

# Abstract Interpretation

29 septembre 2016, 18:00, Amphi 15  
4 Place Jussieu, 75005 Paris

**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

contact : colloquium@lip6.fr  
<http://www.lip6.fr/colloquium/>  
Vidéo disponible sur le site

## 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 debugged 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 professionals are only required to apply state of the art techniques, that is testing on finitely many cases. This state of the art is changing rapidly and so will irresponsibility, as in other manufacturing disciplines.

Scalable and cost-effective tools have appeared recently that can avoid bugs with possible dramatic consequences for example in transportation, banks, 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 to a particular reasoning on programs. This yields proof 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 imprecision. Bugs may be signalled that are impossible in any execution (but still none is forgotten). This has an economic cost, much less than 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 ès 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.



# 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  
Obfuscation  
Denotational semantics  
Theories combination  
Code contracts  
Quantum entanglement detection  
Steganography

Partial evaluation  
Effect systems  
Trace semantics  
Symbolic execution  
Type theory

Systems biology analysis  
Model checking  
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

Obfuscation  
Denotational semantics  
Theories combination  
Code contracts  
Steganography

Model checking  
Dependence analysis  
CEGAR  
Program transformation  
Interpolants  
Integrity analysis  
Bisimulation  
SMT solvers  
Tautology testers

Systems biology analysis  
Database query  
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

Obfuscation  
Denotational semantics  
Theories combination  
Code contracts  
Steganography

Systems biology analysis  
Model checking  
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



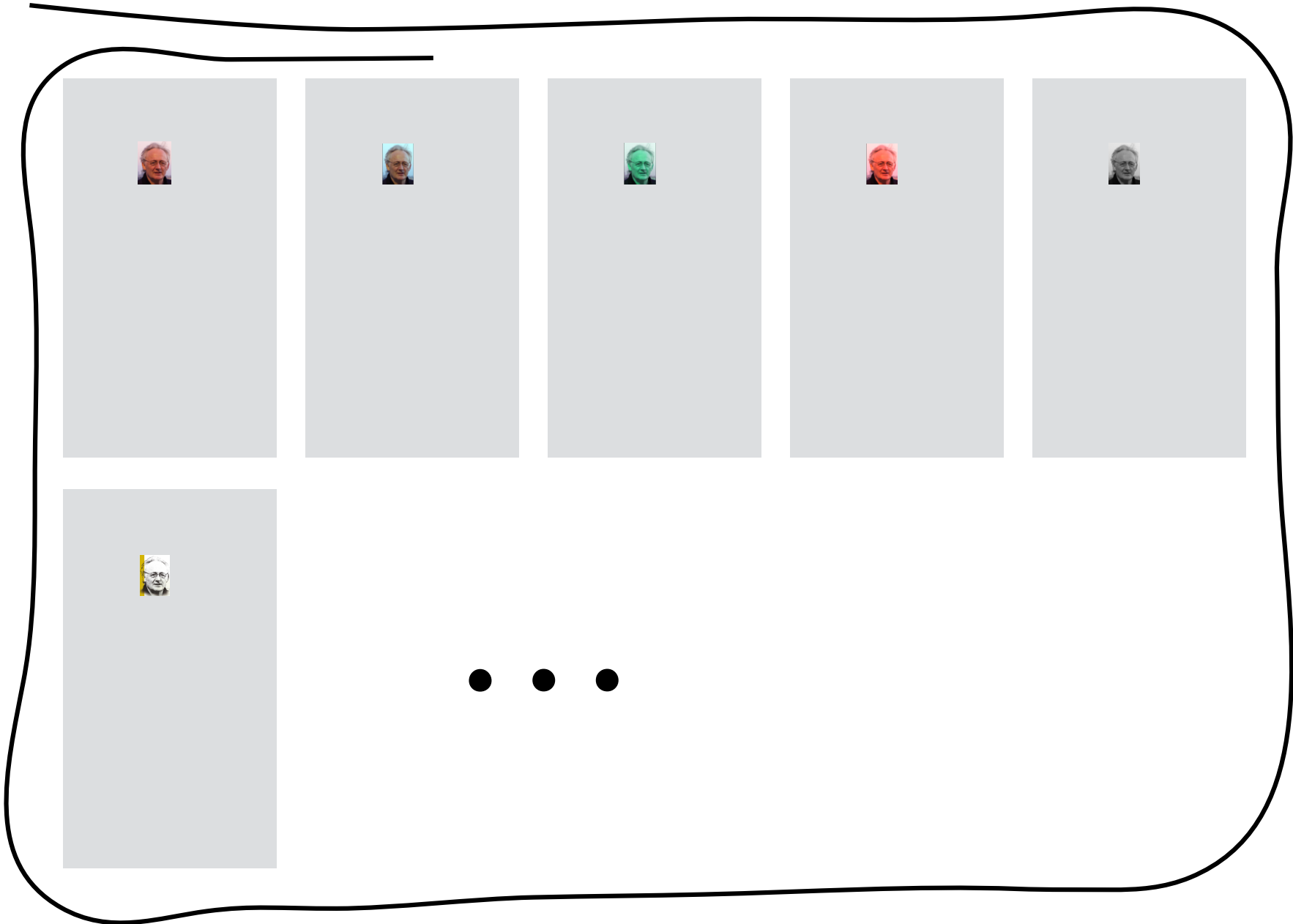
# Abstraction I



# Abstraction 2

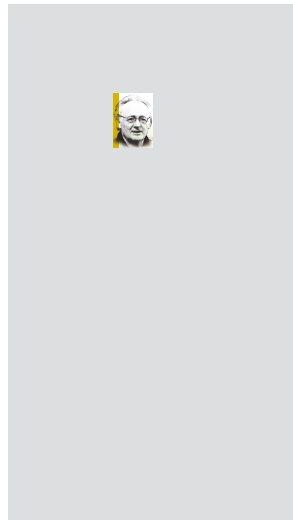
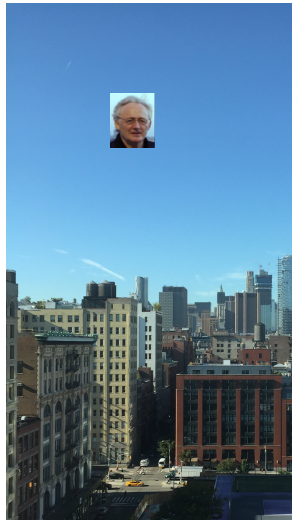
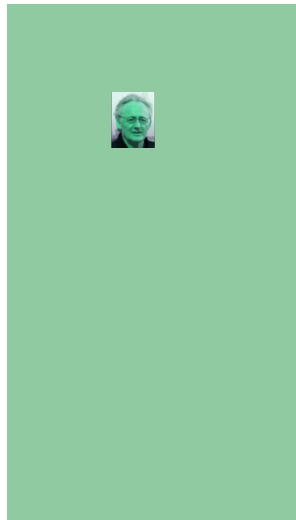
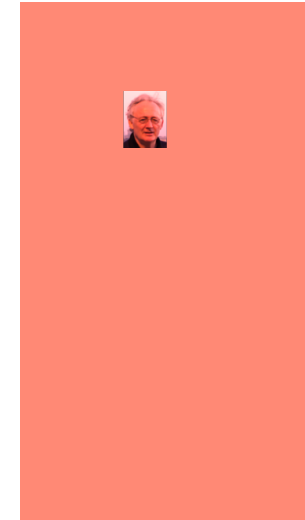
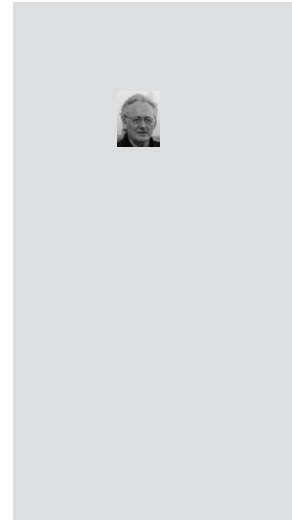
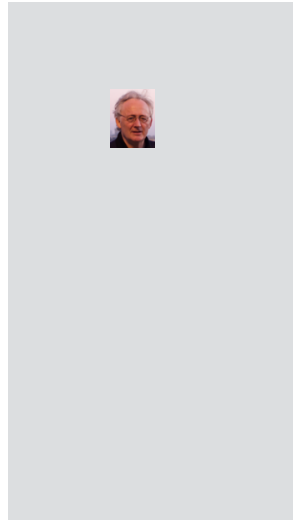


