49th IEEE Conference on Decision and Control
Atlanta, GA
Pre-Conference Workshop on Verification of Control Systems
December 14, 2010

# Verification of Control Systems by Abstract Interpretation

Patrick Cousot
CIMS-NYU & ENS

pcousot@cs.nyu.edu
cousot@ens.fr

cs.nyu.edu/~pcousot
di.ens.fr/~cousot

# Motivation

# All computer scientists have experienced bugs

Ariane 5.01 failure
(overflow error)

Patriot failure
(float rounding error)

Mars orbiter loss
(unit error)

Russian Proton-M rocket
carrying 3 Glonass-M satellites
(unknown programming error)

- Checking the presence of bugs is great
- Proving their absence is even better!

# Abstract interpretation

Patrick Cousot & Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238—252, Los Angeles, California, 1977. ACM Press, New York, NY, USA

Patrick Cousot & Radhia Cousot. Systematic design of program analysis frameworks. In *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* pages 269—282, San Antonio, Texas, 1979. ACM Press, New York, U.S.A.

Patrick Cousot & Radhia Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation,* 2(4):511—547, August 1992.

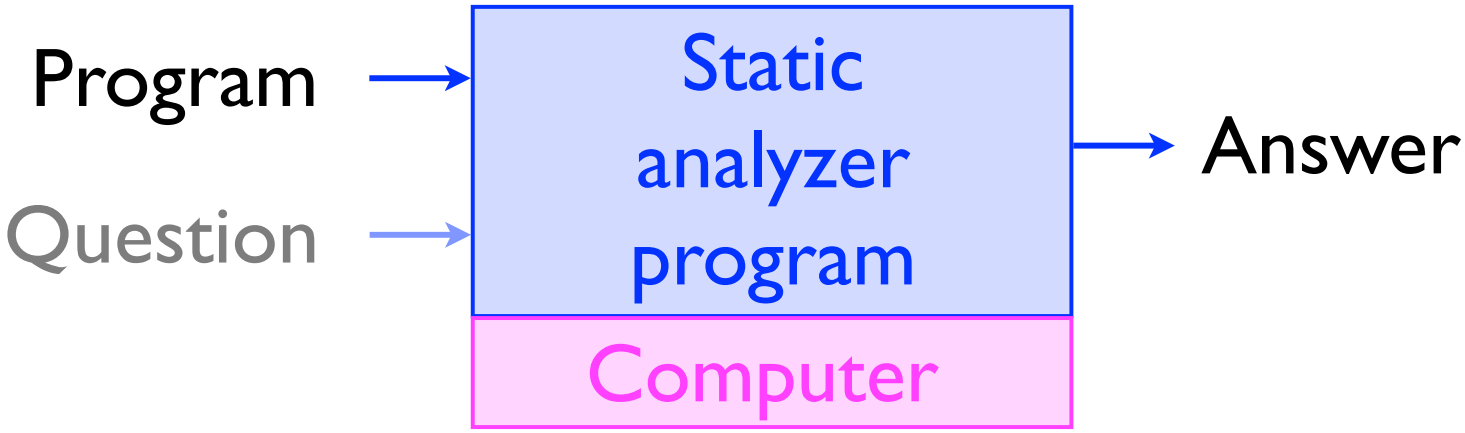
 

# Abstract interpretation

- *Started in the 70's*
- Statically and automatically *inferring properties of the behavior of programs/computer systems* for program analysis (proof, verification, optimization, transformation, etc.)
- Based on the idea that *undecidability* and *complexity* of automated static program analysis can be fought by *approximation*
- *Applications* of abstract interpretation *do scale up!*

# Application to static analysis and verification

- Static analysis consists in automatically answering questions about the runtime executions of programs
- Static means « at compile time », by examining the program text only, without executions on computers
- Automatic means by a computer, without human intervention during the analysis



