/Physics**:  
trend towards **unification** and **synthesis** through **universal principles**
  - In **Computer science**:  
trend towards **dispersion** and **parcellation** through a ever-growing collection of **local ad-hoc techniques** for **specific applications**
- An exponential process, will stop!**

## Example: reasoning on computational structures

WCET  
Axiomatic semantics  
Confidentiality analysis  
Program synthesis  
Grammar analysis  
Statistical model-checking  
Invariance proof  
Probabilistic verification  
Parsing

Security protocols verification  
Dataflow analysis  
Partial evaluation  
Effect systems  
Trace semantics  
Symbolic execution  
Quantum entanglement detection  
Type theory

Systems biology analysis  
Model checking  
Obfuscation  
Denotational semantics  
Theories combination  
Code contracts  
Quantum entanglement detection  
SMT solvers  
Steganography

Database query  
Dependence analysis  
CEGAR  
Program transformation  
Interpolants  
Integrity analysis  
Bisimulation  
SMT solvers  
Tautology testers

Operational semantics  
Abstraction refinement  
Type inference  
Separation logic  
Termination proof  
Shape analysis  
Malware detection  
Code refactoring

## Example: reasoning on computational structures

WCET  
Axiomatic semantics  
Confidentiality analysis  
Program synthesis  
Grammar analysis  
Statistical model-checking  
Invariance proof  
Probabilistic verification  
Parsing

Security protocols verification  
Dataflow analysis  
Partial evaluation  
Effect systems  
Trace semantics  
Symbolic execution  
Quantum entanglement detection  
Type theory

Systems biology analysis  
Model checking  
Obfuscation  
Denotational semantics  
Theories combination  
Code contracts  
Quantum entanglement detection  
SMT solvers  
Steganography

Database query  
Dependence analysis  
CEGAR  
Program transformation  
Interpolants  
Integrity analysis  
Bisimulation  
SMT solvers  
Tautology testers

Operational semantics  
Abstraction refinement  
Type inference  
Separation logic  
Termination proof  
Shape analysis  
Malware detection  
Code refactoring

## Example: reasoning on computational structures

### Abstract interpretation

WCET  
Axiomatic semantics  
Confidentiality analysis  
Program synthesis  
Grammar analysis  
Statistical model-checking  
Invariance proof  
Probabilistic verification  
Parsing

Security protocols verification  
Dataflow analysis  
Partial evaluation  
Effect systems  
Trace semantics  
Symbolic execution  
Quantum entanglement detection  
Type theory

Systems biology analysis  
Model checking  
Obfuscation  
Denotational semantics  
Theories combination  
Code contracts  
Quantum entanglement detection  
SMT solvers  
Steganography

Database query  
Dependence analysis  
CEGAR  
Program transformation  
Interpolants  
Integrity analysis  
Bisimulation  
SMT solvers  
Tautology testers

Operational semantics  
Abstraction refinement  
Type inference  
Separation logic  
Termination proof  
Shape analysis  
Malware detection  
Code refactoring