# Concretization 2

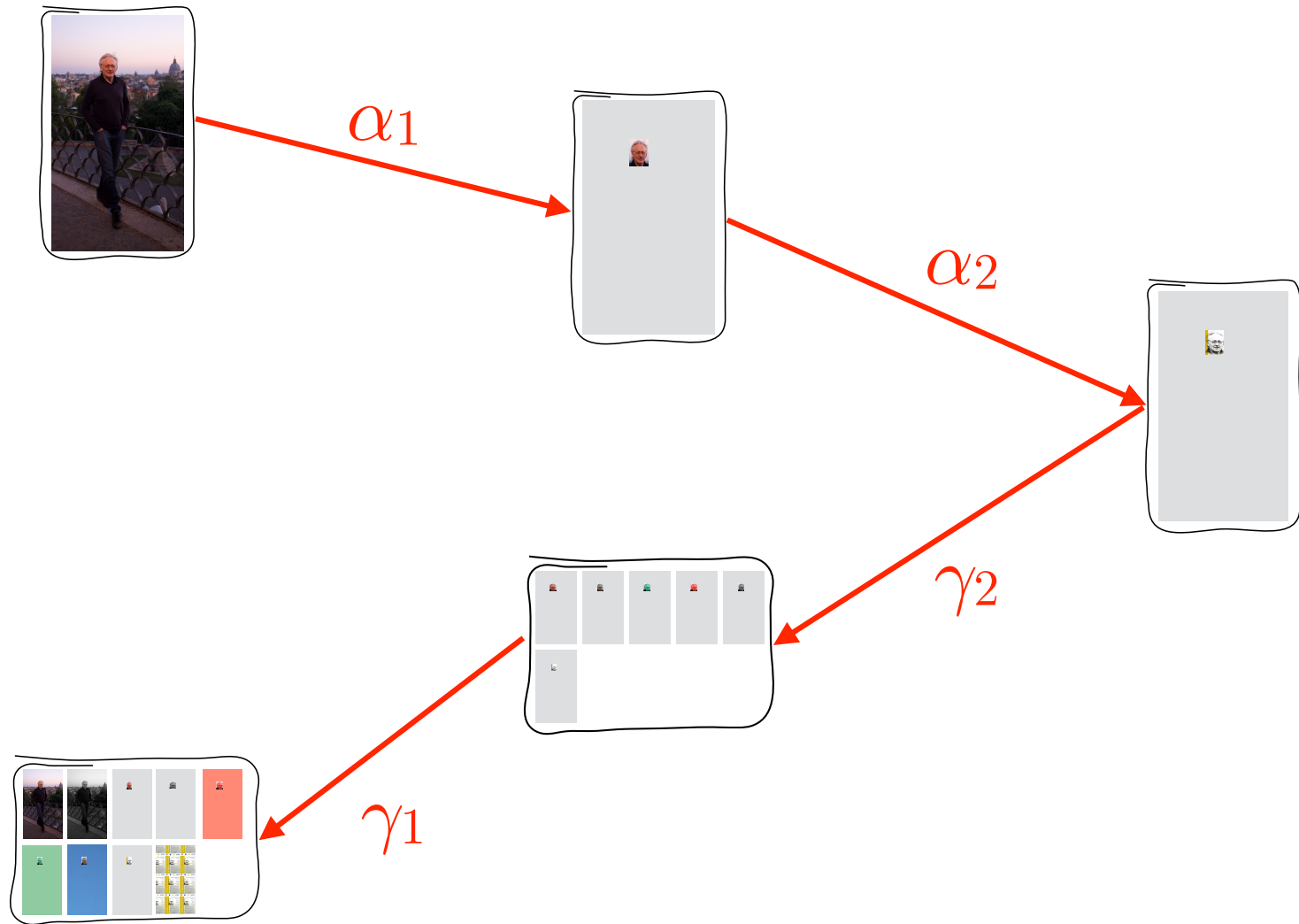




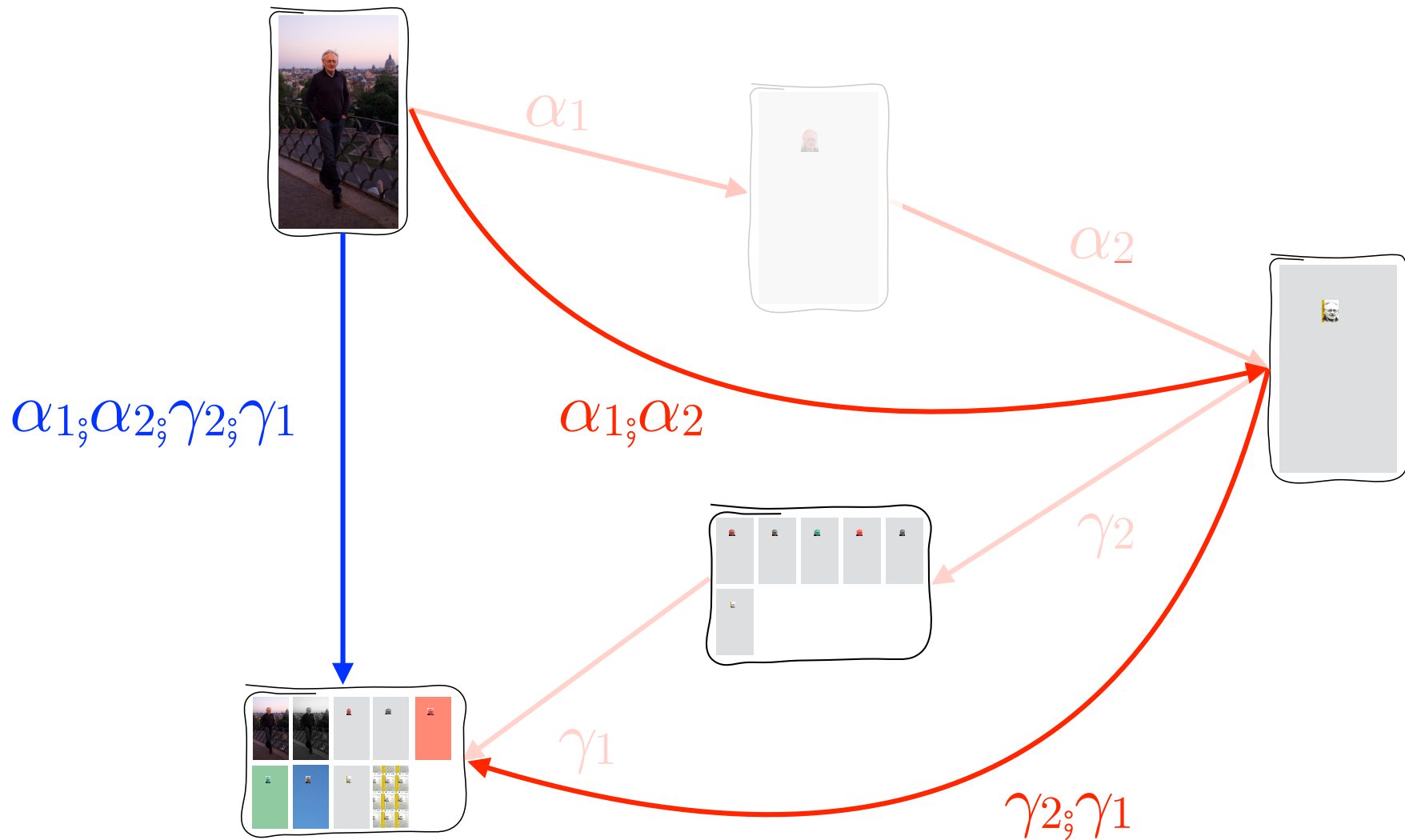
# Concretization I



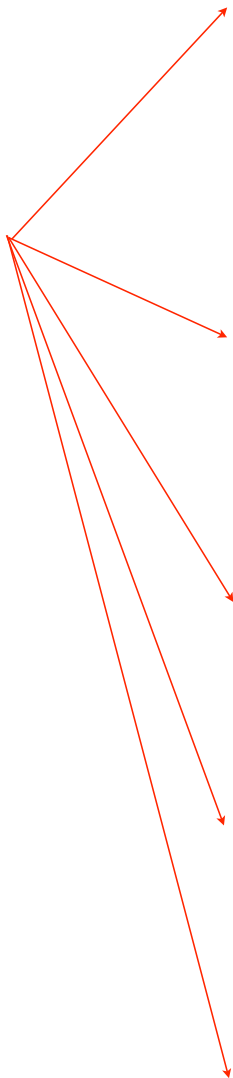
# Abstract interpretations



# Abstract interpretations



# Intuition 2



Height



Fingerprint



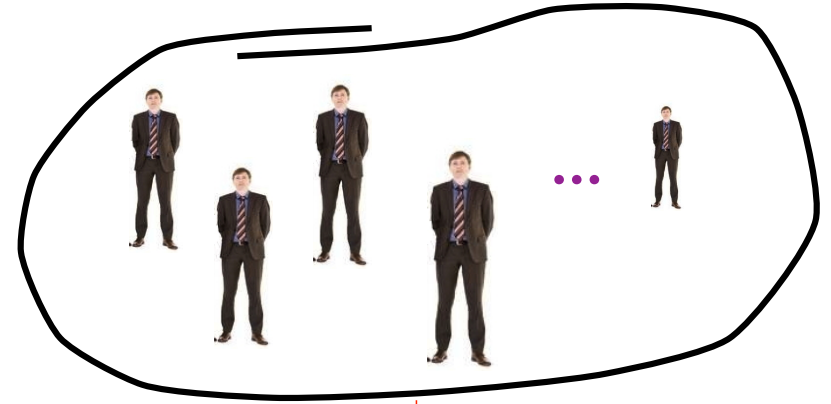
Eye color



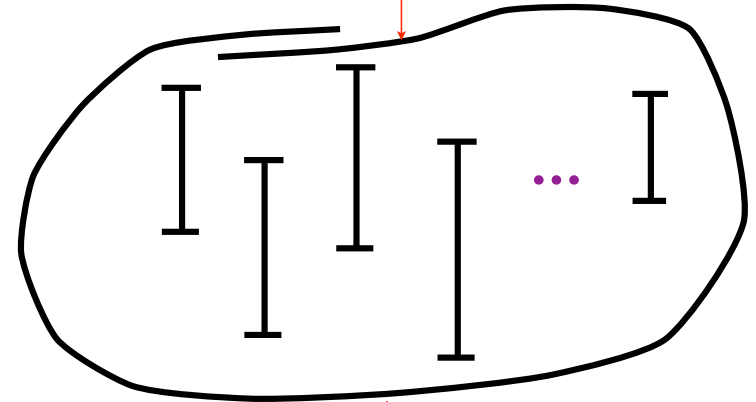
DNA



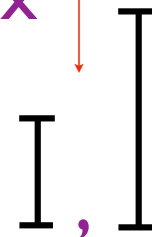
Phone metadata



Individual heights

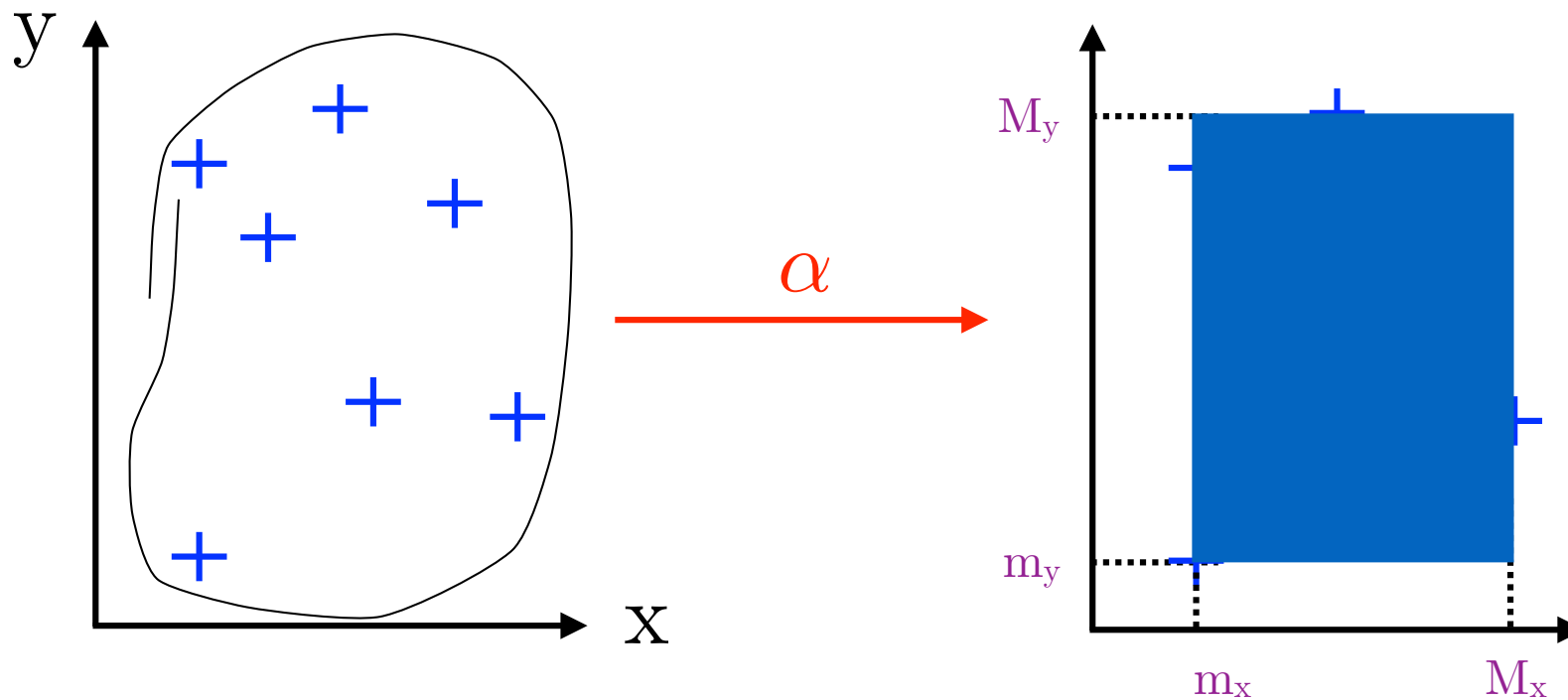


min, max



# Interval abstraction

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



Set of points