# Fighting *undecidability* and *complexity* in program verification

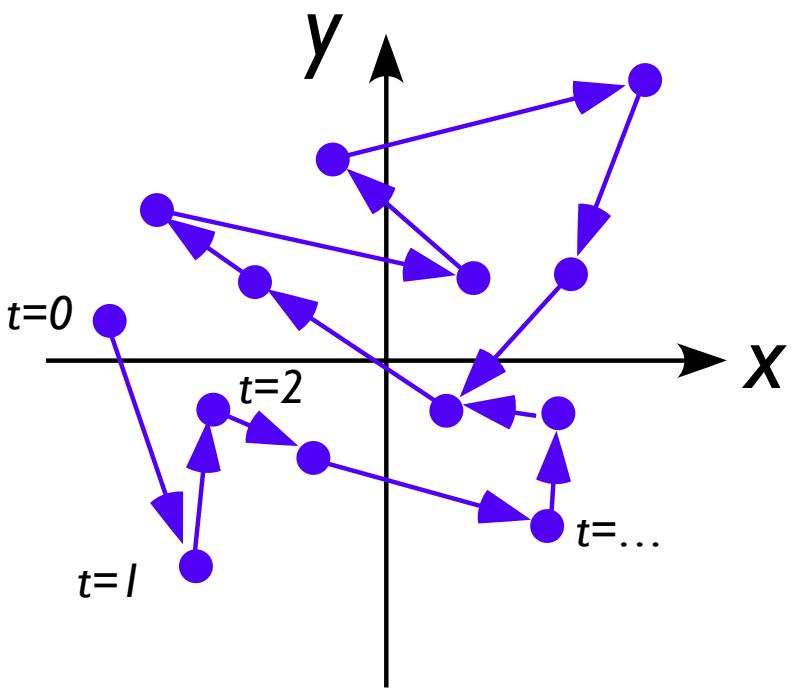
- Any *automatic* program verification method will definitely fail on infinitely many programs (Gödel)
- Solutions:
  - Ask for human help (theorem-prover based *deductive methods*)
  - Consider (small enough) finite systems (*model-checking*)
  - Do complete abstractions or else sound approximations (*abstract interpretation*)

# An informal introduction to abstract interpretation

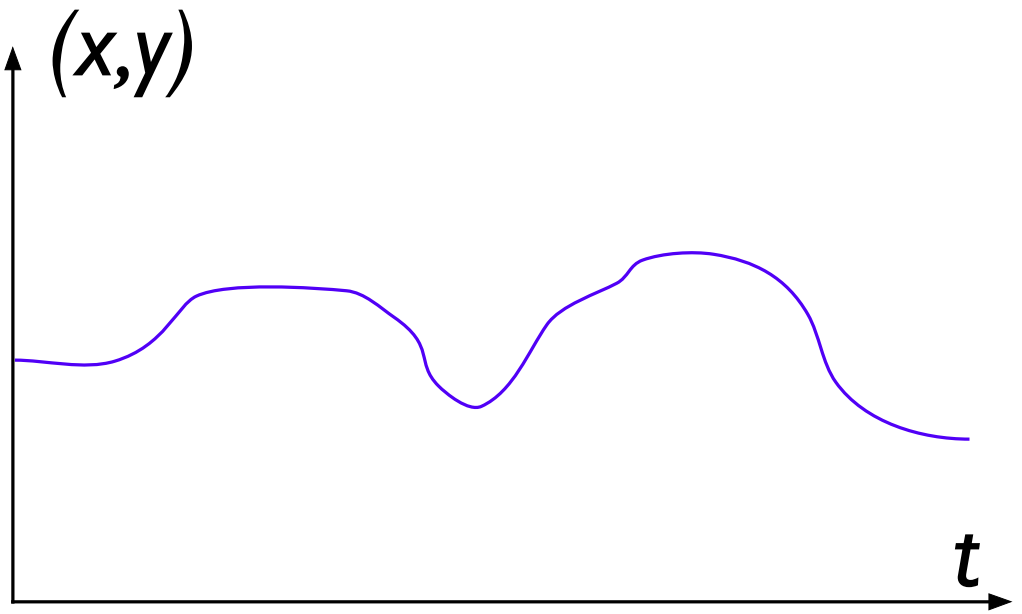
P. Cousot & R. Cousot. A gentle introduction to formal verification of computer systems by abstract interpretation. In *Logics and Languages for Reliability and Security*, J. Esparza, O. Grumberg, & M. Broy (Eds), NATO Science Series III: Computer and Systems Sciences, © IOS Press, 2010, Pages 1—29.