# Intuition I

# Concrete

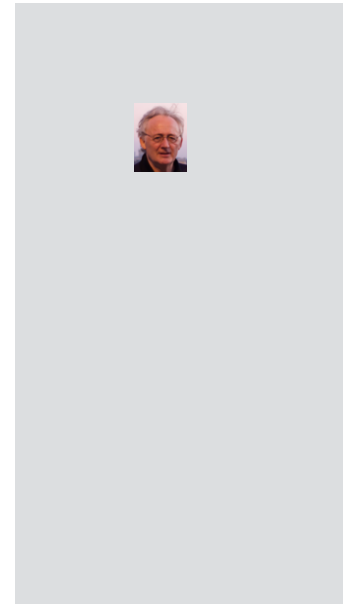


Abstract interpretation, SAVE 16, Changsha, 10 December 2016

9

© P.Cousot

# Abstraction 1

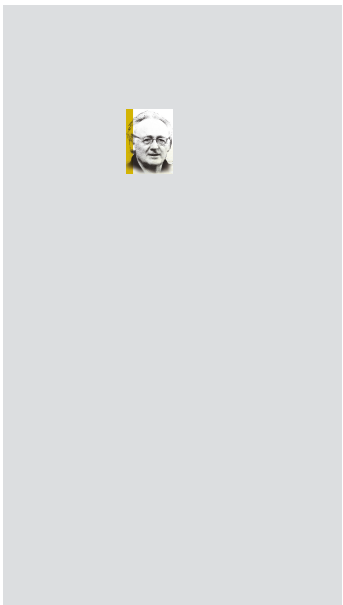


Abstract interpretation, SAVE 16, Changsha, 10 December 2016

10

© P.Cousot

# Abstraction 2

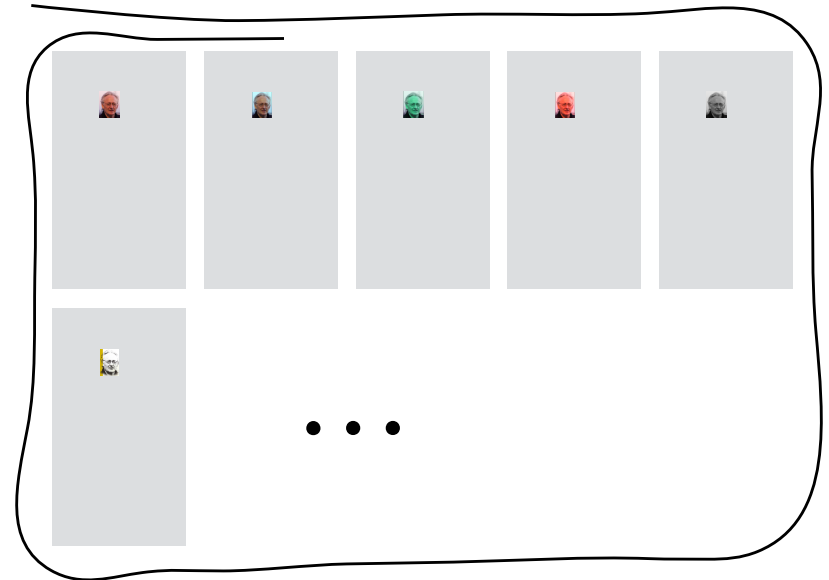


Abstract interpretation, SAVE 16, Changsha, 10 December 2016

11

© P.Cousot

# Concretization 2

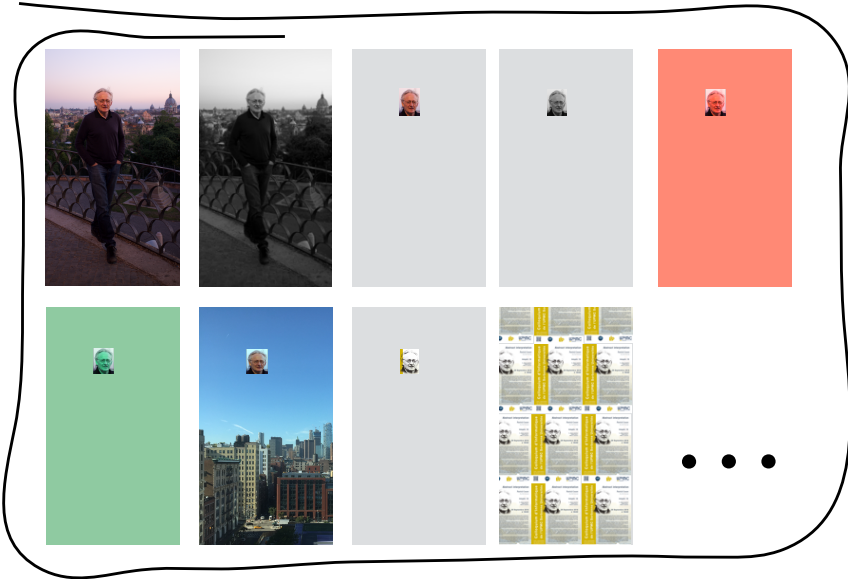


Abstract interpretation, SAVE 16, Changsha, 10 December 2016

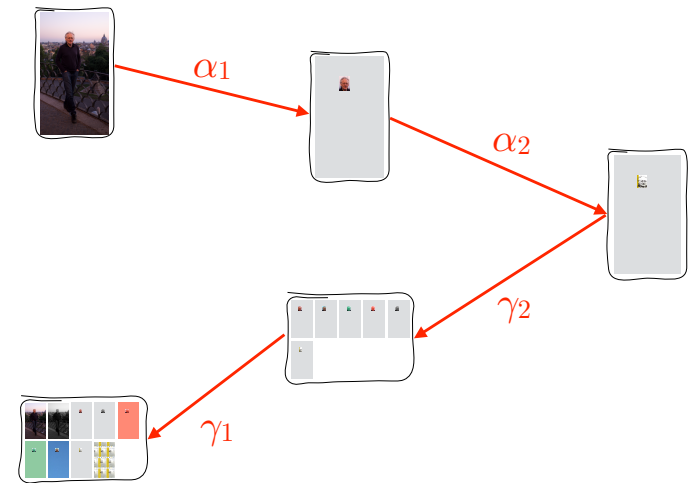
12

© P.Cousot

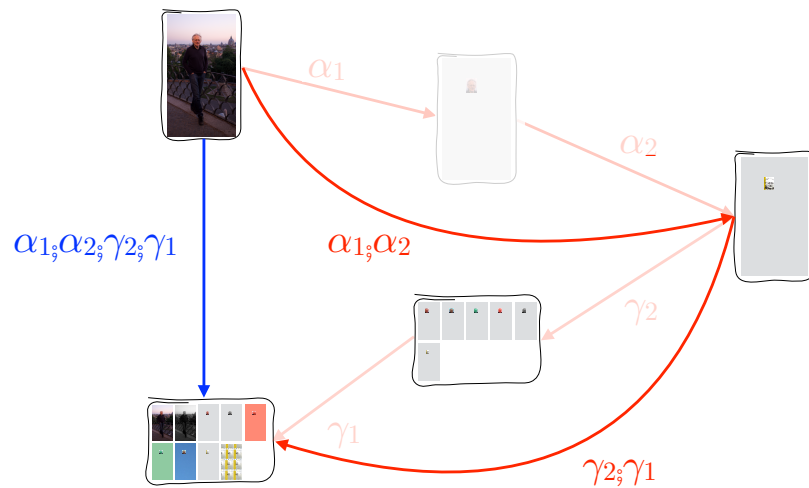
# Concretization I



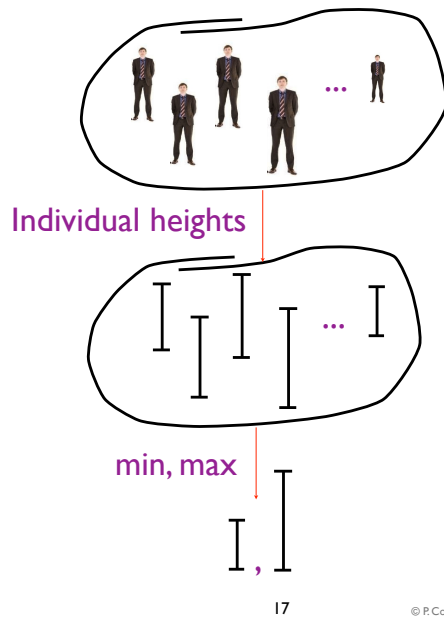
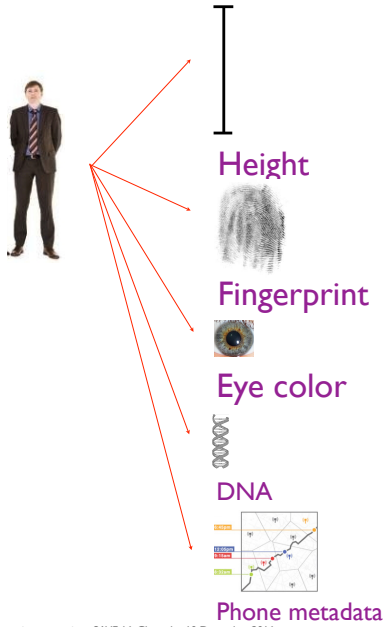
# Abstract interpretations



# Abstract interpretations

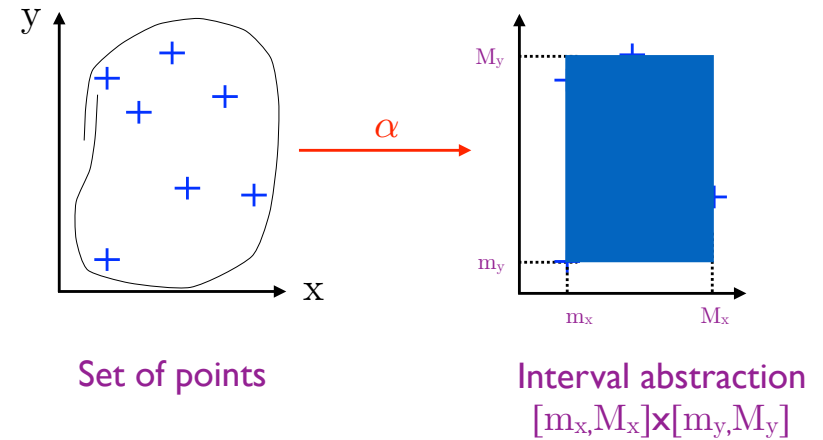


# Intuition 2



## Interval abstraction

- Example: interval abstraction (also called *box abstraction*)



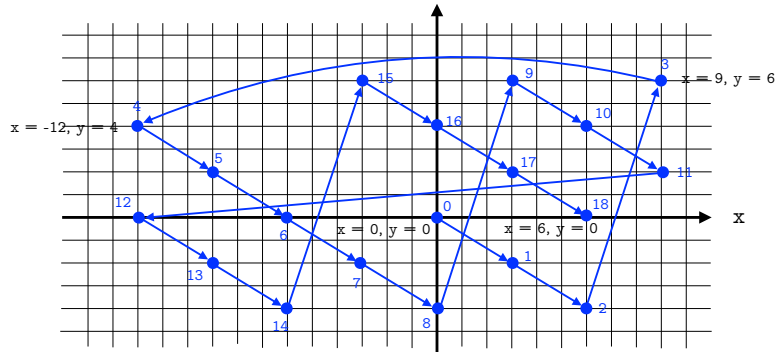
## Intuition 3

## A C program and one of its executions

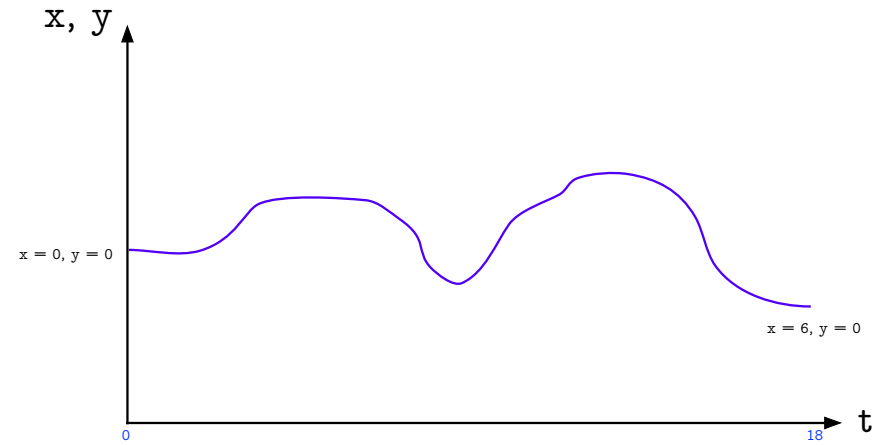
```
#include <stdio.h>
int main()
{
    int x, y;
    printf("Enter two integers: ");
    scanf("%d %d", &x, &y);
    /* 1: */ while ((x != 6) || (y != 0)) {
        printf("x = %d, y = %d\n", x, y);
    /* 2: */ x = x + 3;
    /* 3: */ if (x > 10) x = -x;
    /* 4: */ y = y - 2;
    /* 5: */ if (y < -5) y = -y;
    }
    /* 6: */ printf("x = %d, y = %d\n", x, y);
}
```

```
Enter two integers: x = 0, y = 0
x = 3, y = -2
x = 6, y = -4
x = 9, y = 6
x = -12, y = 4
x = -9, y = 2
x = -6, y = 0
x = -3, y = -2
x = 0, y = -4
x = 3, y = 6
x = 6, y = 4
x = 9, y = 2
x = -12, y = 0
x = -9, y = -2
x = -6, y = -4
x = -3, y = 6
x = 0, y = 4
x = 3, y = 2
x = 6, y = 0
```