Interval abstraction  
 $[m_x, M_x] \times [m_y, M_y]$

# 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

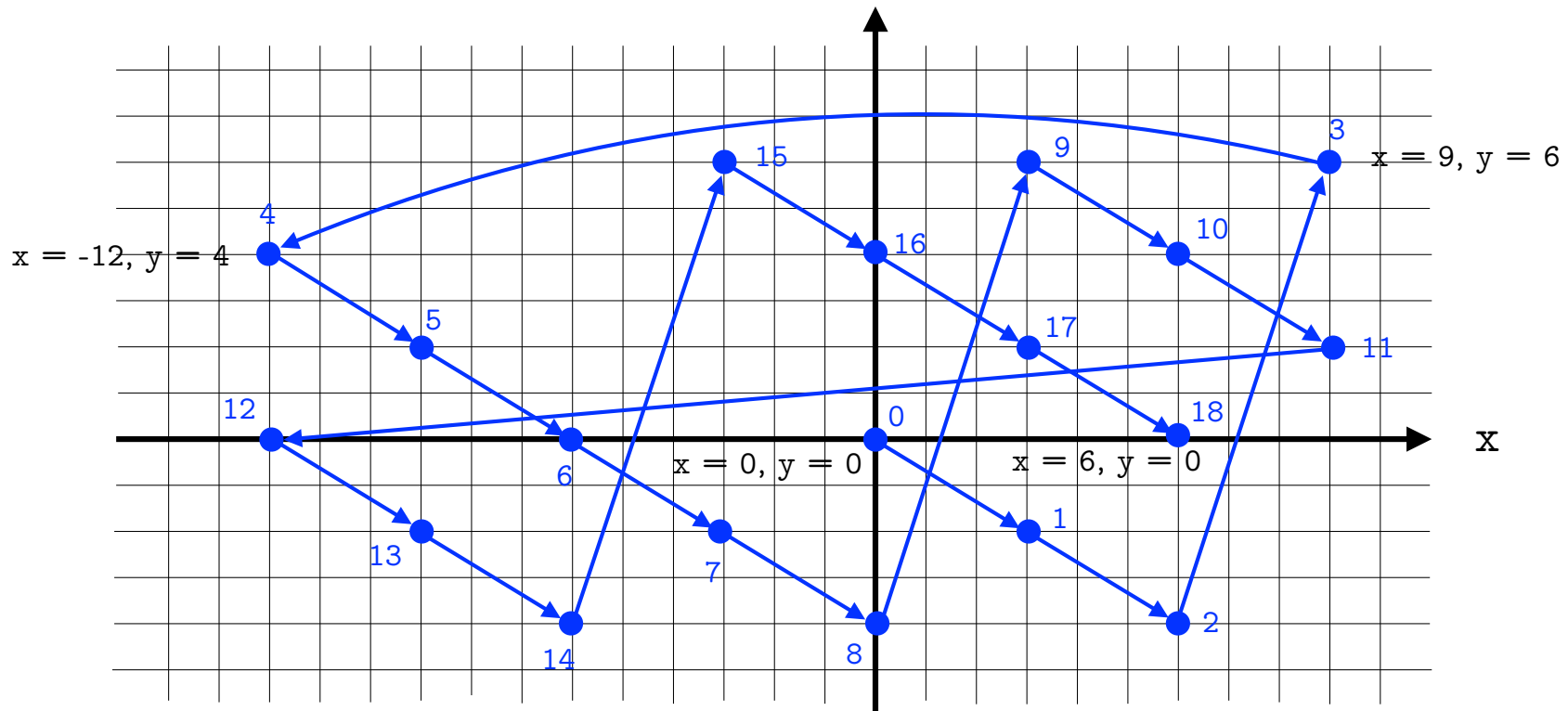
x = 0, y = 4

x = 3, y = 2

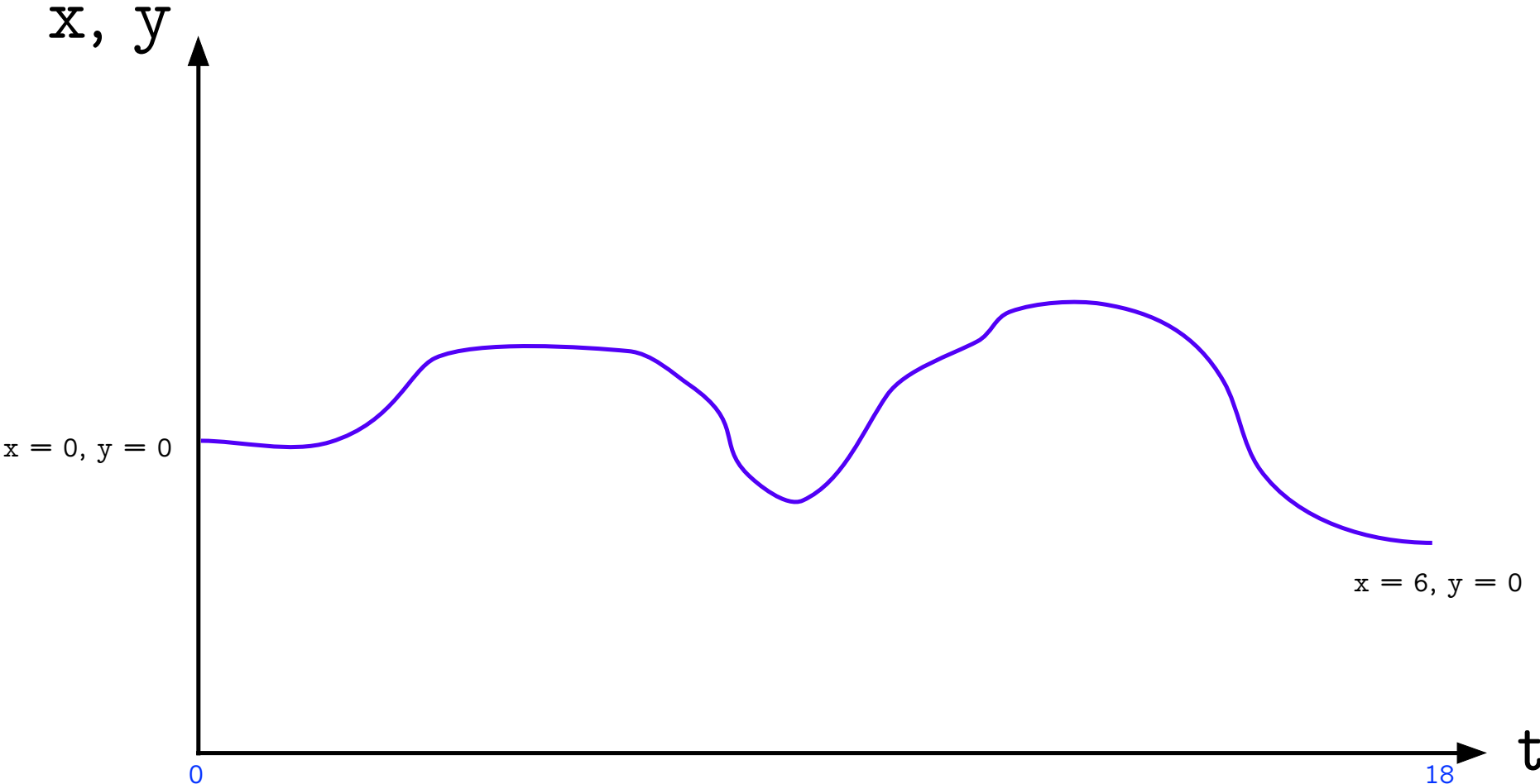
x = 6, y = 0



# Graphical representation of the execution (I)



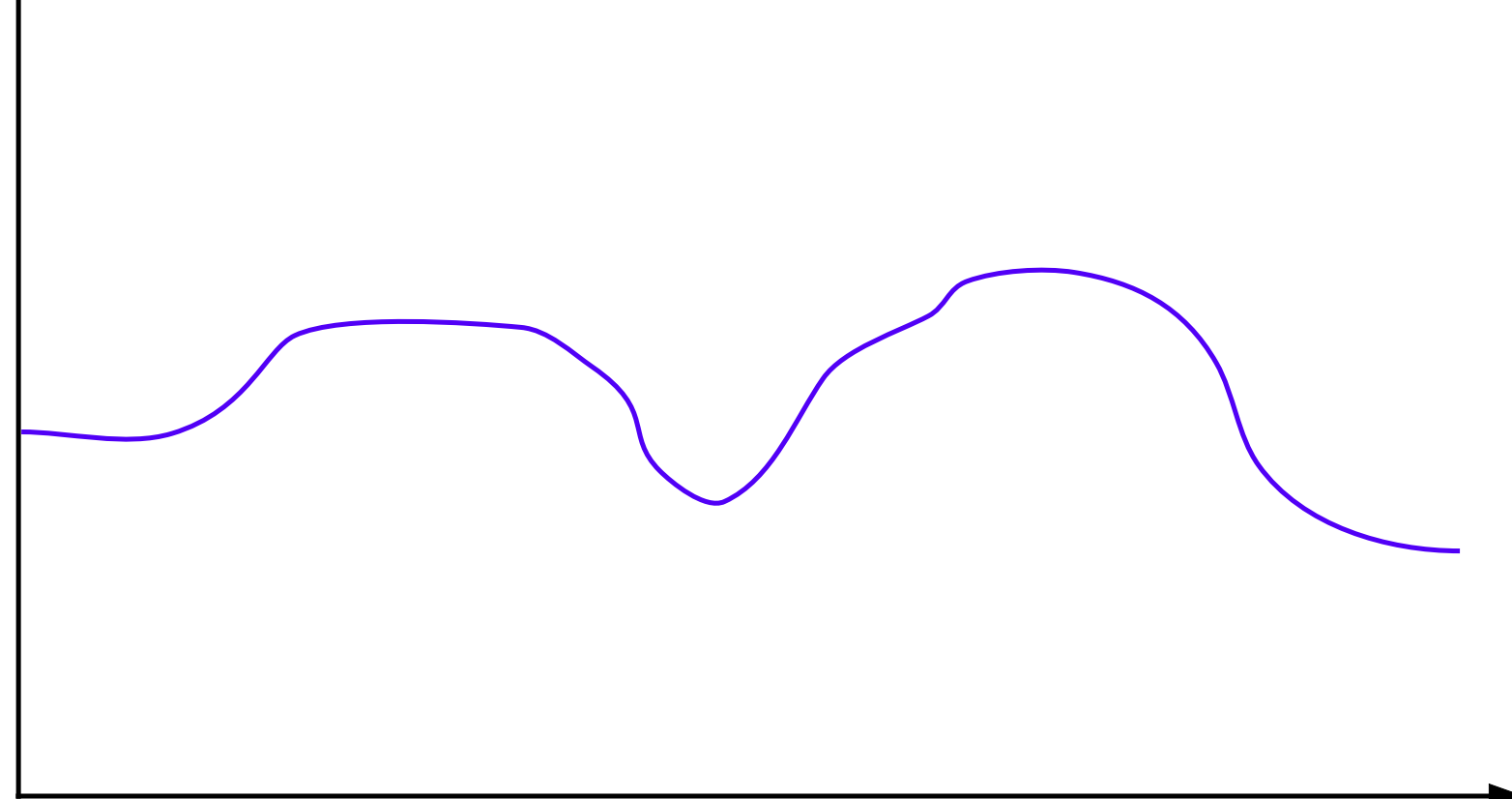
# Graphical representation of the execution (2)



# Semantics

*Formalize what it means to run a program*

**state**

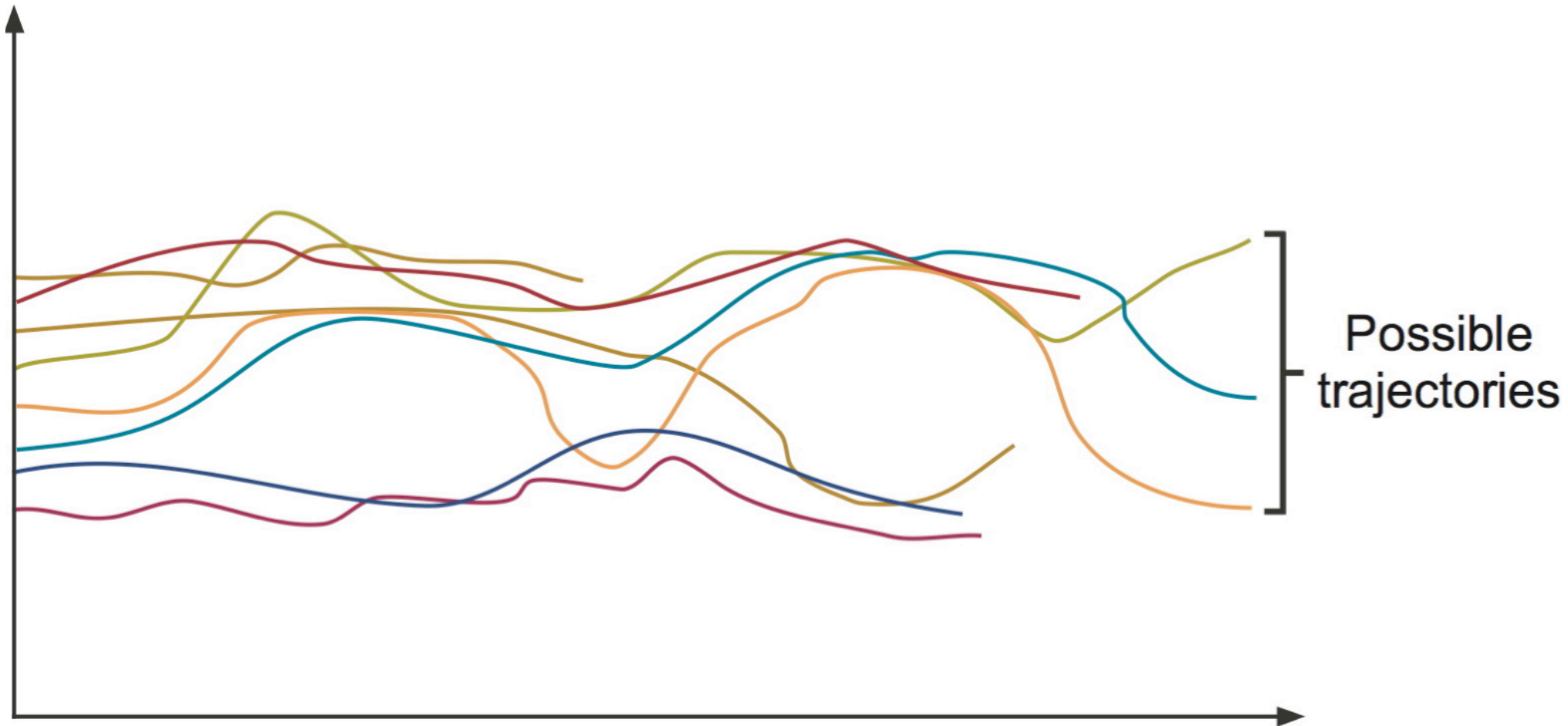


**trajectory**

**time**

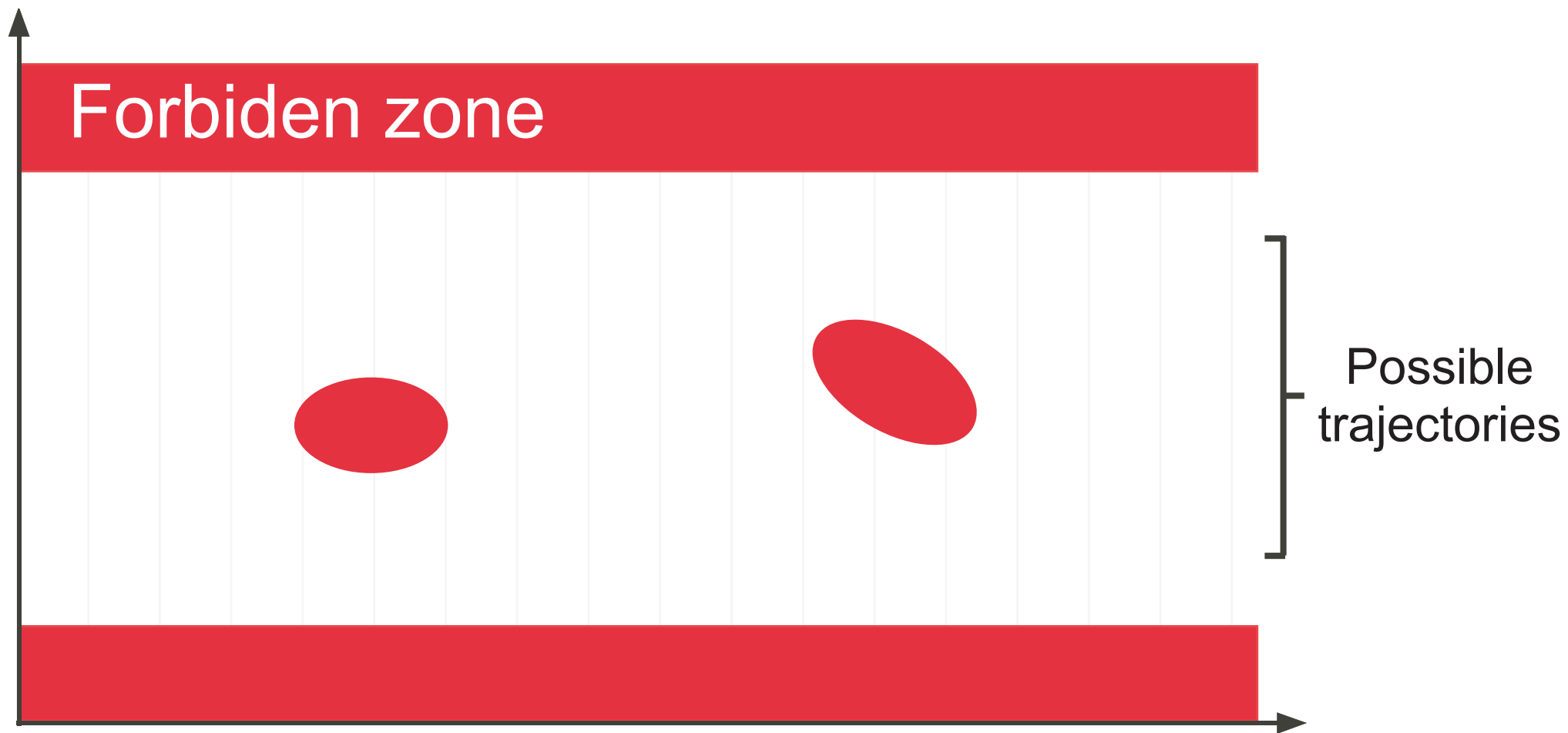
# Properties (Collecting semantics)