# 1) Define the programming language semantics

*Formalize the concrete* ***execution*** *of programs (e.g. transition system)*



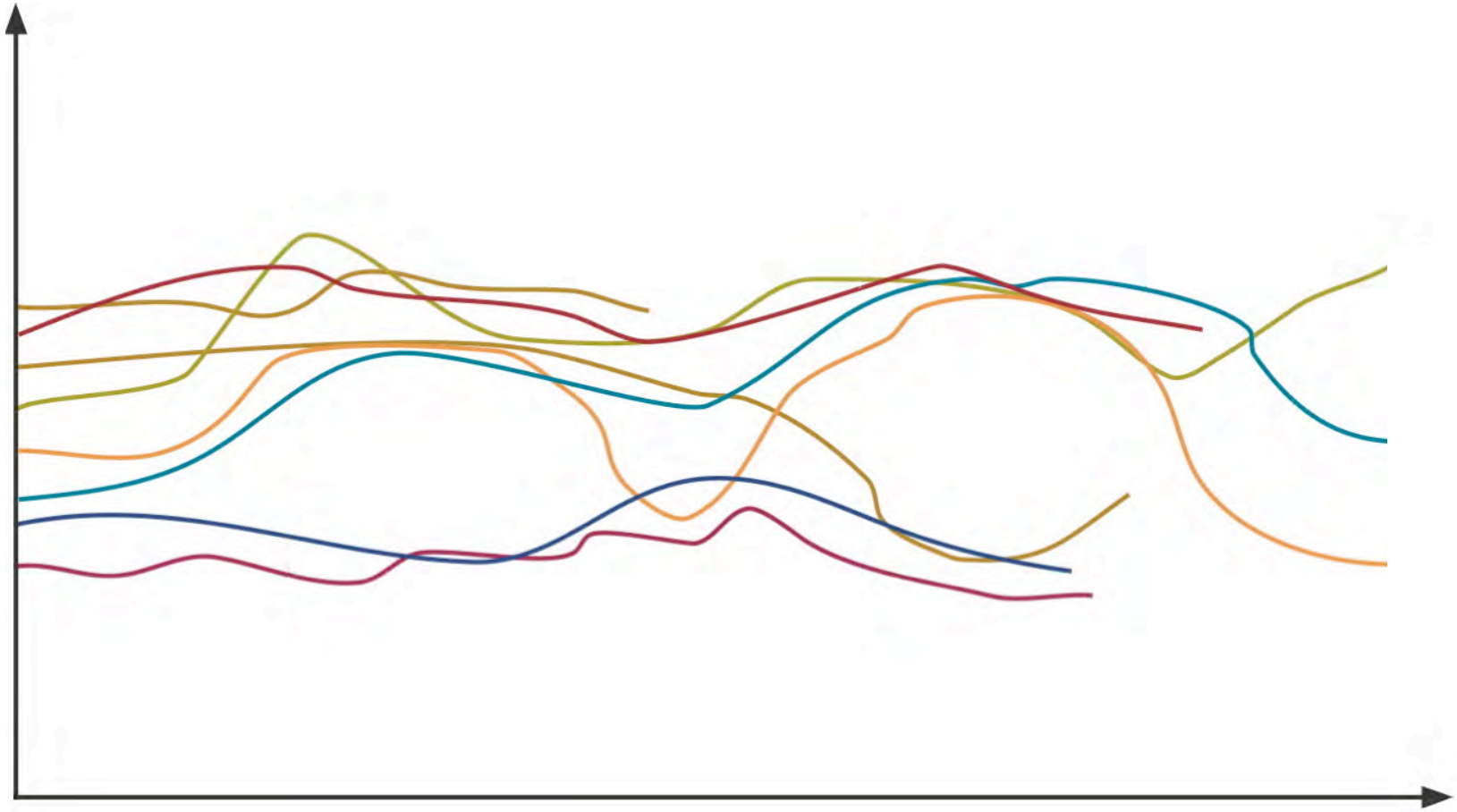