## Graphical representation of the execution (1)

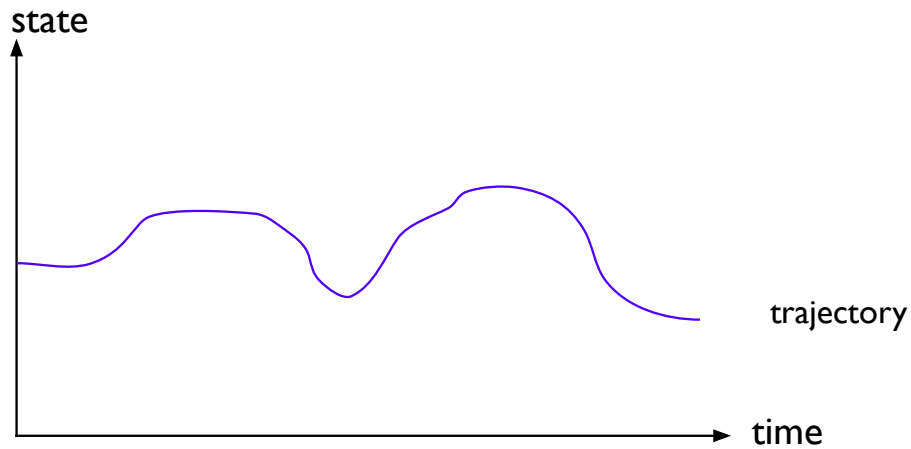


## Graphical representation of the execution (2)



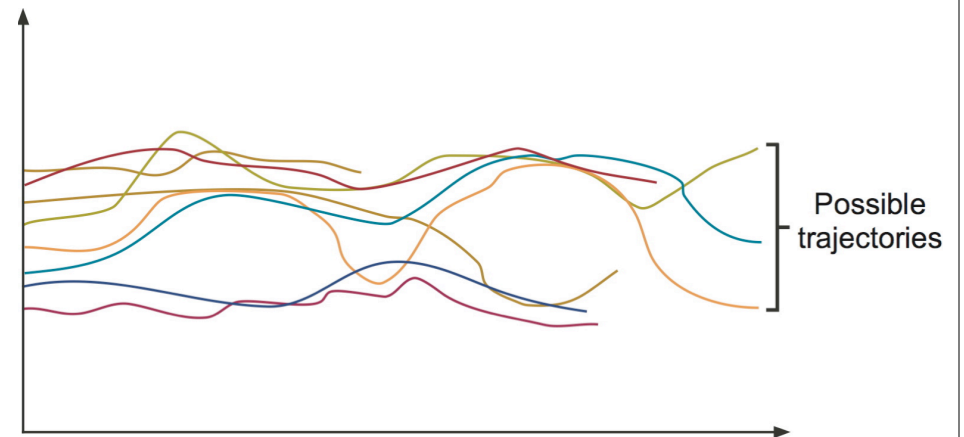
## Semantics

*Formalize what it means to run a program*



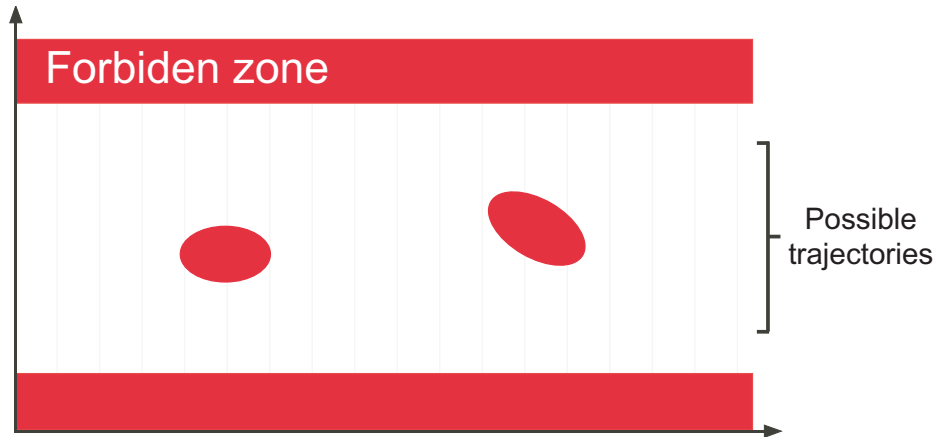
## Properties (Collecting semantics)

*Formalize what you are interested to know about program behaviors*



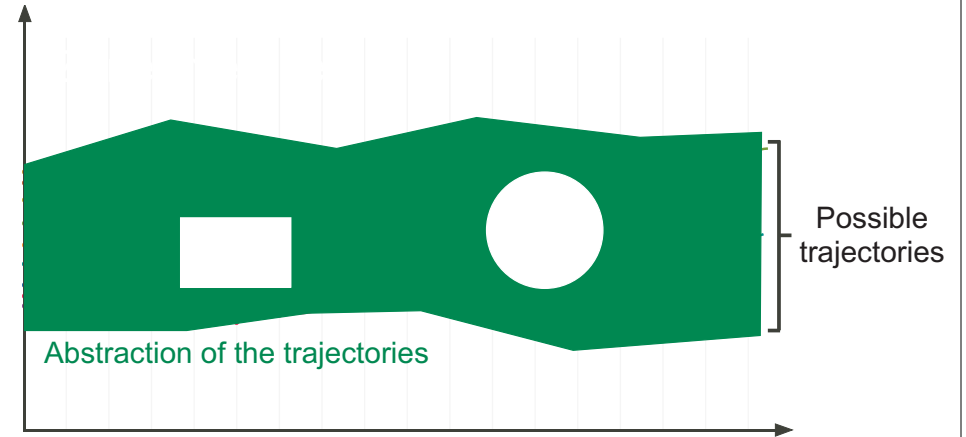
## Specification

Formalize what you are interested to **prove** about program behaviors



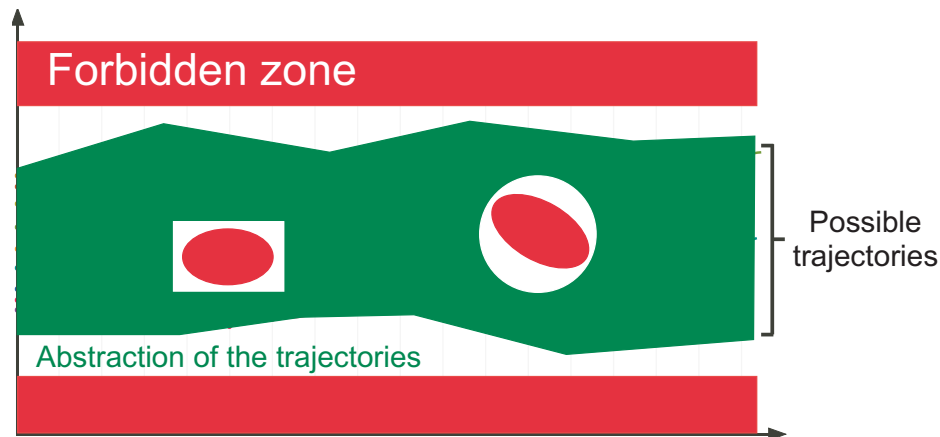
## Abstraction

**Abstract** away all information on program behaviors irrelevant to the proof



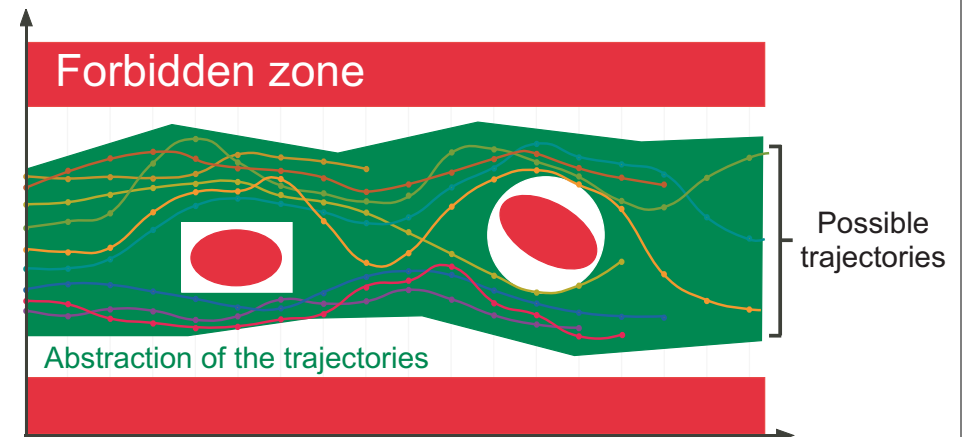
## Verification

The proof is fully **automatic**



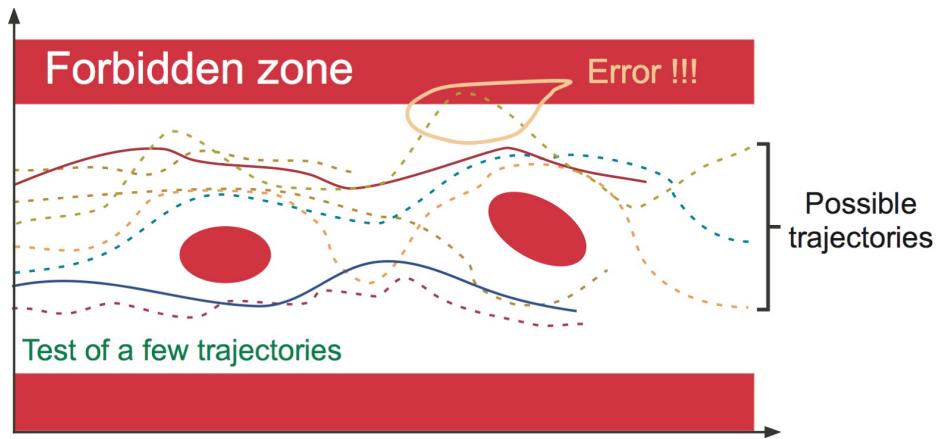
## Soundness

Never forget any possible case so the **abstract proof is correct in the concrete**