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



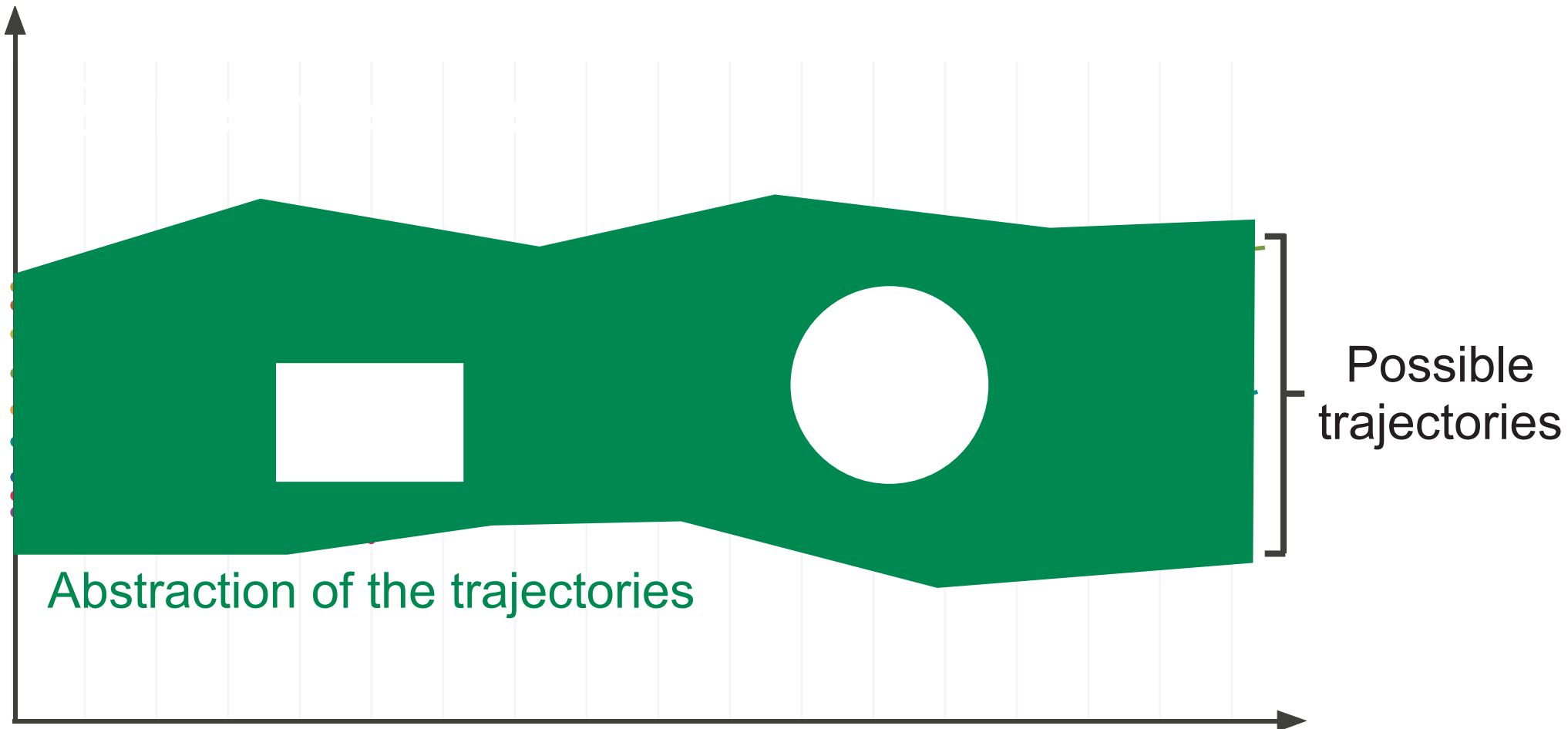
# Specification

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



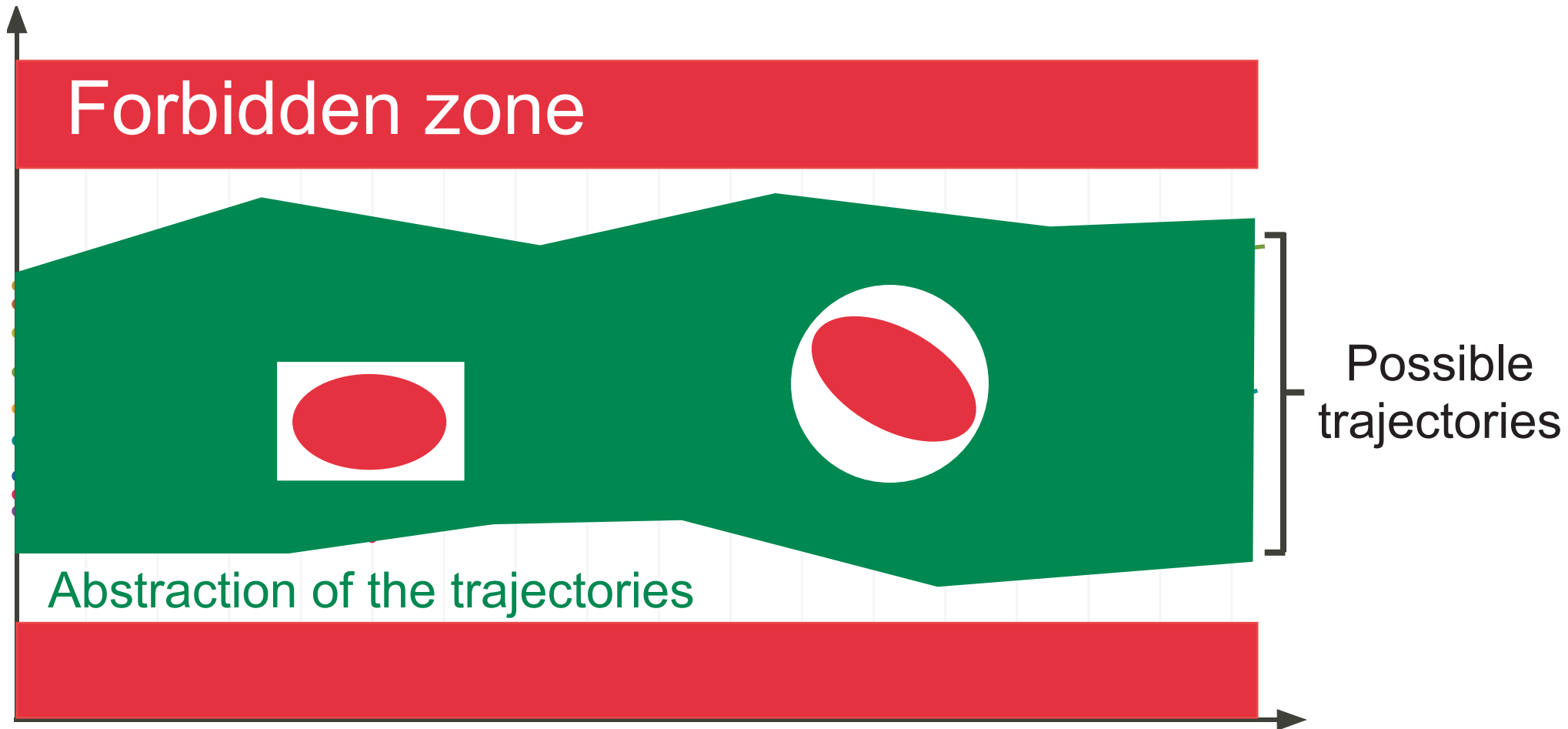
# Abstraction

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



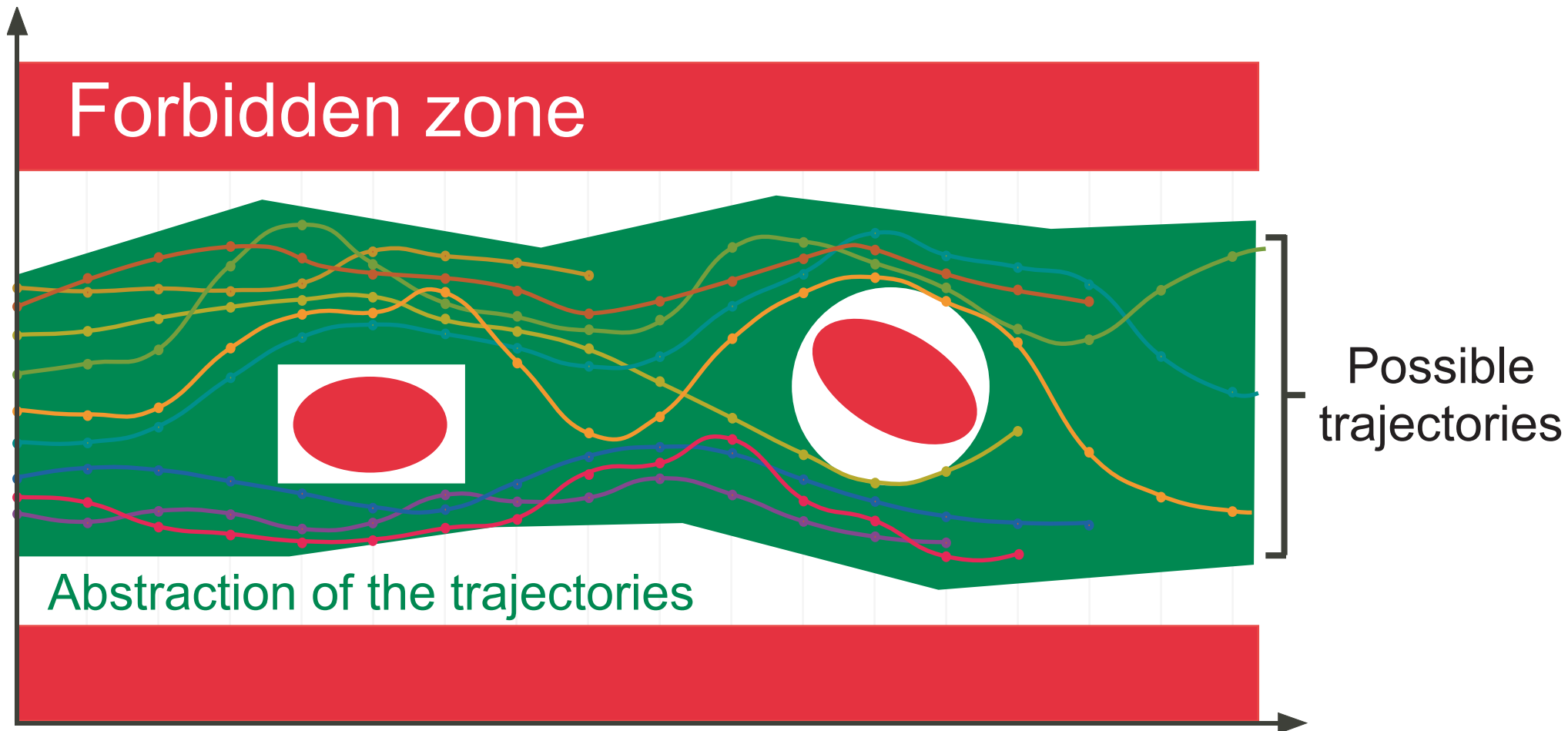
# Verification

*The proof is fully automatic*



# Soundness

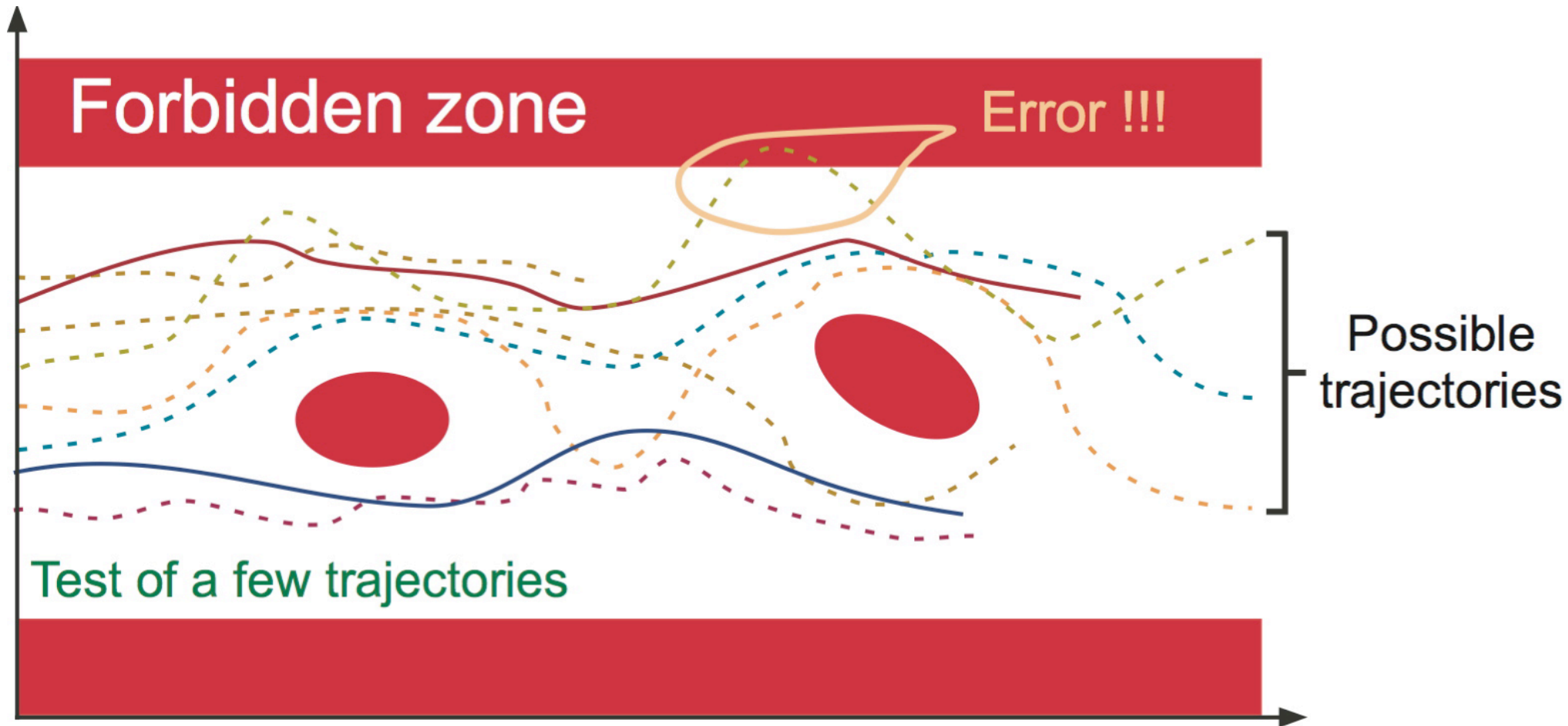
Never forget any possible case so the *abstract proof is correct in the concrete*