Trajectory
in state space




Space/time trajectory

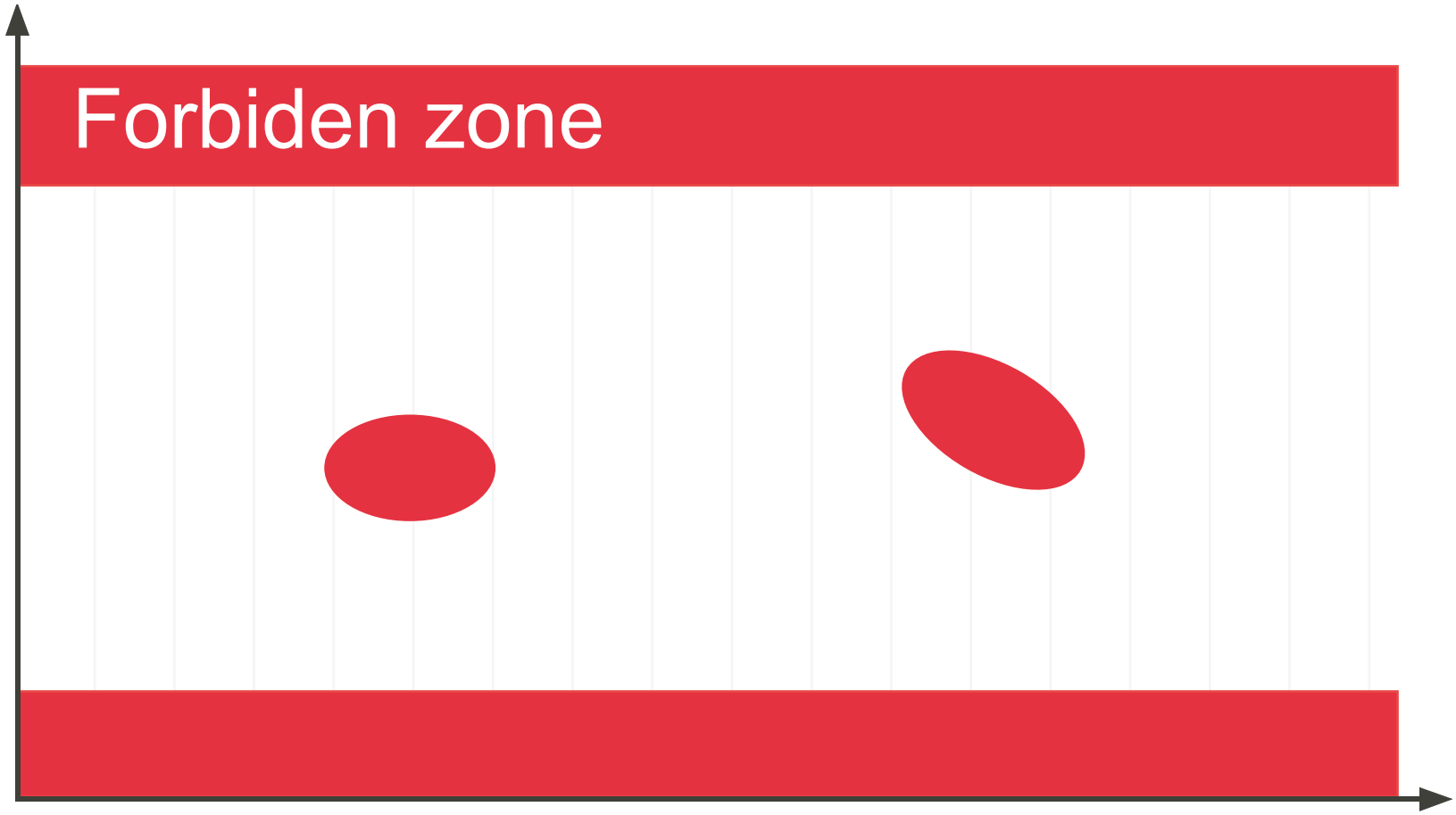
# II) Define the program properties of interest