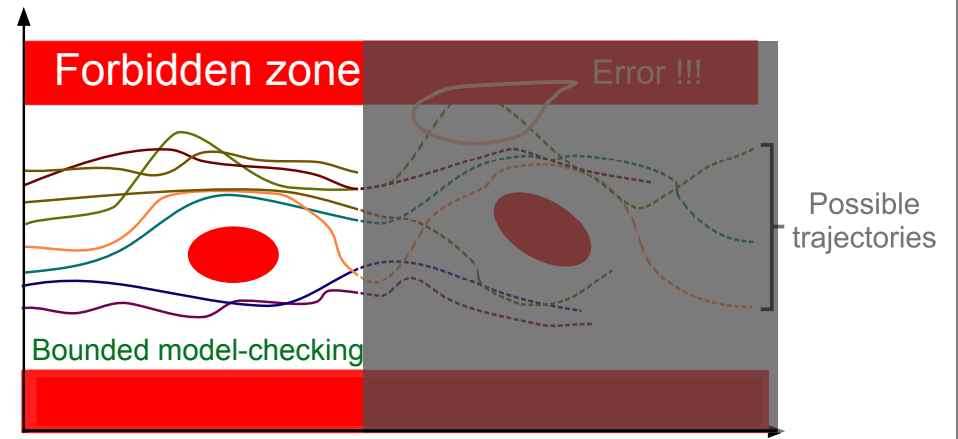
# Unsound methods: testing

Try a few cases



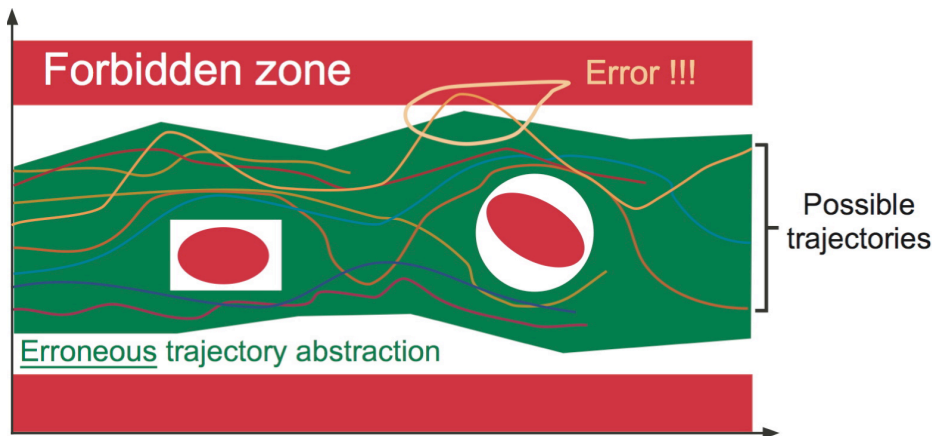
# Unsound methods: bounded model checking

Simulate the beginning of all executions (so called bounded model-checking)



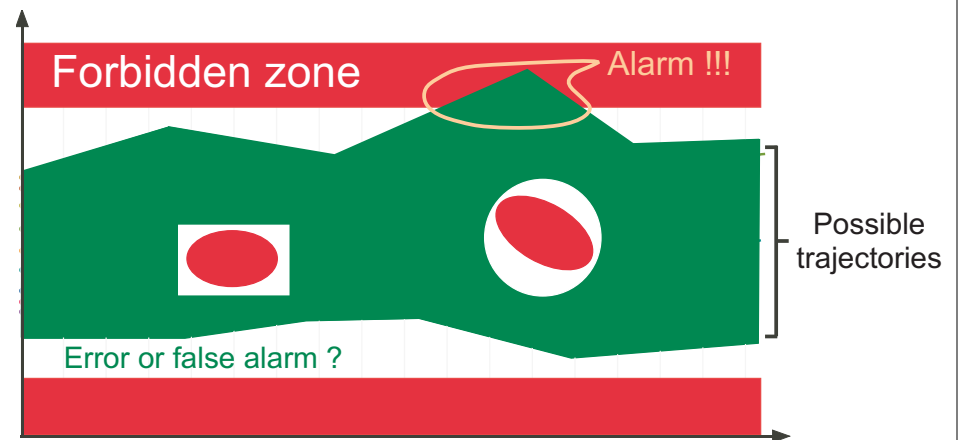
# Unsound methods: soundness

Many static analysis tools are *unsound* (e.g. Coverity, etc.) so inconclusive



# Alarms

When abstract proofs may fail while concrete proofs would succeed

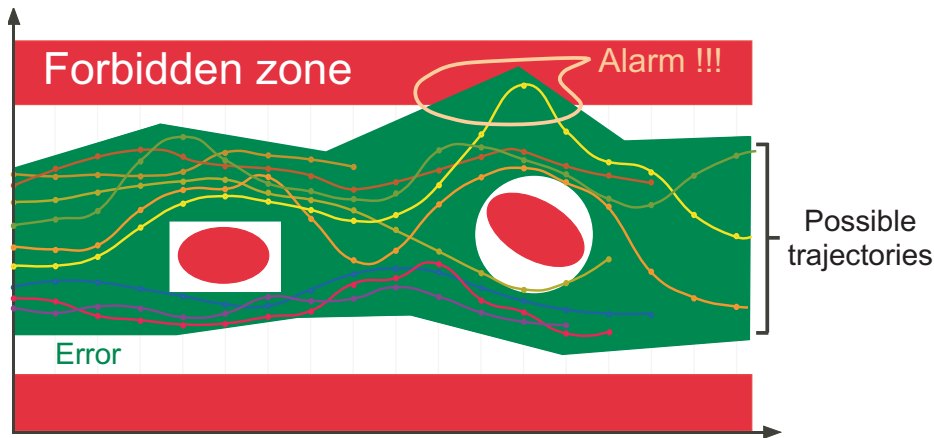


By soundness an alarm must be raised for this over-approximation!



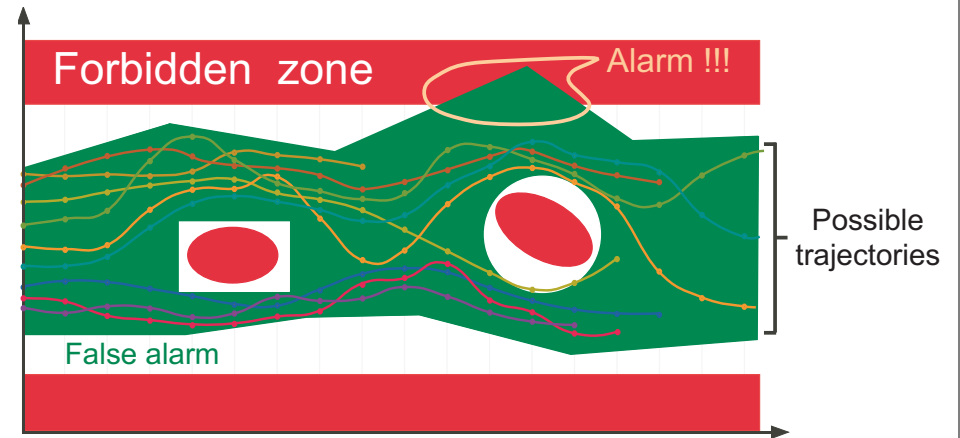
## True alarm

The abstract alarm may correspond to a concrete error



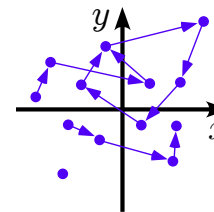
## False alarm

The abstract alarm may correspond to no concrete error (false negative)

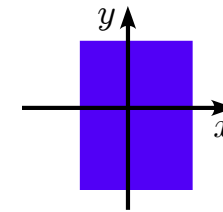


## What to do in presence of false alarms

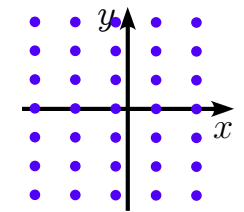
- False alarms are ultimately unavoidable (Gödel's incompleteness)
- Consider **finite** cases or **decidable cases** only (model-checking, *does not scale*)
- Ask for **human help** by providing information on the program behavior (theorem provers, SMT solvers), *program specific and labor costly*
- Have specialists **refine the abstract interpretation** (e.g. Astrée, <http://www.absint.com/astree/index.htm>), *shared cost*