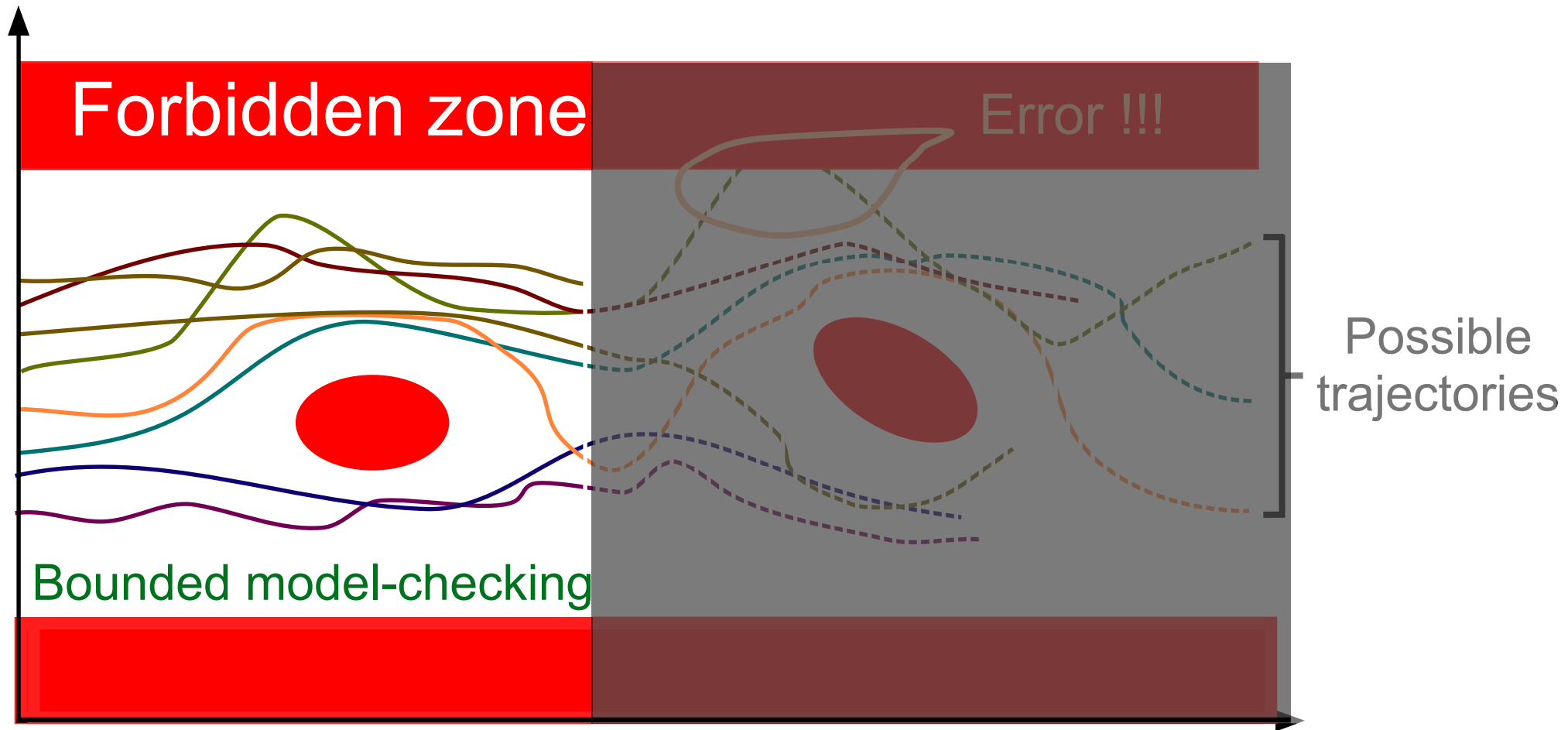
# Unsound methods: testing

*Try a few cases*



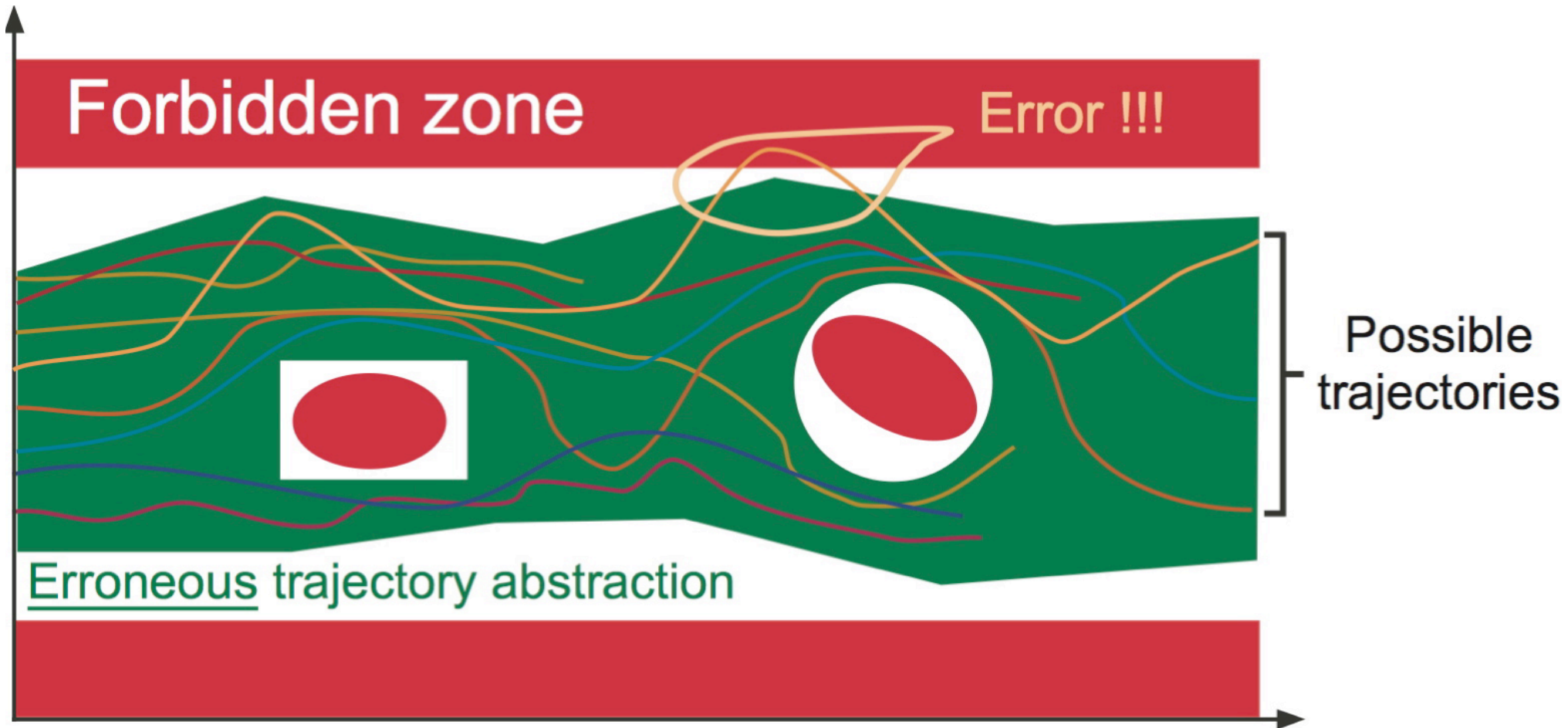
# Unsound methods: bounded model checking

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



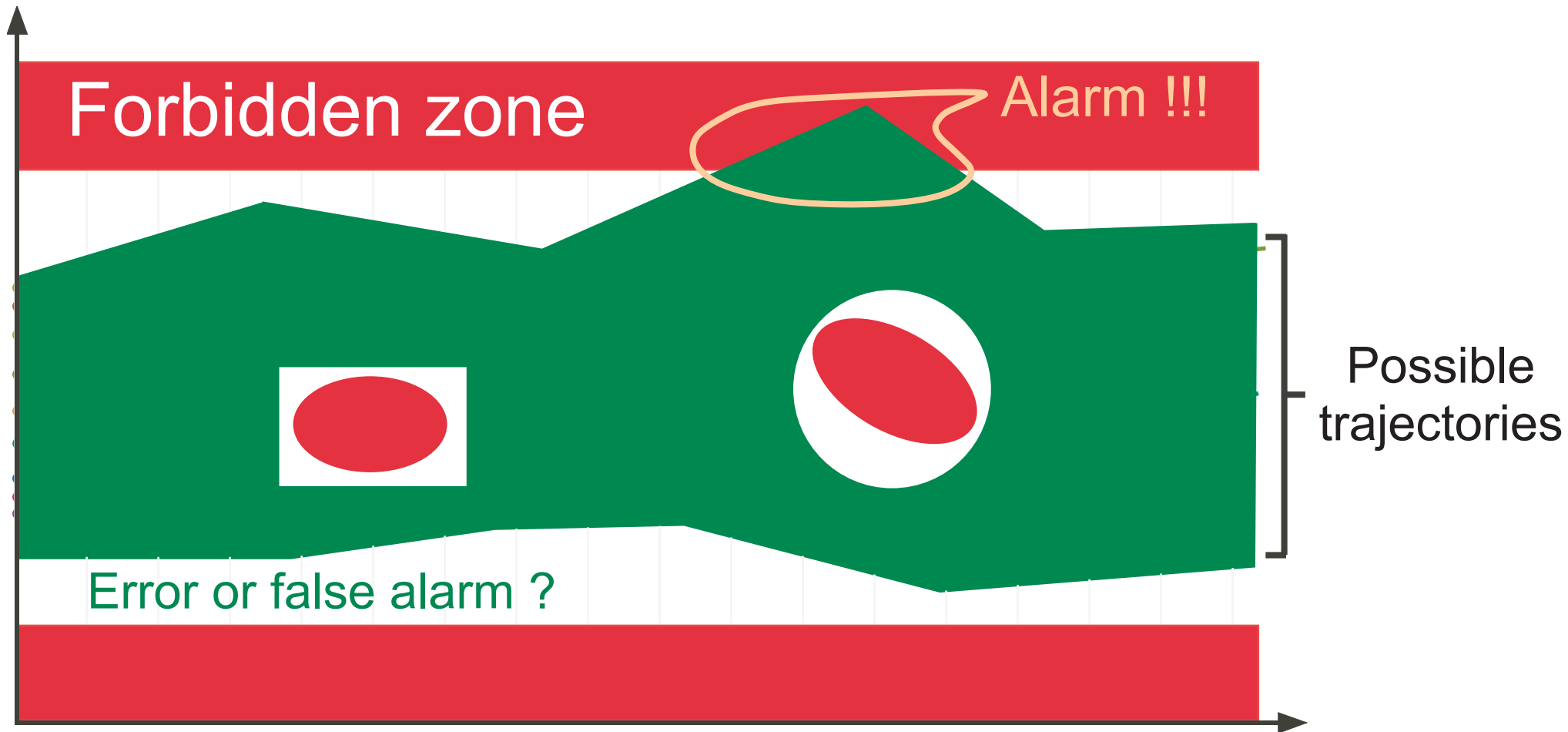
# Unsound methods: soundness

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



# Alarms

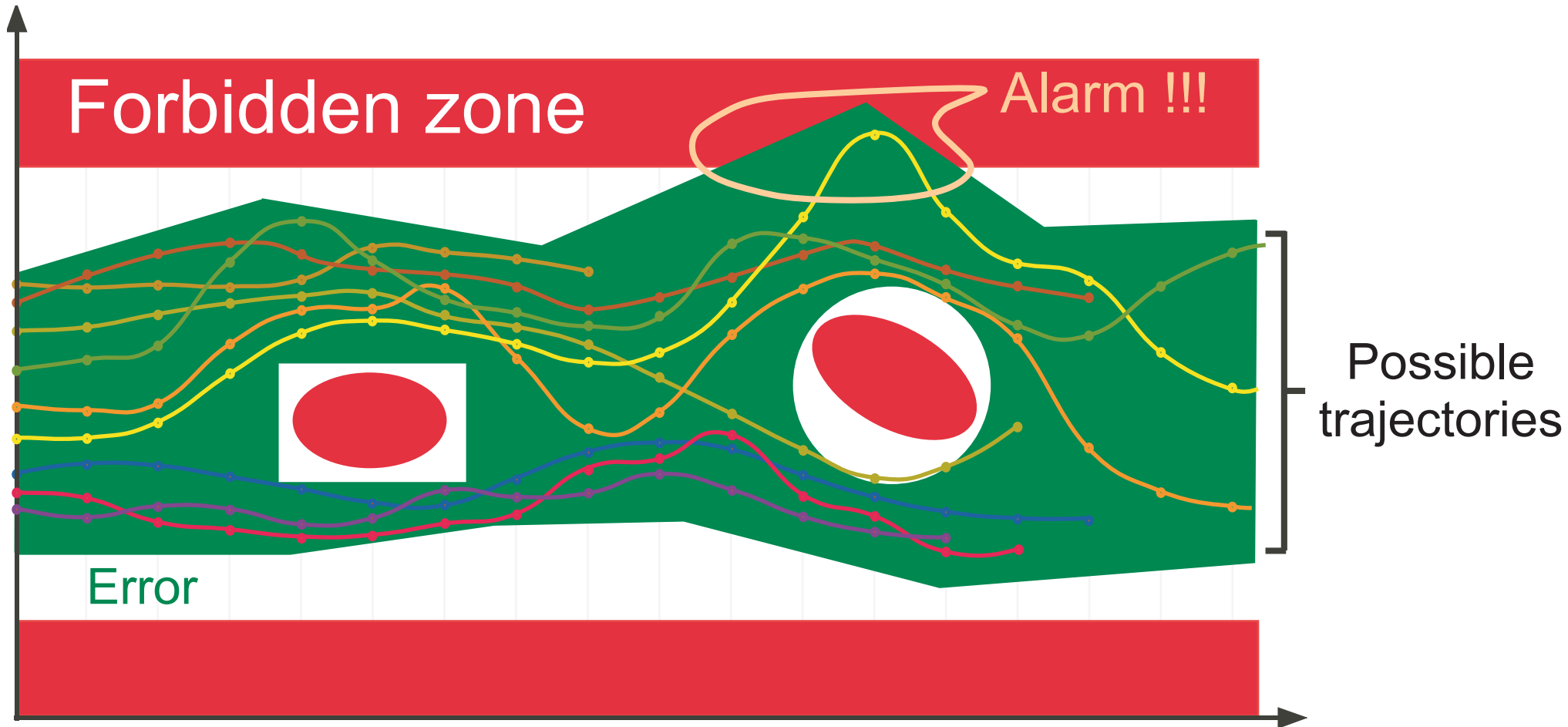
*When abstract proofs may fail while concrete proofs would succeed*



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

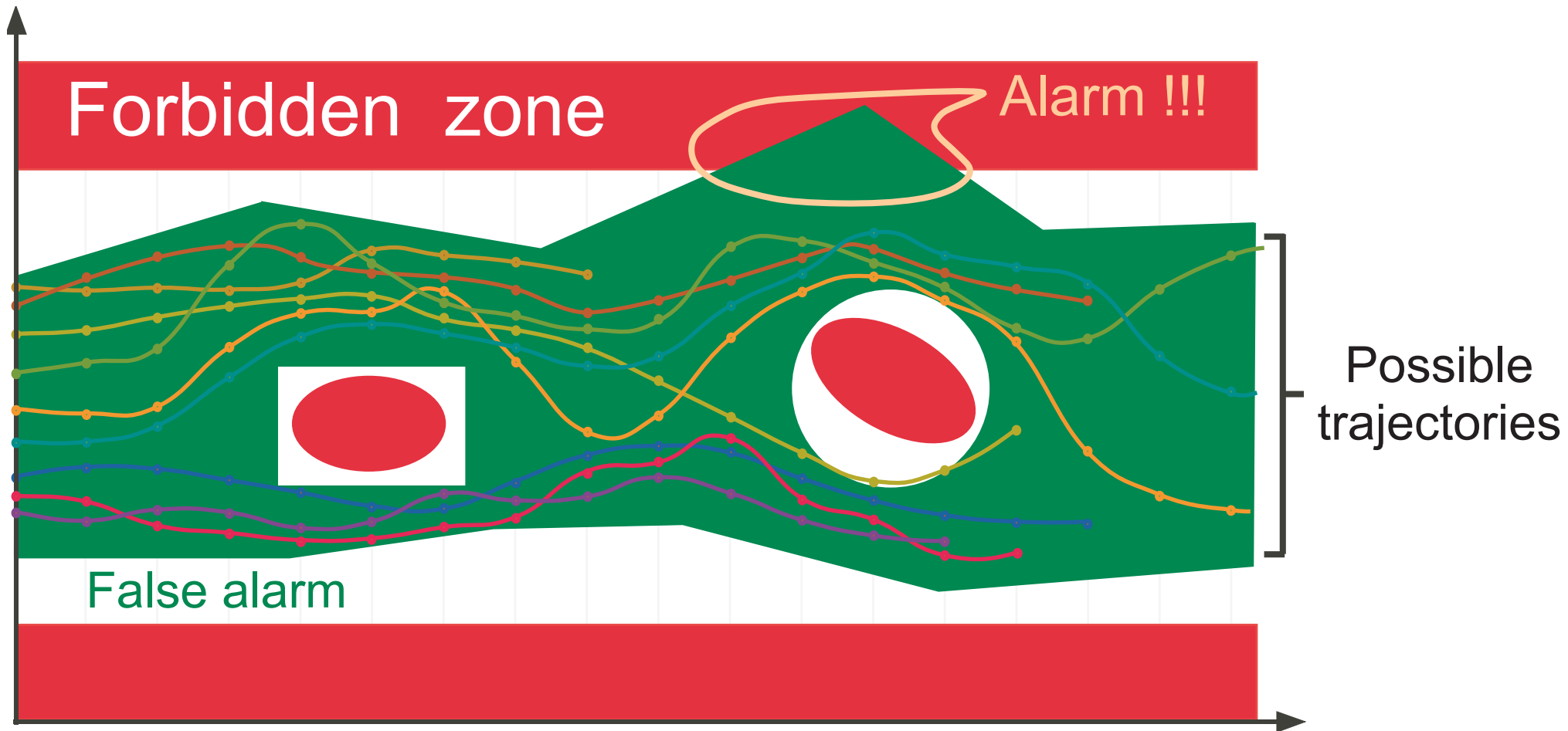
# True alarm

*The abstract alarm may correspond to a concrete error*



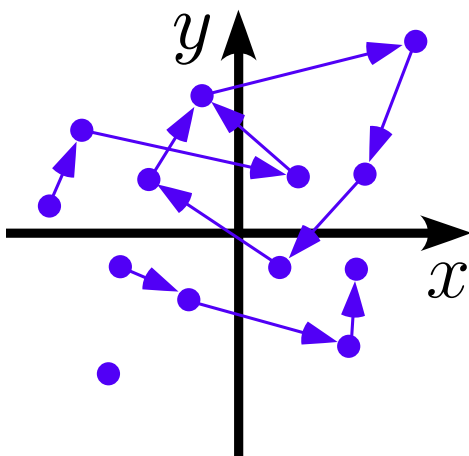
# False alarm

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

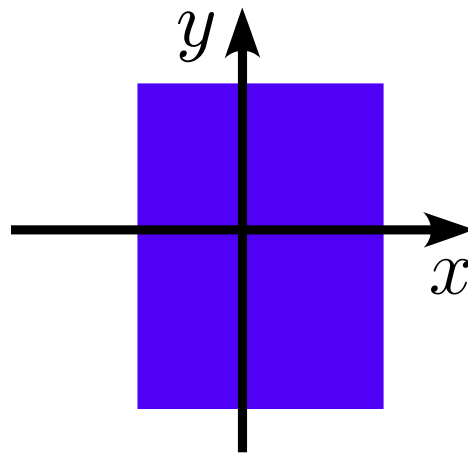


# What to do in presence of false alarms

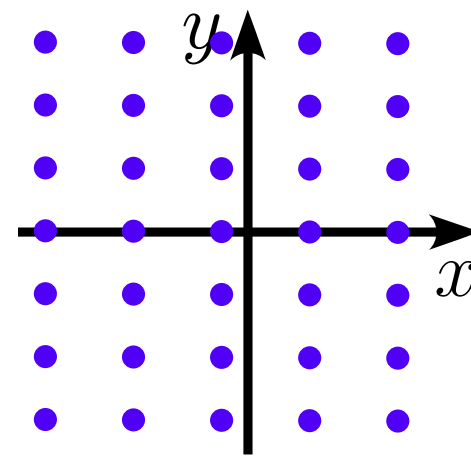
- False alarms are ultimately unavoidable (Gödel's incompleteness)
- Consider **finite** cases or **decidable cases** only (model-checking, *does not scale*)
- Ask for **human help** by providing information on the program behavior (theorem provers, SMT solvers), *program specific and labor costly*
-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