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



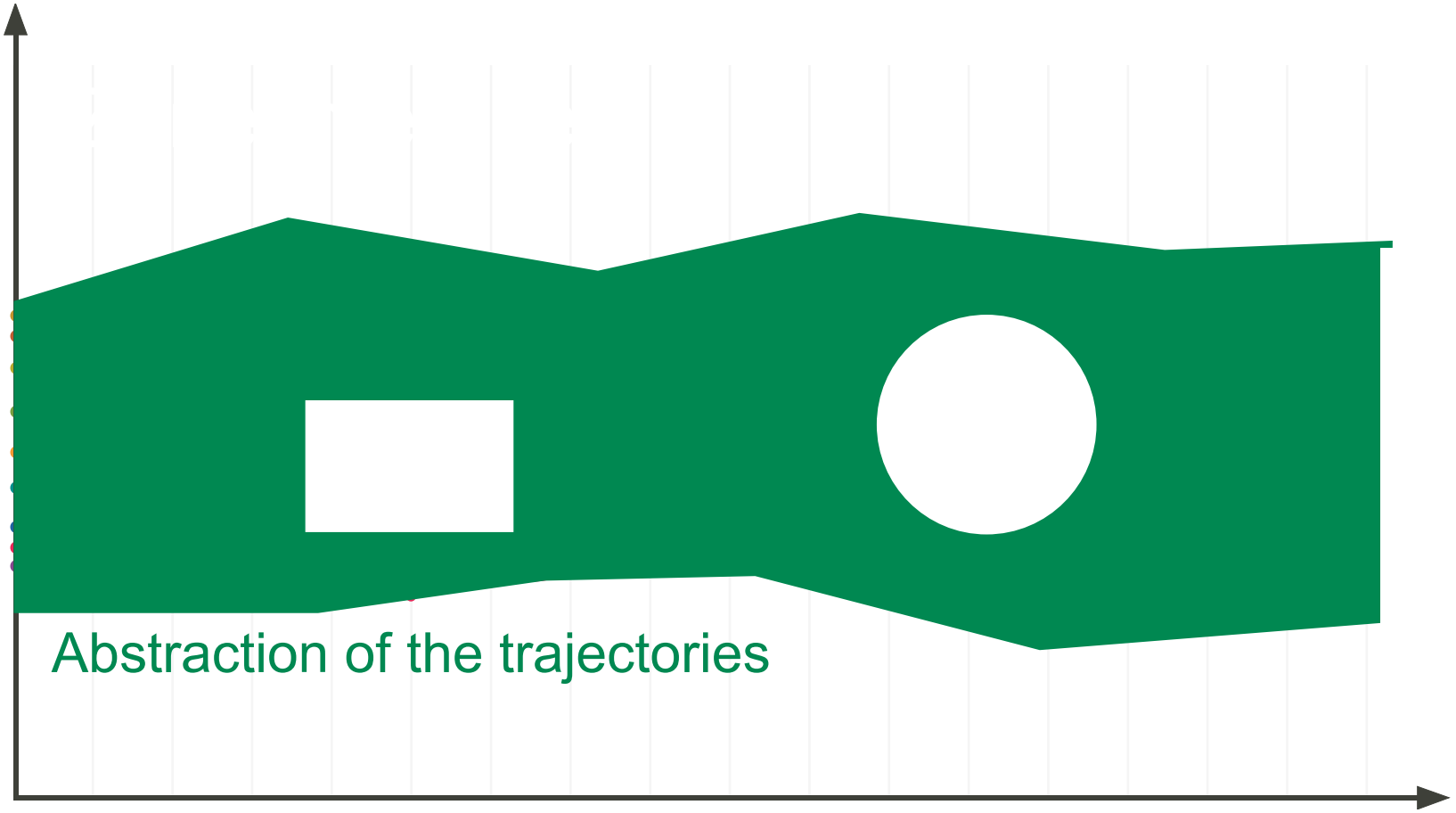
# III) Define which specification must be checked