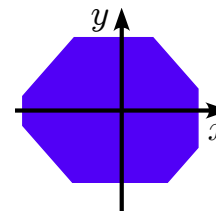
Collecting semantics:  
partial traces



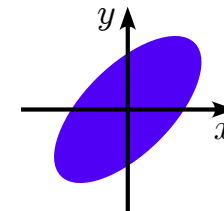
Intervals:  
 $x \in [a, b]$



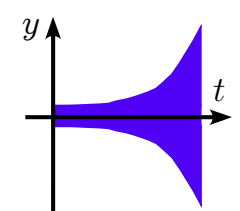
Simple congruences:  
 $x \equiv a[b]$



Octagons:  
 $\pm x \pm y \leq a$



Ellipses:  
 $x^2 + by^2 - axy \leq d$

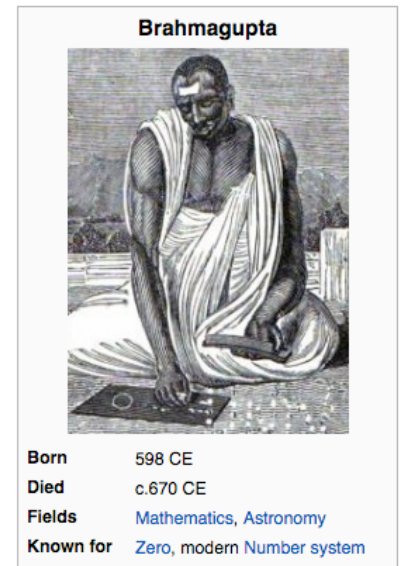


Exponentials:  
 $-a^{bt} \leq y(t) \leq a^{bt}$

# The very first static analysis

## Brahmagupta

**Brahmagupta** (Sanskrit: ब्रह्मगुप्त; (598–c.670 CE) was an Indian mathematician and astronomer who wrote two important works on Mathematics and Astronomy: the *Brāhmasphuṭasiddhānta* (Extensive Treatise of Brahma) (628), a theoretical treatise, and the *Khaṇḍakhādyaka*, a more practical text.



## The rule of signs by Brahmagupta (628)

18.30. [The sum] of two positives is positives, of two negatives negative;

## The rule of signs by Brahmagupta (628)

18.30. [The sum] of two positives is positives, of two negatives negative;

- The **abstraction** is that you do not (always) need to know the **absolute value** of the arguments to know the **sign** of the result;

## The rule of signs by Brahmagupta (628)

18.30. [The sum] of two positives is positives, of two negatives negative;

- The **abstraction** is that you do not (always) need to know the **absolute value** of the arguments to know the **sign** of the result;
- Sometimes **imprecise** (don't know the sign of the sum of a positive and a negative)

## The rule of signs by Brahmagupta (628)

18.30. [The sum] of two positives is positives, of two negatives negative;

- The **abstraction** is that you do not (always) need to know the **absolute value** of the arguments to know the **sign** of the result;
- Sometimes **imprecise** (don't know the sign of the sum of a positive and a negative)
- **Useful in practice** (if you know what to do when you don't know the sign)

## The rule of signs by Brahmagupta (628)

18.30. [The sum] of two positives is positives, of two negatives negative;

- The **abstraction** is that you do not (always) need to know the **absolute value** of the arguments to know the **sign** of the result;
- Sometimes **imprecise** (don't know the sign of the sum of a positive and a negative)
- **Useful in practice** (if you know what to do when you don't know the sign)
- e.g. in **compilation**: do not optimize (a division by 2 into a shift when positive<sup>(\*)</sup>)

<sup>(\*)</sup> Unless processor uses 2's complement and can shift the sign.

## The rule of signs by Brahmagupta (628)

18.30. [The sum] of two positives is positives, of two negatives negative;  
[...]

18.32. A negative minus zero is negative, a positive [minus zero] positive; zero [minus zero] is zero. When a positive is to be subtracted from a negative or a negative from a positive, then it is to be added.

## The rule of signs by Brahmagupta (628)

18.30. [The sum] of two positives is positives, of two negatives negative; [...]

18.32. A negative minus zero is negative, a positive [minus zero] positive; zero [minus zero] is zero. When a positive is to be subtracted from a negative or a negative from a positive, then it is to be added.

18.33. The product of a negative and a positive is negative, of two negatives positive, and of positives positive; the product of zero and a negative, of zero and a positive, or of two zeros is zero.

## The rule of signs by Brahmagupta (628)

18.30. [The sum] of two positives is positives, of two negatives negative; [...]

18.32. A negative minus zero is negative, a positive [minus zero] positive; zero [minus zero] is zero. When a positive is to be subtracted from a negative or a negative from a positive, then it is to be added.

18.33. The product of a negative and a positive is negative, of two negatives positive, and of positives positive; the product of zero and a negative, of zero and a positive, or of two zeros is zero.

18.34. A positive divided by a positive or a negative divided by a negative is positive; **a zero divided by a zero is zero**; a positive divided by a negative is negative; a negative divided by a positive is [also] negative.

wrong

## The rule of signs by Michel Sintzoff (1972)

For example,  $a \times a + b \times b$  yields the value 25 when  $a$  is 3 and  $b$  is  $-4$ , and when  $+$  and  $\times$  are the arithmetic multiplication and addition. But  $a \times a + b \times b$  yields always the object "pos" when  $a$  and  $b$  are the objects "pos" or "neg", and when the valuation is defined as follows :

pos+pos=pos	pos×pos=pos
pos+neg=pos,neg	pos×neg=neg
neg+pos=pos,neg	neg×pos=neg
neg+neg=neg	neg×neg=pos
$V(p+q)=V(p)+V(q)$	$V(p \times q)=V(p) \times V(q)$
$V(0)=V(1)=\dots=pos$	
$V(-1)=V(-2)=\dots=neg$	

The valuation of  $a \times a + b \times b$  yields "pos" by the following computations :

$V(a)=pos,neg$	$V(b)=pos,neg$
$V(a \times a)=pos \times pos, neg \times neg$	$V(b \times b)=pos \times pos, neg \times neg$
$=pos, pos=pos$	$=pos, pos=pos$
$V(a \times a + b \times b)=V(a \times a)+V(b \times b)=pos+pos=pos$	

This valuation proves that the result of  $a \times a + b \times b$  is always positive and hence allows to compute its square root without any preliminary dynamic test on its sign. On the other hand, the

## The rule of signs by Michel Sintzoff (1972)

For example,  $a \times a + b \times b$  yields the value 25 when  $a$  is 3 and  $b$  is  $-4$ , and when  $+$  and  $\times$  are the arithmetic multiplication and addition. But  $a \times a + b \times b$  yields always the object "pos" when  $a$  and  $b$  are the objects "pos" or "neg", and when the valuation is defined as follows :

pos+pos=pos	pos×pos=pos
pos+neg=pos,neg	pos×neg=neg
neg+pos=pos,neg	neg×pos=neg
neg+neg=neg	neg×neg=pos
$V(p+q)=V(p)+V(q)$	$V(p \times q)=V(p) \times V(q)$
$V(0)=V(1)=\dots=pos$	
$V(-1)=V(-2)=\dots=neg$	

The valuation of  $a \times a + b \times b$  yields "pos" by the following computations :

$V(a)=pos,neg$	$V(b)=pos,neg$
$V(a \times a)=pos \times pos, neg \times neg$	$V(b \times b)=pos \times pos, neg \times neg$
$=pos, pos=pos$	$=pos, pos=pos$
$V(a \times a + b \times b)=V(a \times a)+V(b \times b)=pos+pos=pos$	

This valuation proves that the result of  $a \times a + b \times b$  is always positive and hence allows to compute its square root without any preliminary dynamic test on its sign. On the other hand, the

# The rule of signs by Michel Sintzoff (1972)

For example,  $a \times a + b \times b$  yields the value 25 when  $a$  is 3 and  $b$  is  $-4$ , and when  $+$  and  $\times$  are the arithmetic multiplication and addition. But  $a \times a + b \times b$  yields always the object "pos" when  $a$  and  $b$  are the objects "pos" or "neg", and when the valuation is defined as follows :

$pos + pos = pos$                        $pos \times pos = pos$   
 $pos + neg = pos, neg$                  $pos \times neg = neg$  ← wrong  
 $neg + pos = pos, neg$                  $neg \times pos = neg$   
 $neg + neg = neg$                        $neg \times neg = pos$   
 $V(p+q) = V(p) + V(q)$                $V(p \times q) = V(p) \times V(q)$   
 $V(0) = V(1) = \dots = pos$   
 $V(-1) = V(-2) = \dots = neg$