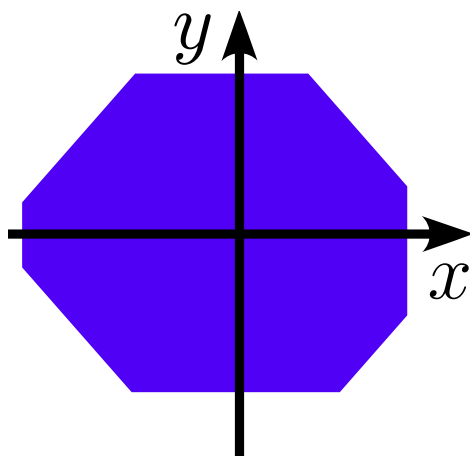
Collecting semantics:  
partial traces



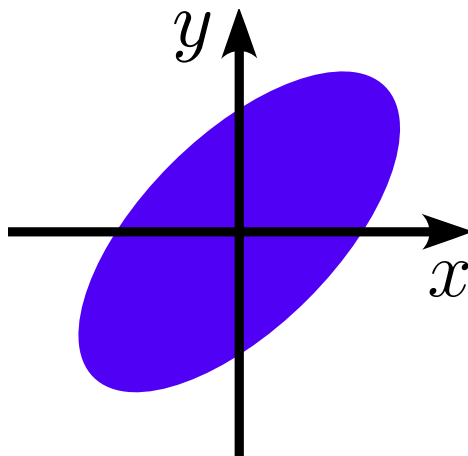
Intervals:  
 $x \in [a, b]$



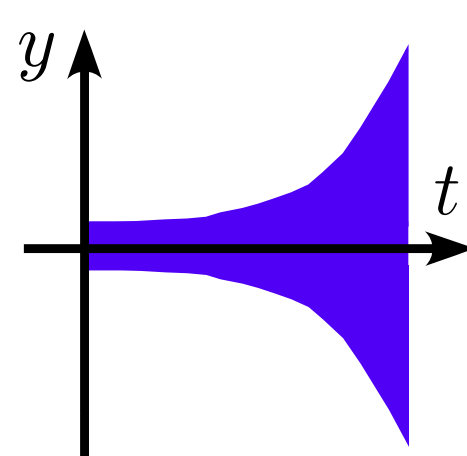
Simple congruences:  
 $x \equiv a[b]$



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



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



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

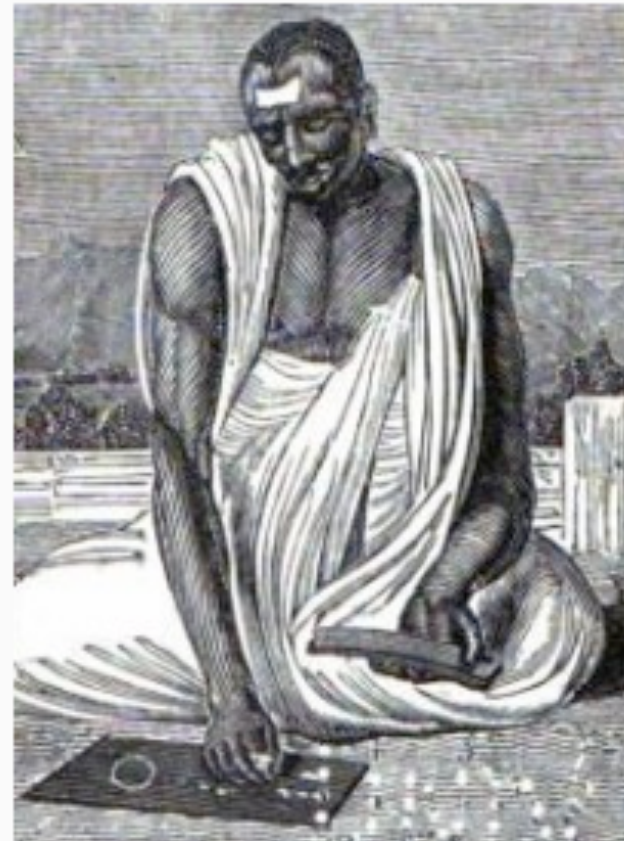


# The very first static analysis

# 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.

**Brahmagupta**



<b>Born</b>	598 CE
<b>Died</b>	c.670 CE
<b>Fields</b>	Mathematics, Astronomy
<b>Known for</b>	Zero, modern Number system

# 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 :

$\text{pos} + \text{pos} = \text{pos}$	$\text{pos} \times \text{pos} = \text{pos}$
$\text{pos} + \text{neg} = \text{pos}, \text{neg}$	$\text{pos} \times \text{neg} = \text{neg}$
$\text{neg} + \text{pos} = \text{pos}, \text{neg}$	$\text{neg} \times \text{pos} = \text{neg}$
$\text{neg} + \text{neg} = \text{neg}$	$\text{neg} \times \text{neg} = \text{pos}$
$V(p+q) = V(p) + V(q)$	$V(p \times q) = V(p) \times V(q)$

$V(0) = V(1) = \dots = \text{pos}$

$V(-1) = V(-2) = \dots = \text{neg}$

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

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

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 :

$\text{pos} + \text{pos} = \text{pos}$	$\text{pos} \times \text{pos} = \text{pos}$
$\text{pos} + \text{neg} = \text{pos}, \text{neg}$	$\text{pos} \times \text{neg} = \text{neg}$
$\text{neg} + \text{pos} = \text{pos}, \text{neg}$	$\text{neg} \times \text{pos} = \text{neg}$
$\text{neg} + \text{neg} = \text{neg}$	$\text{neg} \times \text{neg} = \text{pos}$
$V(p+q) = V(p) + V(q)$	$V(p \times q) = V(p) \times V(q)$

$V(0) = V(1) = \dots = \text{pos}$

$V(-1) = V(-2) = \dots = \text{neg}$

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

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

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 :

$\text{pos} + \text{pos} = \text{pos}$

$\text{pos} + \text{neg} = \text{pos}, \text{neg}$

$\text{neg} + \text{pos} = \text{pos}, \text{neg}$

$\text{neg} + \text{neg} = \text{neg}$

$V(p+q) = V(p) + V(q)$

$V(0) = V(1) = \dots = \text{pos}$

$V(-1) = V(-2) = \dots = \text{neg}$

$\text{pos} \times \text{pos} = \text{pos}$

$\text{pos} \times \text{neg} = \text{neg}$

$\text{neg} \times \text{pos} = \text{neg}$

$\text{neg} \times \text{neg} = \text{pos}$

$V(p \times q) = V(p) \times V(q)$

wrong

$0 \in \text{pos} \times -1 \in \text{neg}$   
 $= 0 \notin \text{neg}$

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

$V(a) = \text{pos}, \text{neg}$

$V(b) = \text{pos}, \text{neg}$

$V(a \times a) = \text{pos} \times \text{pos}, \text{neg} \times \text{neg}$

$V(b \times b) = \text{pos} \times \text{pos}, \text{neg} \times \text{neg}$

$= \text{pos}, \text{pos} = \text{pos}$

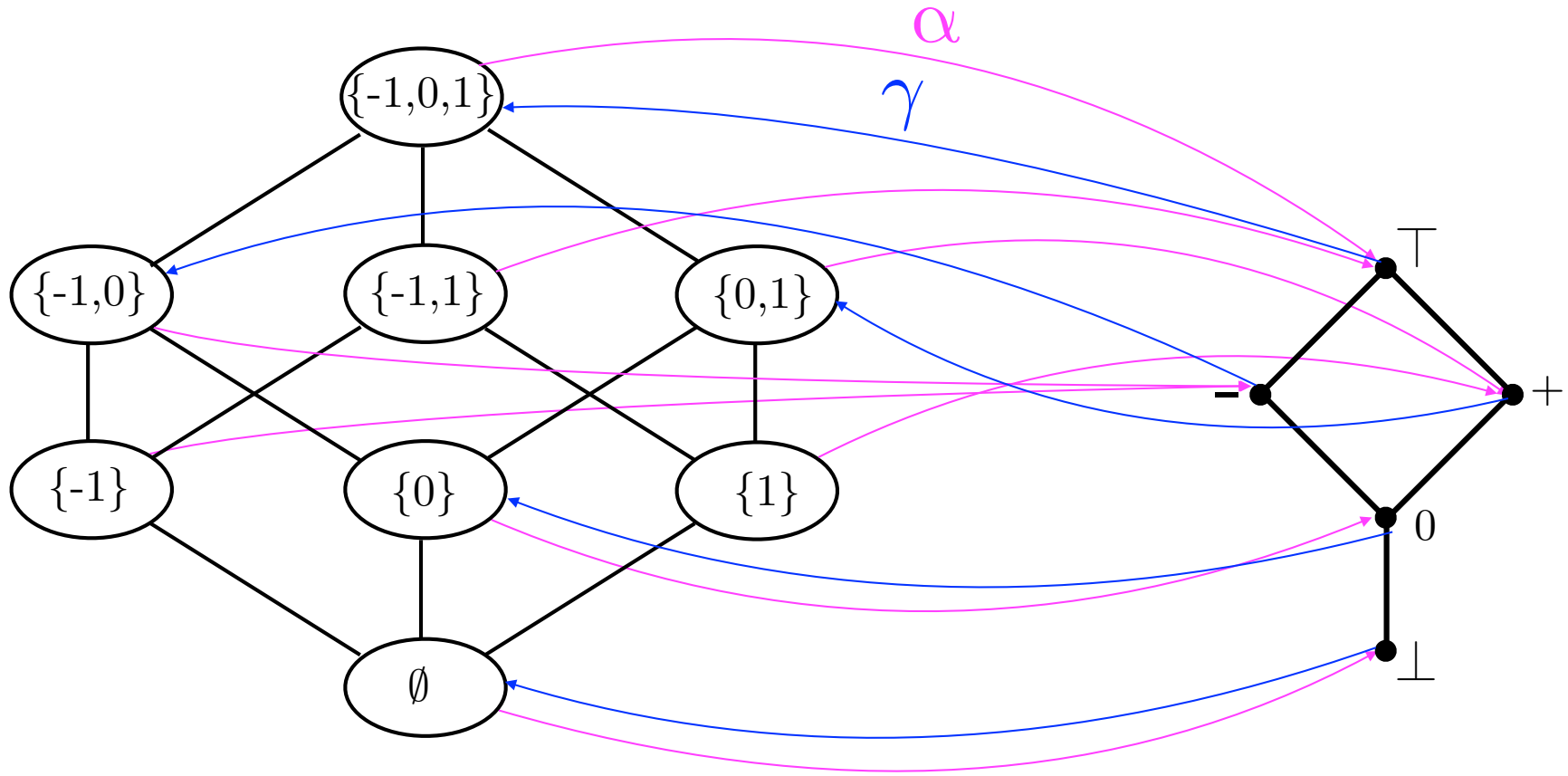
$= \text{pos}, \text{pos} = \text{pos}$

$V(a \times a + b \times b) = V(a \times a) + V(b \times b) = \text{pos} + \text{pos} = \text{pos}$

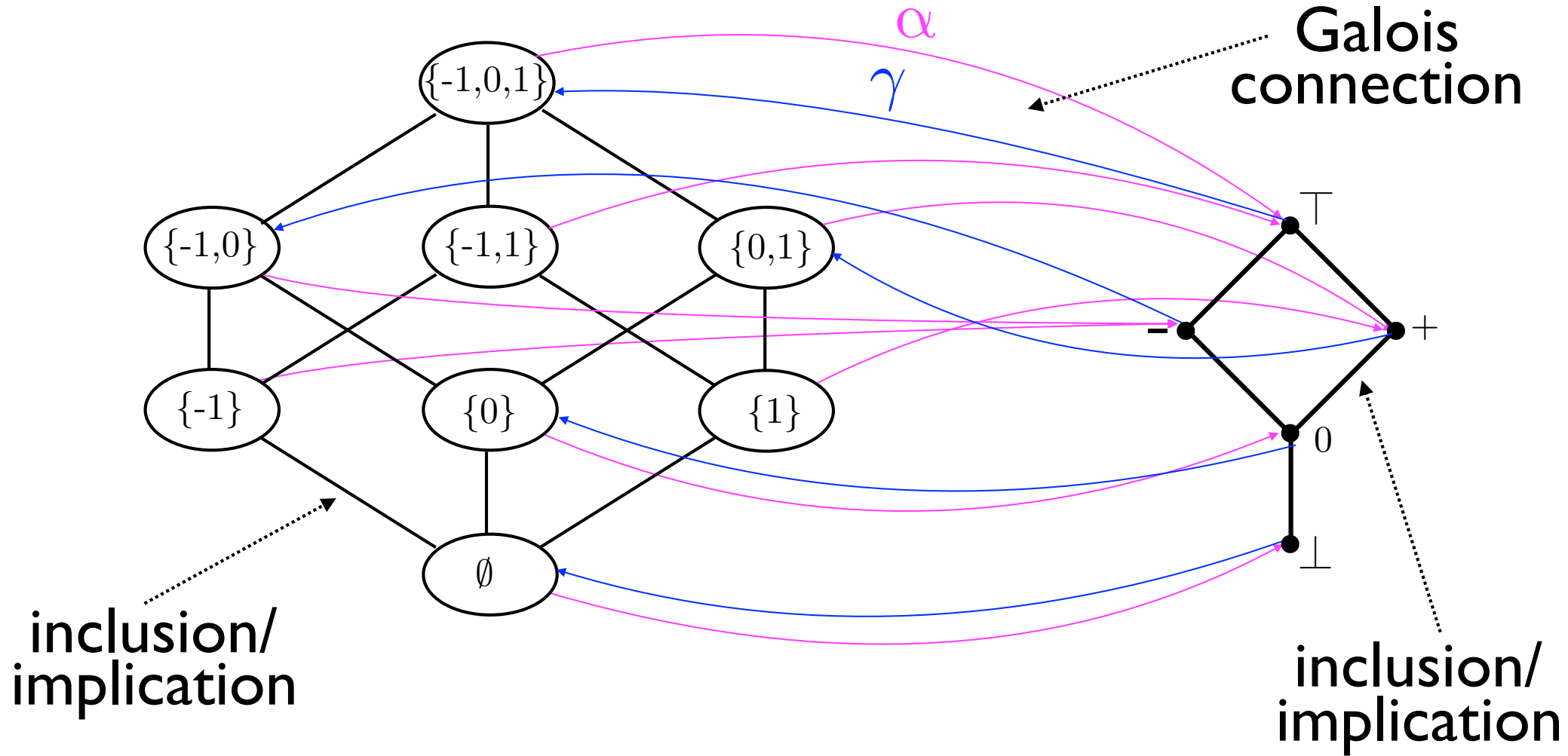
This valuation proves that the result of  $a \times a + b \times b$  is always positive and hence allows to compute its square root without any preliminary dynamic test on its sign. On the other hand, the