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



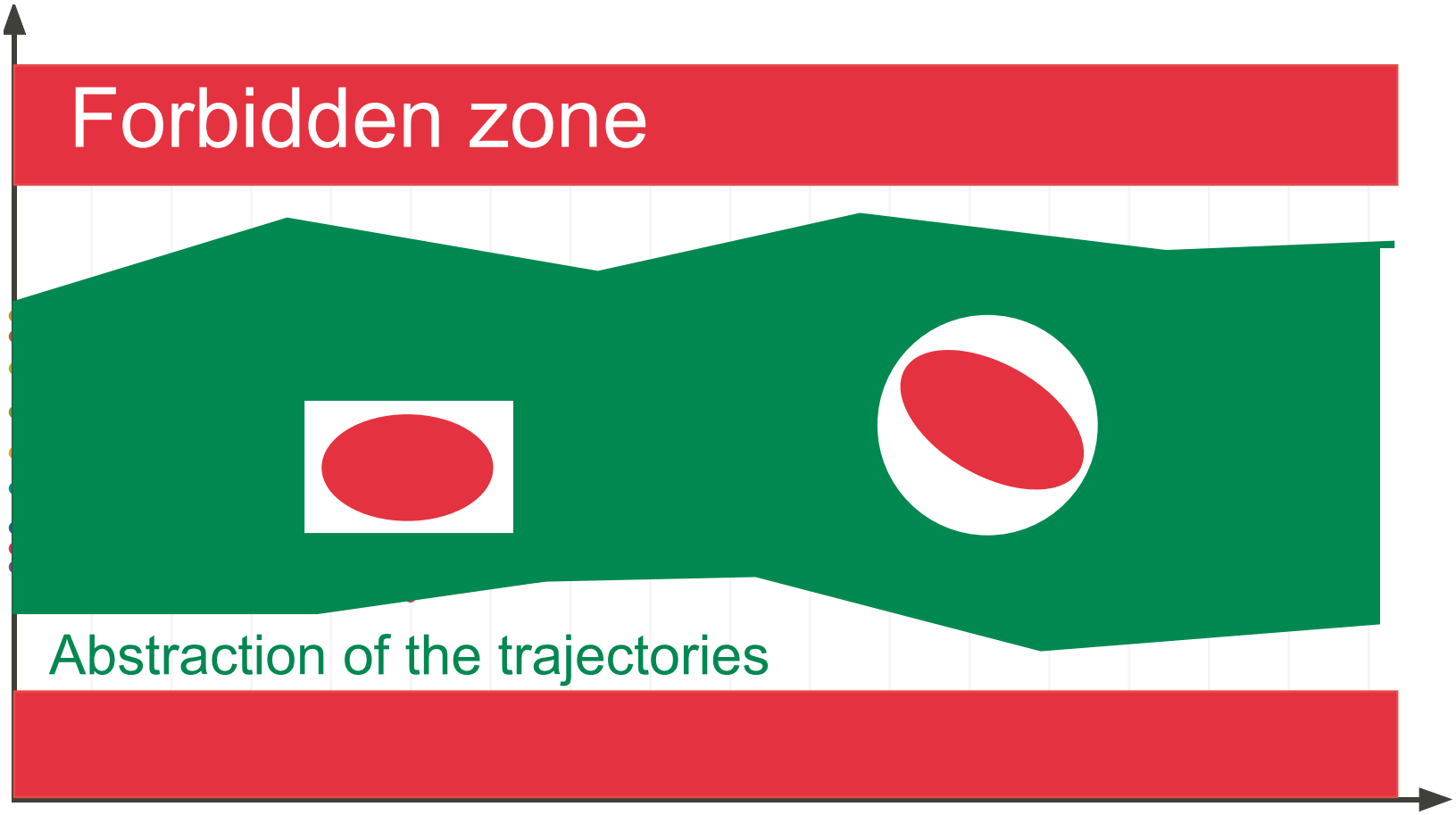
# IV) Choose the appropriate abstraction