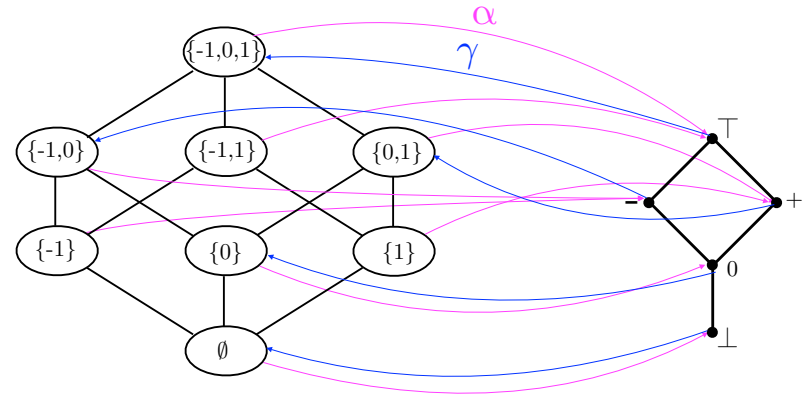
$0 \in pos \times -1 \in neg$   
 $= 0 \notin neg$

The valuation of  $a \times a + b \times b$  yields "pos" by the following computations :

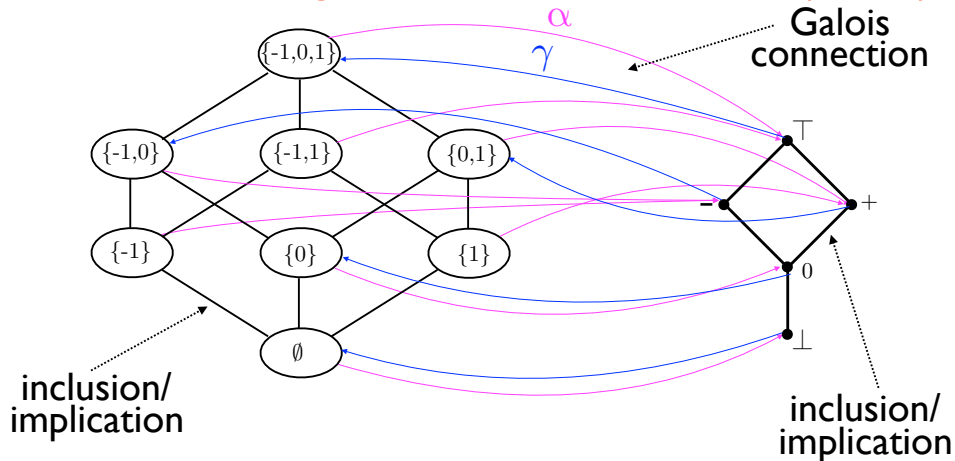
$V(a) = pos, neg$                        $V(b) = pos, neg$   
 $V(a \times a) = pos \times pos, neg \times neg$      $V(b \times b) = pos \times pos, neg \times neg$   
 $= pos, pos = pos$                        $= pos, pos = pos$   
 $V(a \times a + b \times b) = V(a \times a) + V(b \times b) = pos + pos = pos$

This valuation proves that the result of  $a \times a + b \times b$  is always positive and hence allows to compute its square root without any preliminary dynamic test on its sign. On the other hand, the

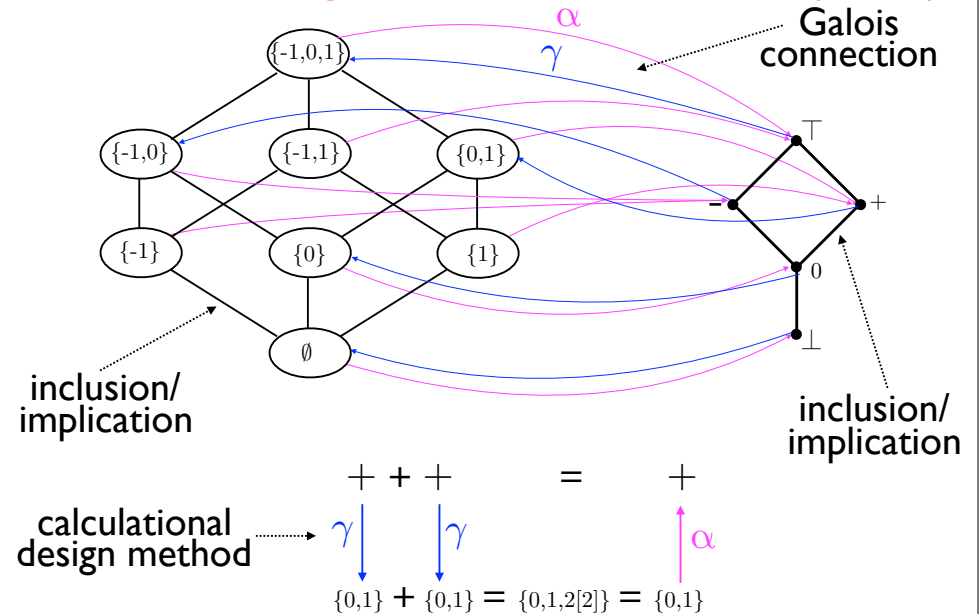
# The rule of signs Cousot & Cousot (1979)



# The rule of signs Cousot & Cousot (1979)



# The rule of signs Cousot & Cousot (1979)



# Application of abstract interpretation to static analysis

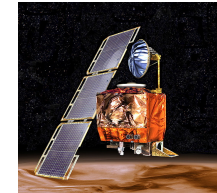
## All computer scientists have experienced bugs



Ariane 501 failure  
(overflow)



Patriot failure  
(float rounding)



Mars orbiter loss  
(unit error)



Heartbleed  
(buffer overrun)

- Checking the **presence** of bugs by debugging is great
- Proving their **absence** by static analysis is even better!

## Static analysis

- Check program properties (automatically, using the program text only, without running the program)
- Difficulties:
  - Undecidability / complexity:
    - Precision
    - Scalability
  - Soundness (correctness)
  - Induction: widening/narrowing

## Fixpoint

```

{y ≥ 0} ← hypothesis
x = y
{I(x, y)} ← loop invariant
while (x > 0) {
  x = x - 1;
}
    
```

Fixpoint equation

Floyd-Naur-Hoare verification conditions:

$$(y \geq 0 \wedge x = y) \implies I(x, y) \quad \text{initialisation}$$

$$(I(x, y) \wedge x > 0 \wedge x' = x - 1) \implies I(x', y) \quad \text{iteration}$$


Equivalent fixpoint equation:

$$I(x, y) = x \geq 0 \wedge (x = y \vee I(x + 1, y)) \quad (\text{i.e. } I = F(I)^{(5)})$$

<sup>(5)</sup> We look for the most precise invariant  $I$ , implying all others, that is  $\text{lfp } F$ .

# Iterates

Iterates  $I = \lim_{n \rightarrow \infty} F^n(\text{false})$

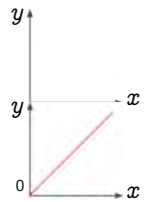
$$I^0(x, y) = \text{false}$$


# Iterates

Iterates  $I = \lim_{n \rightarrow \infty} F^n(\text{false})$

$$I^0(x, y) = \text{false}$$

$$I^1(x, y) = x \geq 0 \wedge (x = y \vee I^0(x + 1, y))$$

$$= 0 \leq x = y$$


# Iterates

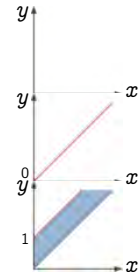
Iterates  $I = \lim_{n \rightarrow \infty} F^n(\text{false})$

$$I^0(x, y) = \text{false}$$

$$I^1(x, y) = x \geq 0 \wedge (x = y \vee I^0(x + 1, y))$$

$$= 0 \leq x = y$$

$$I^2(x, y) = x \geq 0 \wedge (x = y \vee I^1(x + 1, y))$$

$$= 0 \leq x \leq y \leq x + 1$$


# Iterates

Iterates  $I = \lim_{n \rightarrow \infty} F^n(\text{false})$

$$I^0(x, y) = \text{false}$$

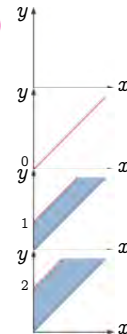
$$I^1(x, y) = x \geq 0 \wedge (x = y \vee I^0(x + 1, y))$$

$$= 0 \leq x = y$$

$$I^2(x, y) = x \geq 0 \wedge (x = y \vee I^1(x + 1, y))$$

$$= 0 \leq x \leq y \leq x + 1$$

$$I^3(x, y) = x \geq 0 \wedge (x = y \vee I^2(x + 1, y))$$

$$= 0 \leq x \leq y \leq x + 2$$


# Convergence acceleration: widening

Accelerated Iterates  $I = \lim_{n \rightarrow \infty} F^n(\text{false})$

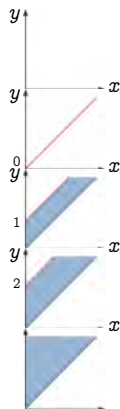
$$I^0(x, y) = \text{false}$$

$$I^1(x, y) = x \geq 0 \wedge (x = y \vee I^0(x + 1, y)) \\ = 0 \leq x = y$$

$$I^2(x, y) = x \geq 0 \wedge (x = y \vee I^1(x + 1, y)) \\ = 0 \leq x \leq y \leq x + 1$$

$$I^3(x, y) = x \geq 0 \wedge (x = y \vee I^2(x + 1, y)) \\ = 0 \leq x \leq y \leq x + 2$$

$$I^4(x, y) = I^2(x, y) \nabla I^3(x, y) \leftarrow \text{widening} \\ = 0 \leq x \leq y$$