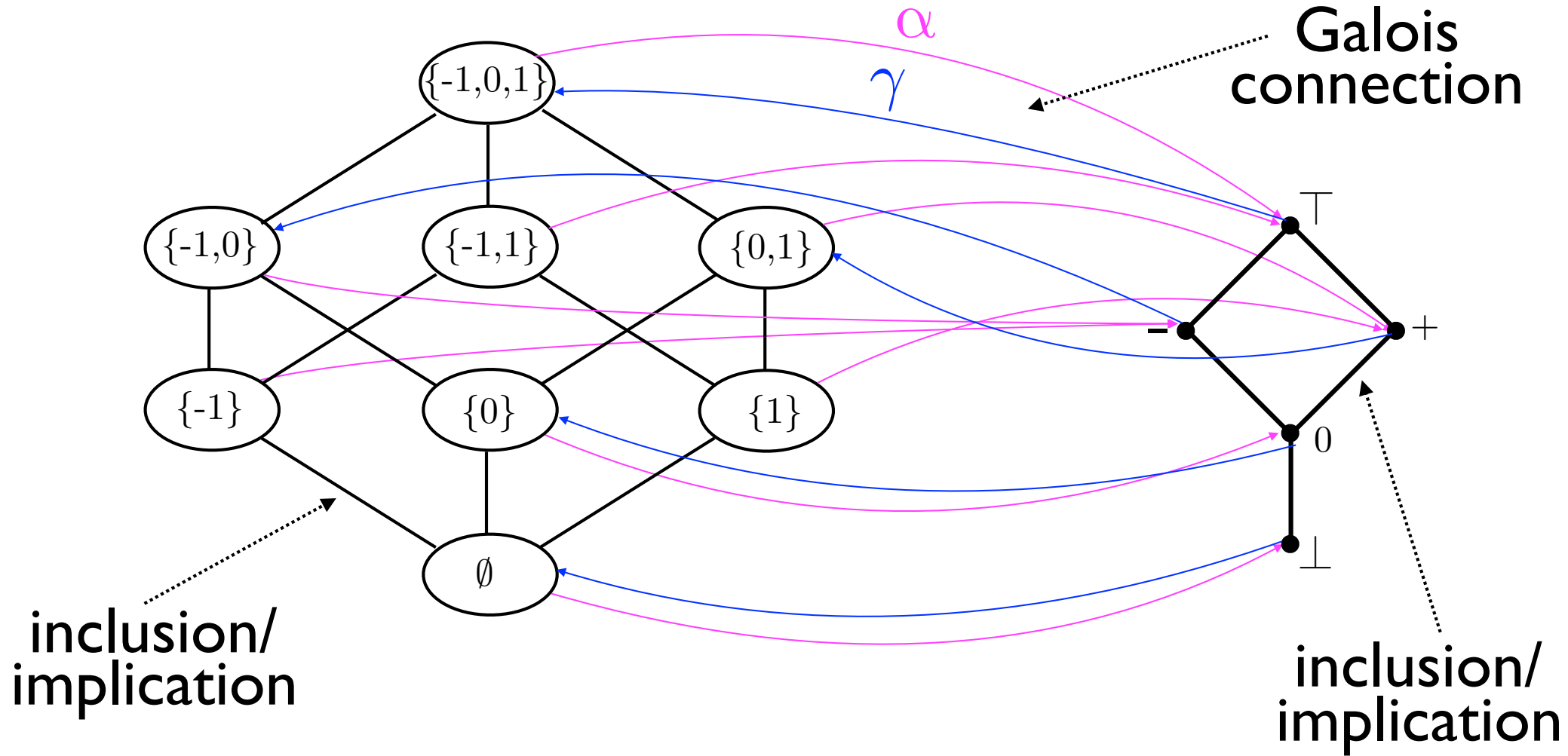
# The rule of signs Cousot & Cousot (1979)



# The rule of signs Cousot & Cousot (1979)



# The rule of signs Cousot & Cousot (1979)



calculational design method

$$\begin{array}{c}
 + \quad + \quad + \\
 \downarrow \gamma \quad \downarrow \gamma \\
 \{0,1\} \quad + \quad \{0,1\} \\
 \uparrow \alpha \\
 +
 \end{array}
 = \{0,1,2[2]\} = \{0,1\}$$



# Application of abstract interpretation to static analysis

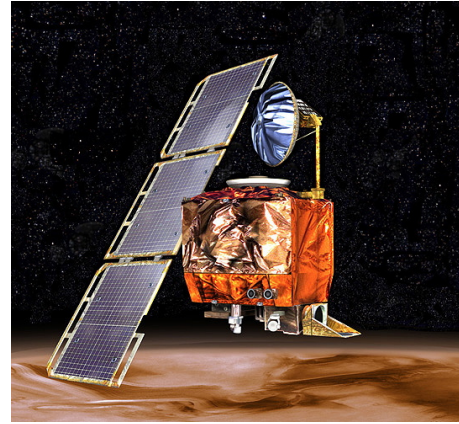
# All computer scientists have experienced bugs



Ariane 5.01 failure  
(overflow)



Patriot failure  
(float rounding)



Mars orbiter loss  
(unit error)

```
unsigned int payload = 18; /* Sequence number + random bytes */
unsigned int padding = 16; /* Use minimum padding */

/* Check if padding is too long, payload and padding
 * must not exceed 2^14 - 1 = 16381 bytes in total.
 */
OPENSSL_assert(payload + padding <= 16381);

/* Create HeartBeat message, we just use a sequence number
 * as payload to distinguish different messages and add
 * some random stuff.
 * - Message Type, 1 byte
 * - Payload Length, 2 bytes (unsigned int)
 * - Payload, the sequence number (2 bytes uint)
 * - Payload, random bytes (16 bytes uint)
 * - Padding
 */

buf = OPENSSL_malloc(1 + 2 + payload + padding);
p = buf;
/* Message Type */
*pp++ = TLS1_HB_REQUEST;
/* Payload length (18 bytes here) */
s2n(payload, p);
/* Sequence number */
s2n(s->tlsextr.hb_seq, p);
/* 16 random bytes */
RAND_pseudo_bytes(p, 16);
p += 16;
/* Random padding */
RAND_pseudo_bytes(p, padding);

ret = dtls1_write_bytes(s, TLS1_RT_HEARTBEAT, buf, 3 + payload + padding);
```

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;
}
```

Floyd-Naur-Hoare verification conditions:

$$(y \geq 0 \wedge x = y) \implies I(x, y)$$

*initialisation*

$$(I(x, y) \wedge x > 0 \wedge x' = x - 1) \implies I(x', y)$$

*iteration*

Equivalent fixpoint equation:

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

$$(i.e. I = F(I)^{(5)})$$

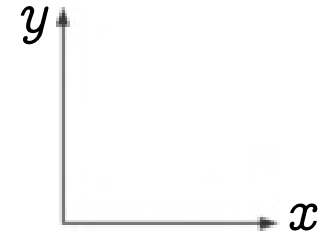
---

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

# Iterates

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

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

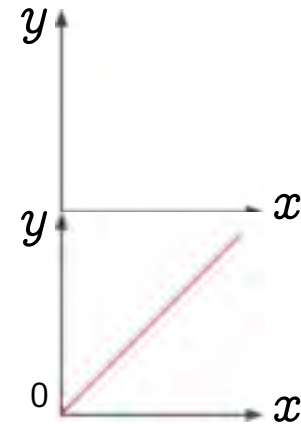


# Iterates

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

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

$$\begin{aligned} I^1(x, y) &= x \geq 0 \wedge (x = y \vee I^0(x + 1, y)) \\ &= 0 \leq x = y \end{aligned}$$



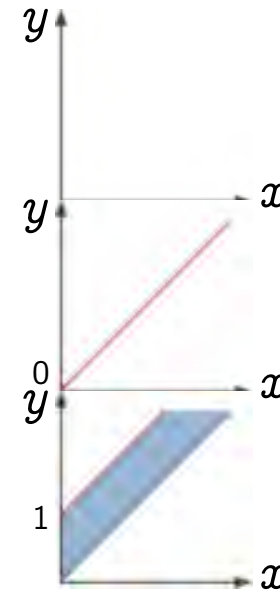
# Iterates

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

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

$$\begin{aligned} I^1(x, y) &= x \geq 0 \wedge (x = y \vee I^0(x + 1, y)) \\ &= 0 \leq x = y \end{aligned}$$

$$\begin{aligned} I^2(x, y) &= x \geq 0 \wedge (x = y \vee I^1(x + 1, y)) \\ &= 0 \leq x \leq y \leq x + 1 \end{aligned}$$



# Iterates

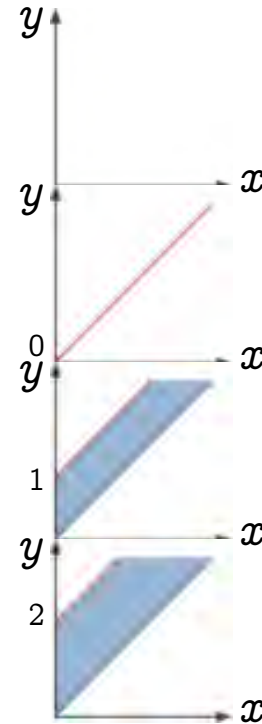
Iterates  $I = \lim_{n \rightarrow \infty} F^n(\text{false})$

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

$$\begin{aligned} I^1(x, y) &= x \geq 0 \wedge (x = y \vee I^0(x + 1, y)) \\ &= 0 \leq x = y \end{aligned}$$

$$\begin{aligned} I^2(x, y) &= x \geq 0 \wedge (x = y \vee I^1(x + 1, y)) \\ &= 0 \leq x \leq y \leq x + 1 \end{aligned}$$

$$\begin{aligned} I^3(x, y) &= x \geq 0 \wedge (x = y \vee I^2(x + 1, y)) \\ &= 0 \leq x \leq y \leq x + 2 \end{aligned}$$





# Convergence acceleration: widening

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

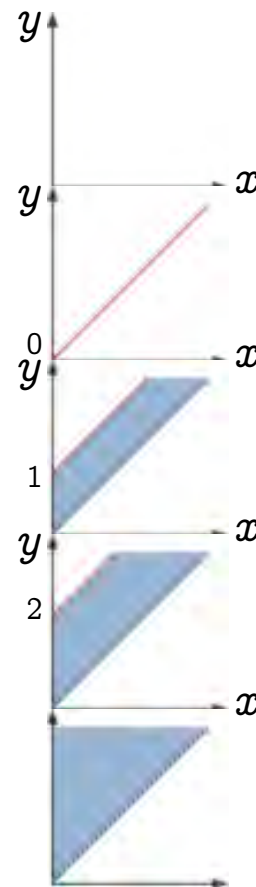
$$I^0(x, y) = \text{false}$$

$$\begin{aligned} I^1(x, y) &= x \geq 0 \wedge (x = y \vee I^0(x + 1, y)) \\ &= 0 \leq x = y \end{aligned}$$

$$\begin{aligned} I^2(x, y) &= x \geq 0 \wedge (x = y \vee I^1(x + 1, y)) \\ &= 0 \leq x \leq y \leq x + 1 \end{aligned}$$

$$\begin{aligned} I^3(x, y) &= x \geq 0 \wedge (x = y \vee I^2(x + 1, y)) \\ &= 0 \leq x \leq y \leq x + 2 \end{aligned}$$

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



# Fixed point

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

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

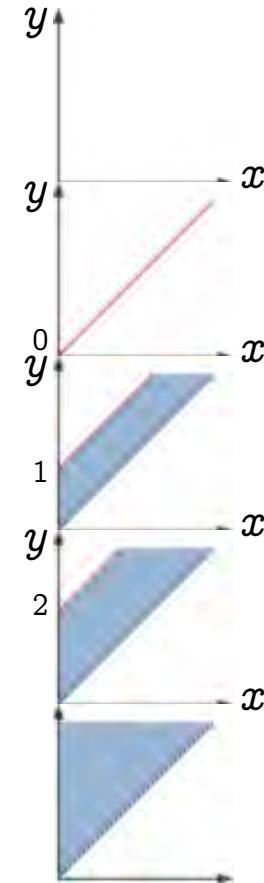
$$\begin{aligned} I^1(x, y) &= x \geq 0 \wedge (x = y \vee I^0(x + 1, y)) \\ &= 0 \leq x = y \end{aligned}$$

$$\begin{aligned} I^2(x, y) &= x \geq 0 \wedge (x = y \vee I^1(x + 1, y)) \\ &= 0 \leq x \leq y \leq x + 1 \end{aligned}$$

$$\begin{aligned} I^3(x, y) &= x \geq 0 \wedge (x = y \vee I^2(x + 1, y)) \\ &= 0 \leq x \leq y \leq x + 2 \end{aligned}$$

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

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



# Octagons

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

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

$$\begin{aligned} I^1(x, y) &= x \geq 0 \wedge (x = y \vee I^0(x + 1, y)) \\ &= 0 \leq x = y \end{aligned}$$

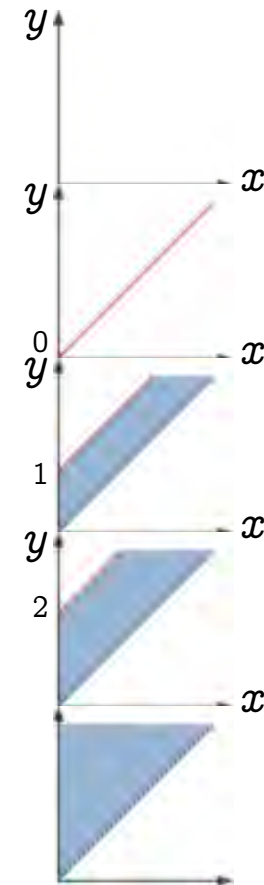
$$\begin{aligned} I^2(x, y) &= x \geq 0 \wedge (x = y \vee I^1(x + 1, y)) \\ &= 0 \leq x \leq y \leq x + 1 \end{aligned}$$

$$\begin{aligned} I^3(x, y) &= x \geq 0 \wedge (x = y \vee I^2(x + 1, y)) \\ &= 0 \leq x \leq y \leq x + 2 \end{aligned}$$

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

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

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)

---

(2) 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
...
```