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



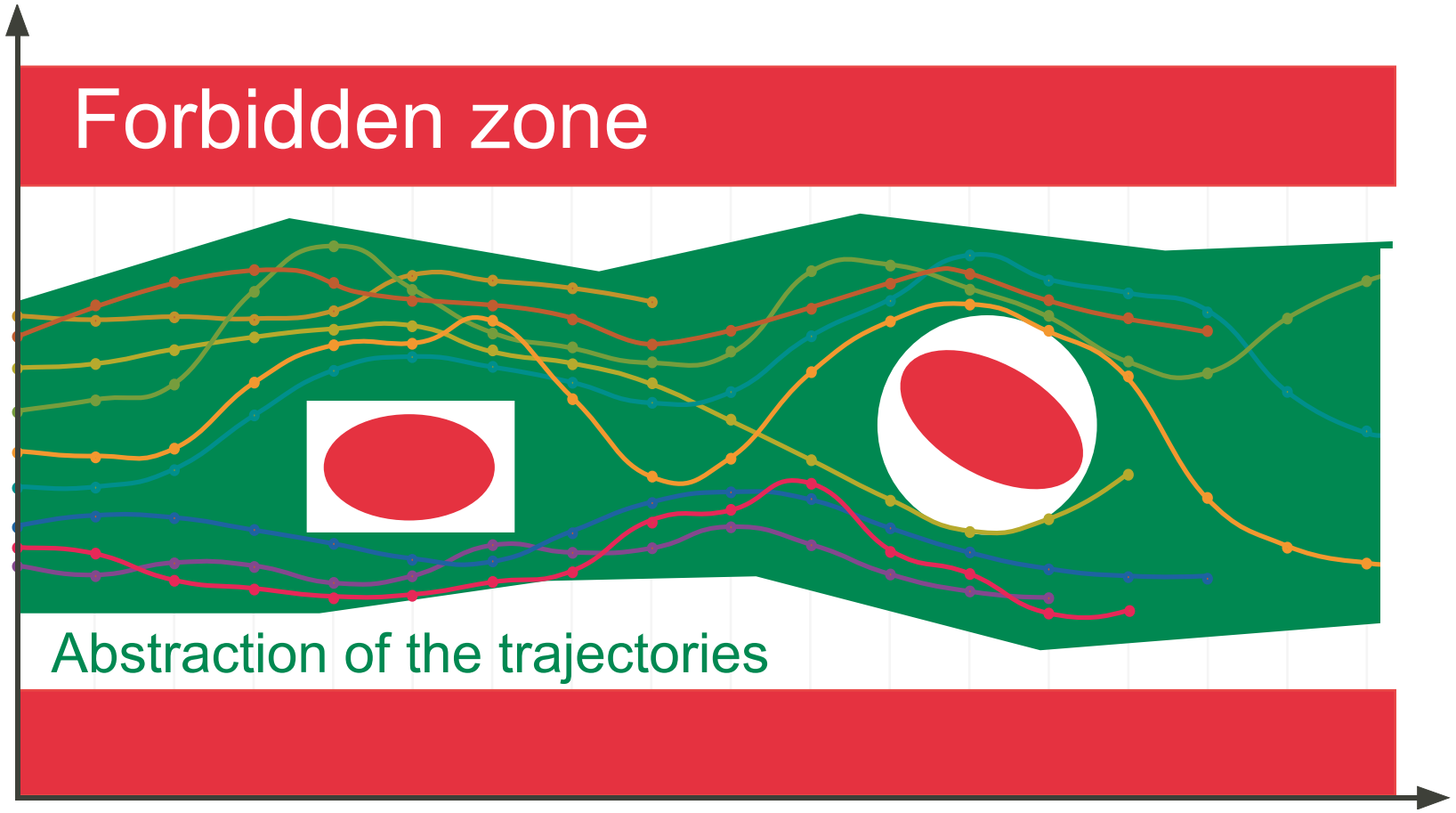
# V) Mechanically verify in the abstract