# Fixed point

Accelerated Iterates  $I = \lim_{n \rightarrow \infty} F^n(\text{false})$

$$I^0(x, y) = \text{false}$$

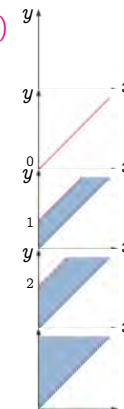
$$I^1(x, y) = x \geq 0 \wedge (x = y \vee I^0(x + 1, y)) \\ = 0 \leq x = y$$

$$I^2(x, y) = x \geq 0 \wedge (x = y \vee I^1(x + 1, y)) \\ = 0 \leq x \leq y \leq x + 1$$

$$I^3(x, y) = x \geq 0 \wedge (x = y \vee I^2(x + 1, y)) \\ = 0 \leq x \leq y \leq x + 2$$

$$I^4(x, y) = I^2(x, y) \nabla I^3(x, y) \leftarrow \text{widening} \\ = 0 \leq x \leq y$$

$$I^5(x, y) = x \geq 0 \wedge (x = y \vee I^4(x + 1, y)) \\ = I^4(x, y) \text{ fixed point!}$$



# Octagons

Accelerated Iterates  $I = \lim_{n \rightarrow \infty} F^n(\text{false})$

$$I^0(x, y) = \text{false}$$

$$I^1(x, y) = x \geq 0 \wedge (x = y \vee I^0(x + 1, y)) \\ = 0 \leq x = y$$

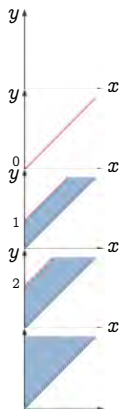
$$I^2(x, y) = x \geq 0 \wedge (x = y \vee I^1(x + 1, y)) \\ = 0 \leq x \leq y \leq x + 1$$

$$I^3(x, y) = x \geq 0 \wedge (x = y \vee I^2(x + 1, y)) \\ = 0 \leq x \leq y \leq x + 2$$

$$I^4(x, y) = I^2(x, y) \nabla I^3(x, y) \leftarrow \text{widening} \\ = 0 \leq x \leq y$$

$$I^5(x, y) = x \geq 0 \wedge (x = y \vee I^4(x + 1, y)) \\ = I^4(x, y) \text{ fixed point!}$$

The invariants are computer representable with octagons!



# Industrialisation: Development in cooperation with Airbus France

- Automatic proofs of absence of runtime errors in **Electric Flight Control Software**:



- A340/600: 132.000 lines of C, 40mn on a PC 2.8 GHz, 300 Mb (Nov. 2003)

- A380: 1.000.000 lines of C, 34h, 8 Gb (Nov. 2005)

**no false alarm, World premières !**



- Automatic proofs of absence of runtime errors in the **ATV software**<sup>(2)</sup>:

- C version of the automatic docking software: 102.000 lines of C, 23s on a Quad-Core AMD Opteron™ processor, 16 Gb (Apr. 2008)

<sup>(2)</sup> the Jules Vernes Automated Transfer Vehicle (ATV) enabling ESA to transport payloads to the International Space Station.



# Application of abstract interpretation to program proof methods

## Maximal execution trace

```
#include <stdio.h>
int main() {
    int x,y;
    printf("Enter an integer: ");
    scanf("%d",&x); y = x;
    /* 1: */ while (x != 0) {
        printf("x = %d, y = %d\n",x,y);
    /* 2: */     x = x - 1;
    /* 3: */     y = y + 2;
    }
    /* 4: */ printf("x = %d, y = %d\n",x,y); }

Enter an integer: 3
x = 3, y = 3
x = 2, y = 5
x = 1, y = 7
x = 0, y = 9

Enter an integer: -1
x = -1, y = -1
x = -2, y = 1
x = -3, y = 3
x = -4, y = 5
...
x = -738245, y = 1476487
...
```

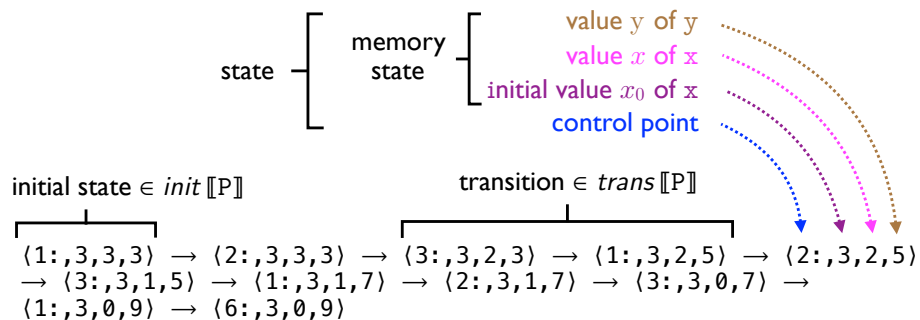
$\langle 1:,3,3,3 \rangle \rightarrow \langle 2:,3,3,3 \rangle \rightarrow \langle 3:,3,2,3 \rangle \rightarrow \langle 1:,3,2,5 \rangle \rightarrow \langle 2:,3,2,5 \rangle$   
 $\rightarrow \langle 3:,3,1,5 \rangle \rightarrow \langle 1:,3,1,7 \rangle \rightarrow \langle 2:,3,1,7 \rangle \rightarrow \langle 3:,3,0,7 \rangle \rightarrow$   
 $\langle 1:,3,0,9 \rangle \rightarrow \langle 6:,3,0,9 \rangle$

## Maximal execution trace

```
#include <stdio.h>
int main() {
    int x,y;
    printf("Enter an integer: ");
    scanf("%d",&x); y = x;
    /* 1: */ while (x != 0) {
        printf("x = %d, y = %d\n",x,y);
    /* 2: */     x = x - 1;
    /* 3: */     y = y + 2;
    }
    /* 4: */ printf("x = %d, y = %d\n",x,y); }

Enter an integer: 3
x = 3, y = 3
x = 2, y = 5
x = 1, y = 7
x = 0, y = 9

Enter an integer: -1
x = -1, y = -1
x = -2, y = 1
x = -3, y = 3
x = -4, y = 5
...
x = -738245, y = 1476487
```



## Maximal trace semantics

- The **trace semantics of a program** is the set of all possible maximal finite or infinite execution traces for that program
- The **trace semantics of a programming language** maps programs to their trace semantics

## Inductive definition

- **Partial traces:**
  - A trace with one initial state is a partial trace
  - A partial trace extended by a transition is a partial trace
- **Maximal traces:**
  - Finite traces with no extension by a transition
  - Infinite traces which prefixes are all partial traces

## Fixpoint partial trace semantics

- initial states of program P:  $init \llbracket P \rrbracket$
- transitions of programs P:  $trans \llbracket P \rrbracket$
- $F^t \llbracket P \rrbracket X = \{ s \mid s \in init \llbracket P \rrbracket \} \cup \{ \sigma s s' \mid \sigma s \in X \wedge s s' \in trans \llbracket P \rrbracket \}$
- $S^t \llbracket P \rrbracket = lfp^{\subseteq} F^t \llbracket P \rrbracket$

## Invariance abstraction

- Collect at each control point the possible values of variables when execution reaches that control point
- $\alpha(X)c = \{ m \mid \exists \sigma, \sigma'. \sigma \langle c, m \rangle \sigma' \in X \}$
- Invariance semantics:  $S^i \llbracket P \rrbracket = \alpha(S^t \llbracket P \rrbracket)$