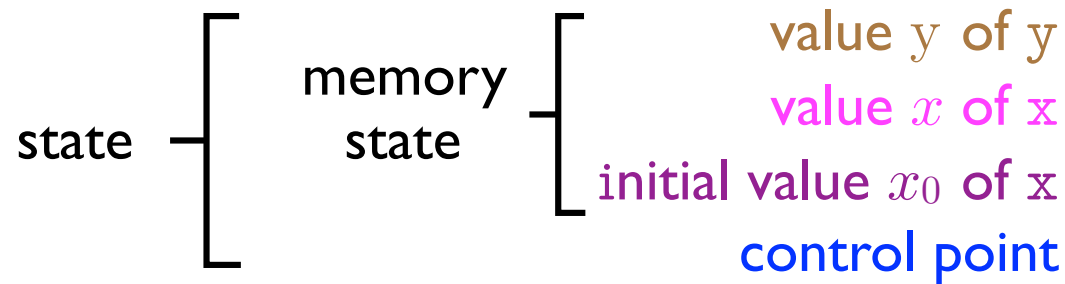
$\langle 1:, 3, 3, 3 \rangle \rightarrow \langle 2:, 3, 3, 3 \rangle \rightarrow \langle 3:, 3, 2, 3 \rangle \rightarrow \langle 1:, 3, 2, 5 \rangle \rightarrow \langle 2:, 3, 2, 5 \rangle$   
 $\rightarrow \langle 3:, 3, 1, 5 \rangle \rightarrow \langle 1:, 3, 1, 7 \rangle \rightarrow \langle 2:, 3,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

# 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); }
```

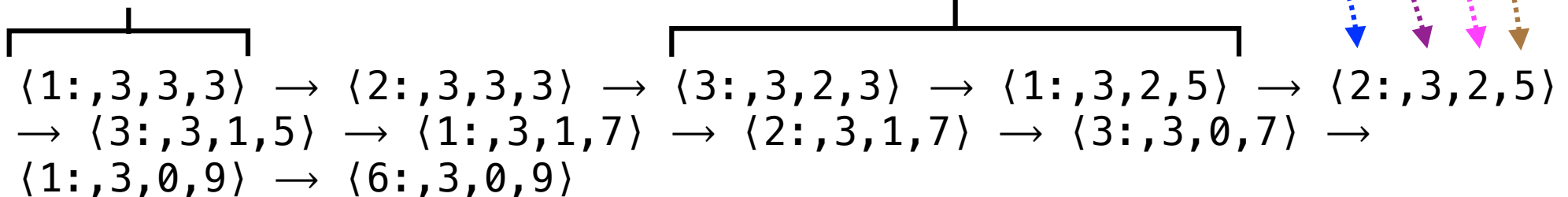
```
Enter an integer: 3
x = 3, y = 3
x = 2, y = 5
x = 1, y = 7
x = 0, y = 9
```

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



initial state  $\in \text{init} [P]$

transition  $\in \text{trans} [P]$



# 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 whose prefixes are all partial traces

# Fixpoint partial trace semantics

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

# 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:**  $S^i[[P]] = \alpha(S^t[[P]])$

# Invariance abstraction

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

```

#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); }

```

Abstraction of the code above:

- $\{ \langle x_0, x, y \rangle \mid y = 3x_0 - 2x \}$  ← `/* 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

# General idea

- All known semantics are abstractions of a most precise semantics

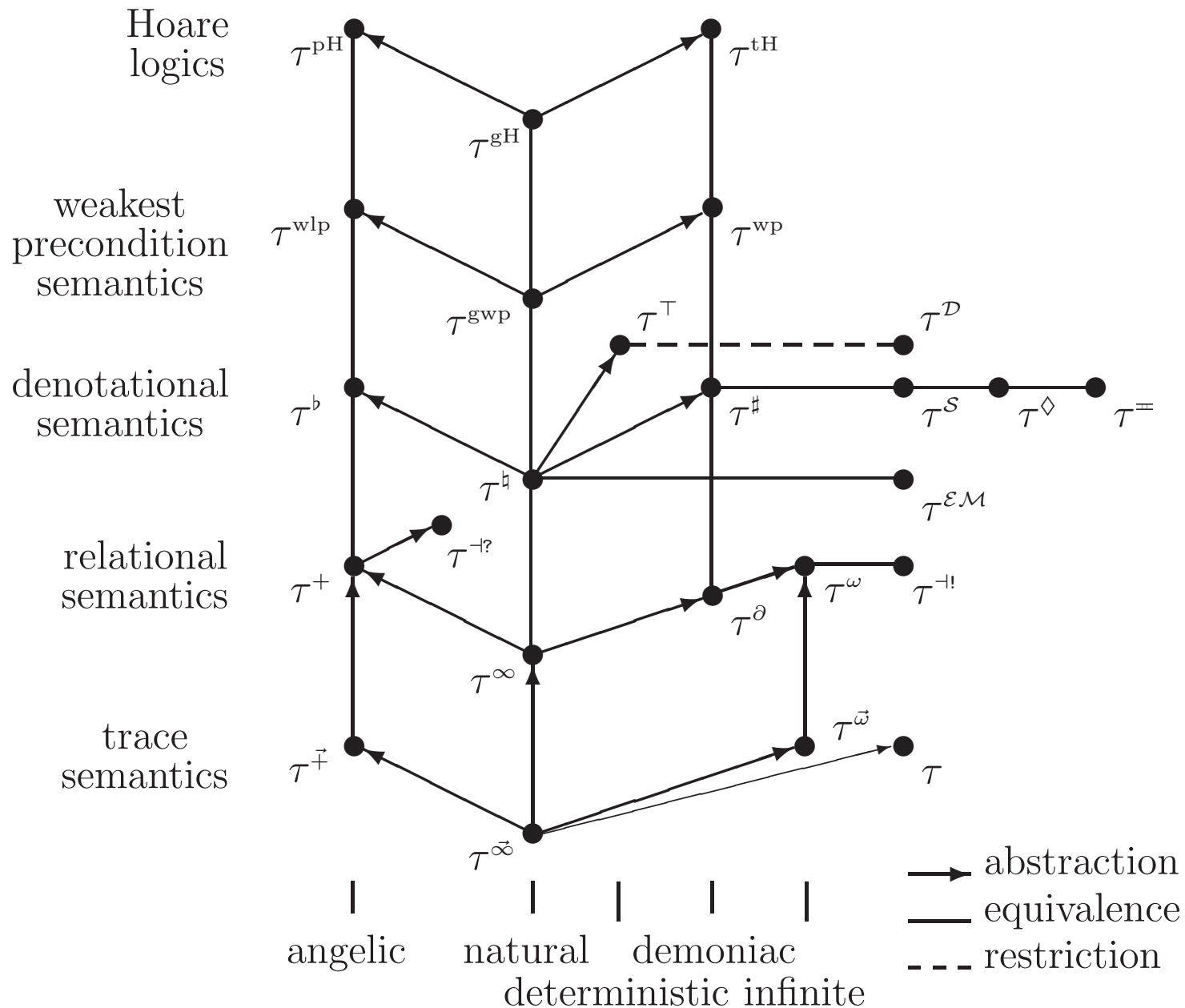
# Abstraction to denotational semantics

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

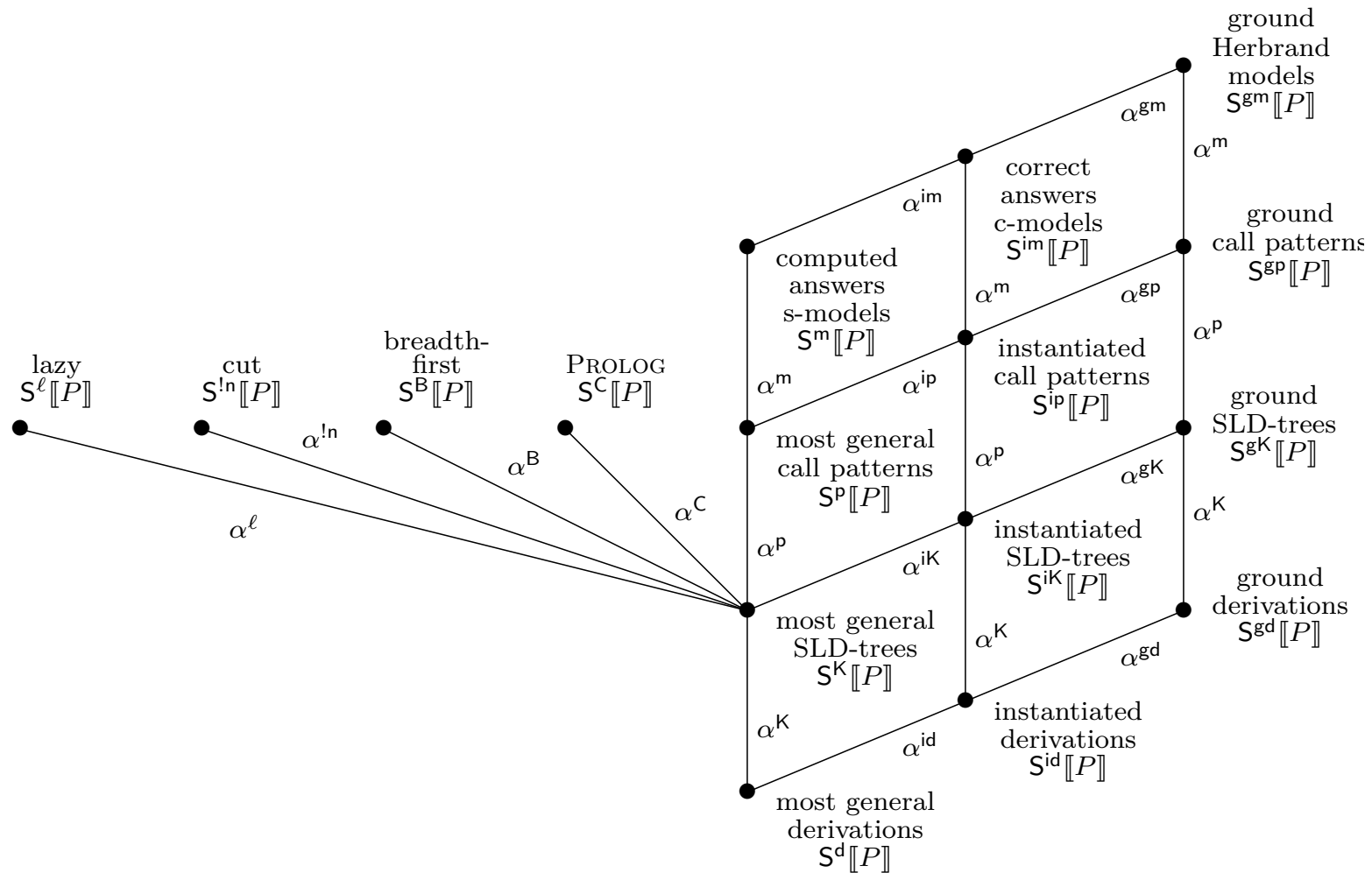
*i.e.* a map of initial states to the set of final states plus  $\perp$  in case of non-termination



# Hierarchy of abstractions



# idem for Prolog



- all semantics are abstractions of  $S^d[[P]]$

# Conclusion

# Abstract interpretation

- A well-developed **theory**, still in progress
- Active **research** e.g.
  - abstract domains to handle e.g. complex data structures
  - abstraction of parallelism with weak memory models
  - applications to biology, ...
- Industrial-quality **static analyzers**

# Industrialisation: Astrée

**Example 1: scenarios**

Project Analysis Editors Edit Tools Help

**Analyzed file: db/invalid/path/scenarios.c**

```

70 /*
71  * Type cast causing overflow.
72  */
73 s = SPEED_SENSOR;
74
75 /*
76  * Precise handling of pointer arithmetics.
77  */
78 ptr = &ArrayBlock[0];
79
80 if (uninitialized_1) {
81   ArrayBlock[15] = 0x15; // easy case
82 }
83
84 if (uninitialized_2) {
85   *(ptr + 15) = 0x10; // hard case
86 }
87
88 /*
89  * Precise handling of compute-through-overflow and
90  * Note that, by default, alarms on explicit typeca
91  * deactivated (see Options->General tab).
92  */
93 z = (short)((unsigned short)vx + (unsigned short)vy
94   __ASTREE_assert((-2<=z && z<=2));
95
96 */

```

**Original source: src/scenarios.c**

```

66 /*
67  * Type cast causing overflow.
68  */
69 s = SPEED_SENSOR;
70
71 /*
72  * Precise handling of pointer arithmetics.
73  */
74 ptr = &ArrayBlock[0];
75
76 if (uninitialized_1) {
77   ArrayBlock[15] = 0x15; // easy case
78 }
79
80 if (uninitialized_2) {
81   *(ptr + 15) = 0x10; // hard case
82 }
83
84 /*
85  * Precise handling of compute-through-overflow and
86  * Note that, by default, alarms on explicit typeca
87  * deactivated (see Options->General tab).
88  */
89 z = (short)((unsigned short)vx + (unsigned short)vy
90   __ASTREE_assert((-2<=z && z<=2));
91
92 */

```

Line 81, column 7      Line 77, column 1

[ call#main@58 at scenarios.c:58.0-130.1  
if@80=true at scenarios.c:80.4-82.5  
ALARM (A): invalid dereference: dereferencing 1 byte(s) at offset(s) 15 may overflow the variable ArrayBlock of byte-size 10 at scenarios.c:81.6-20 ]  
ERROR: Definite runtime error during assignment in this context. Analysis stopped for this context.

Filter:      Type filters      Comment filters      8 of 8 findings visible      Show unused comments      Generate Annotations

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 incl
5	Definite Alarm (A)	Possible overflow upon dereference	# scenarios.c:81.6-20		invalid dereference: dereferencing 1 b
6	Alarm (A)	Use of uninitialized variables	# scenarios.c:84.8-23		uninitialized read: reading 4 byte(s) at
7	Definite Alarm (A)	Possible overflow upon dereference	# scenarios.c:85.6-17		invalid dereference: dereferencing 1 b
8	Alarm (A)	Assertion failure	# scenarios.c:127.4-40		assert failure __ASTREE_assert((second

Project Summary      Resource Monitor

**Errors:** 2

**Alarms on code locations**

Run-time errors: 7

Data flow anomalies: 0

Rule violations: 0

**Alarms on memory locations**

Data races: 0

**Reached code:** 100%

**Duration:** 20s

Output      Findings      Not reached      Watch      Search

# Industrialisation: Astrée

The screenshot displays the Astrée static analysis tool interface for a project named "Example 1: scenarios (1)". The interface is divided into several sections:

- Configuration:** Shows the analysis configuration, including the Preprocessor, Parser, Analyzer, and Annotations.
- Results:** The "Overview" tab is selected, showing a summary of findings. A pie chart titled "Alarms (7 findings)" indicates that 42.9% of the findings are Alarms.
- Files:** A list of files is shown, including "astree.cfg" and "scenarios.c" with its sub-files: "main", "msg1", "msg2", "registerMsg", and "sendMsg".
- Findings:** A list of findings is displayed, categorized by type and name. The findings include:
  - 7 Alarms
  - 3 Invalid usage of pointers and arrays
    - 1 Out-of-bound array access
    - 2 Possible overflow upon dereference
  - 1 Invalid ranges and overflows
  - 1 Overflow in conversion
  - 1 Failed or invalid directives
  - 1 Assertion failure
  - 2 Uninitialized variables
  - 2 Use of uninitialized variables
  - 2 Errors
- Findings Table:** A detailed table of findings is shown at the bottom, with columns for Type, Category, Location, Classification, and Comment. The table lists 8 findings, including a notification about invalid conversion and several alarms related to overflow, uninitialized variables, and out-of-bound array access.

# 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 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