## Invariance abstraction

- Collect at each control point the possible values of variables when execution reaches that control point
- $S^i \llbracket P \rrbracket = \alpha(S^t \llbracket P \rrbracket)c = \{ m \mid \exists \sigma, \sigma'. \sigma \langle c, m \rangle \sigma' \in S^t \llbracket P \rrbracket \}$

```

#include <stdio.h>
int main() {
    int x,y;
    printf("Enter an integer: ");
    scanf("%d",&x); y = x;
    while (x != 0) {
        printf("x = %d, y = %d\n",x,y);
        x = x - 1;
        y = y + 2;
    }
    printf("x = %d, y = %d\n",x,y); }

```

$\{ \langle x_0, x, y \rangle \mid y = 3x_0 - 2x \}$  ← /\* 1: \*/  
 $\{ \langle x_0, x, y \rangle \mid y = 3x_0 - 2x \}$  ← /\* 2: \*/  
 $\{ \langle x_0, x, y \rangle \mid y = 3x_0 - 2x - 2 \}$  ← /\* 3: \*/  
 $\{ \langle x_0, x, y \rangle \mid y = 3x_0 \wedge x = 0 \}$  ← /\* 4: \*/

## Calculations design of the verification conditions

- $\alpha(F^t[[P]]X)$   
=  $\lambda c. \{m \mid \exists \sigma, \sigma'. \sigma \langle c, m \rangle \sigma' \in X\}$   
= ...  
=  $F^i[[P]](\alpha(X))$   
where  $F^i[[P]]$  are the Turing/Floyd/Naur/Hoare verification conditions
- It follows that  $S^i[[P]] = \text{lfp}^{\dot{c}} F^i[[P]]$
- The proof method is then by fixpoint induction (Tarski 1955)

## Application to the semantics of programming languages

## Abstraction to denotational semantics

- All known semantics are abstractions of a most precise semantics

- The maximal trace semantics  $S^m[[P]]$  (maximal finite and infinite execution traces)
- Denotational semantics abstraction:
  - $S^d[[P]] = \alpha(S^m[[P]])$
  - $\alpha(X) = \lambda s. \{s' \mid \exists \sigma. s \sigma s' \in X\} \cup \{\perp \mid \exists \sigma. s \sigma \dots \in X\}$

*i.e.* a map of initial states to the set of final states plus  $\perp$  in case of non-termination



# Industrialisation: Astrée

The screenshot shows the Astrée IDE interface. The main window displays the original source code in C, with annotations for type casts and pointer arithmetic. The left sidebar shows the project structure and analysis results. The bottom panel shows a list of findings, including an alarm for an out-of-bound array access.

Order	Type	Category	Location	Classification	Comment
4	Alarm (C)	Out-of-bound array access	scenarios.c:81.17-19		out-of-bound array index (15) not included in [0, 14]
5	Definite Alarm (A)	Possible overflow upon dereference	scenarios.c:81.6-20		invalid dereference: dereferencing 1 byte(s) at scenarios.c:81.6-20
6	Alarm (A)	Use of uninitialized variables	scenarios.c:84.8-23		uninitialized read: reading 4 byte(s) at scenarios.c:84.8-23
7	Definite Alarm (A)	Possible overflow upon dereference	scenarios.c:85.6-17		invalid dereference: dereferencing 1 byte(s) at scenarios.c:85.6-17
8	Alarm (A)	Assertion failure	scenarios.c:127.4-40		assert failure: _ASTREE_assert(second == 1) at scenarios.c:127.4-40

# Industrialisation: Astrée

The screenshot shows the Astrée IDE interface with the 'Overview' tab selected. A pie chart displays the distribution of findings, with 62.5% of findings being 'Alarms'. Below the chart, a table lists the findings.

Type	Category	Location	Classification	Comment
Notification	Invalid conversion	scenarios.c:73.4-20		translate_warning(type): conversion from floating-point to integer
Alarm (C)	Overflow in conversion	scenarios.c:73.4-20		double->signed short conversion range [0, 40000] not included in [0, 32767]
Alarm (A)	Use of uninitialized variables	scenarios.c:80.8-23		uninitialized read: reading 4 byte(s) at offset(s) 0 in scenarios.c:80.8-23
Alarm (C)	Out-of-bound array access	scenarios.c:81.17-19		out-of-bound array index (15) not included in [0, 14]
Definite Alarm (A)	Possible overflow upon dereference	scenarios.c:81.6-20		invalid dereference: dereferencing 1 byte(s) at scenarios.c:81.6-20
Alarm (A)	Use of uninitialized variables	scenarios.c:84.8-23		uninitialized read: reading 4 byte(s) at offset(s) 0 in scenarios.c:84.8-23

## Many other static analyzers

- Julia (Java) <http://www.juliasoft.com>
- Ikos, NASA <https://ti.arc.nasa.gov/opensource/ikos/>
- Clousot for code contract, Microsoft, <https://github.com/Microsoft/CodeContracts>
- Infer (Facebook) <http://fbinfer.com>
- Zoncolan (Facebook)
- Google
- ...

## Static memory analysis for software development

- Users of Astrée:



- Why not all software developers use static analysis tools?

## Irresponsibility

- Computer engineering is the only technology where developers are not responsible for their errors, even the trivial ones:

DISCLAIMER OF WARRANTIES. ... MICROSOFT AND ITS SUPPLIERS PROVIDE THE SOFTWARE, AND SUPPORT SERVICES (IF ANY) AS IS AND WITH ALL FAULTS, AND MICROSOFT AND ITS SUPPLIERS HEREBY DISCLAIM ALL OTHER WARRANTIES AND CONDITIONS, WHETHER EXPRESS, IMPLIED OR STATUTORY, INCLUDING, BUT NOT LIMITED TO, ANY (IF ANY) IMPLIED WARRANTIES, DUTIES OR CONDITIONS OF MERCHANTABILITY, OF FITNESS FOR A PARTICULAR PURPOSE, OF RELIABILITY OR AVAILABILITY, OF ACCURACY OR COMPLETENESS OF RESPONSES, OF RESULTS, OF WORKMANLIKE EFFORT, OF LACK OF VIRUSES, AND OF LACK OF NEGLIGENCE, ALL WITH REGARD TO THE SOFTWARE, AND THE PROVISION OF OR FAILURE TO PROVIDE SUPPORT OR OTHER SERVICES, INFORMATION, SOFTWARE, AND RELATED CONTENT THROUGH THE SOFTWARE OR OTHERWISE ARISING OUT OF THE USE OF THE SOFTWARE. ...

## The future

- Safety and security does matter to the general public
- Computer scientists will ultimately be held responsible for there errors
- At least the automatically discoverable ones
- Since this is now part of the state of the art
- Automatic static analysis, verification, etc has a brilliant future.

Francesco Logozzo, designer of the Zoncolan static analyzer at Facebook wrote me on 09/12/2016:

“Finding people who really know static analysis is very hard, you should tell your students that if they want a great job in a Silicon Valley company they should study abstract interpretation not JavaScript. Feel free to quote me on that ;-)”

## Selected bibliography

- Patrick Cousot, Radhia Cousot: Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. POPL 1977: 238-252
- Patrick Cousot, Nicolas Halbwachs: Automatic Discovery of Linear Restraints Among Variables of a Program. POPL 1978: 84-96
- Patrick Cousot, Radhia Cousot: Systematic Design of Program Analysis Frameworks. POPL 1979: 269-282
- Patrick Cousot, Radhia Cousot: Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation. PLILP 1992: 269-295
- Patrick Cousot: Types as Abstract Interpretations. POPL 1997: 316-331
- Patrick Cousot, Radhia Cousot: Temporal Abstract Interpretation. POPL 2000: 12-25
- Patrick Cousot, Radhia Cousot: Systematic design of program transformation frameworks by abstract interpretation. POPL 2002: 178-190
- Patrick Cousot: Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. Theor. Comput. Sci. 277(1-2): 47-103 (2002)
- Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, Xavier Rival: A static analyzer for large safety-critical software. PLDI 2003: 196-207
- Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, Xavier Rival: The ASTREÉ Analyzer. ESOP 2005: 21-30
- Patrick Cousot, Radhia Cousot, Roberto Giacobazzi: Abstract interpretation of resolution-based semantics. Theor. Comput. Sci. 410(46): 4724-4746 (2009)
- Patrick Cousot, Radhia Cousot: An abstract interpretation framework for termination. POPL 2012: 245-258
- Patrick Cousot, Radhia Cousot: A Galois connection calculus for abstract interpretation. POPL 2014: 3-4
- Julien Bertrane, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, Xavier Rival: Static Analysis and Verification of Aerospace Software by Abstract Interpretation. Foundations and Trends in Programming Languages 2(2-3): 71-190 (2015)

# The End, Thank You