*The proof is fully* ***automatic***



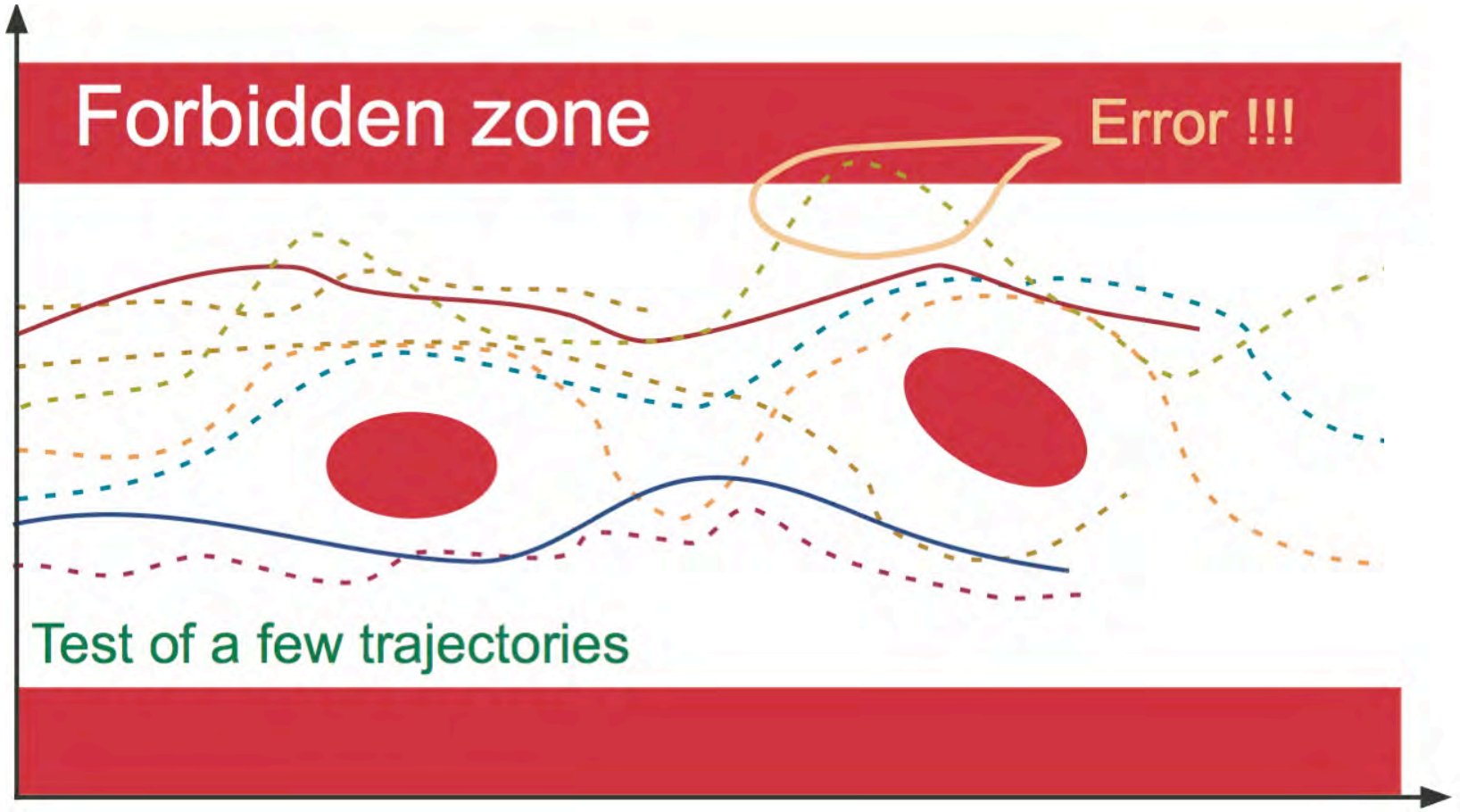
# Soundness of the abstract verification

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



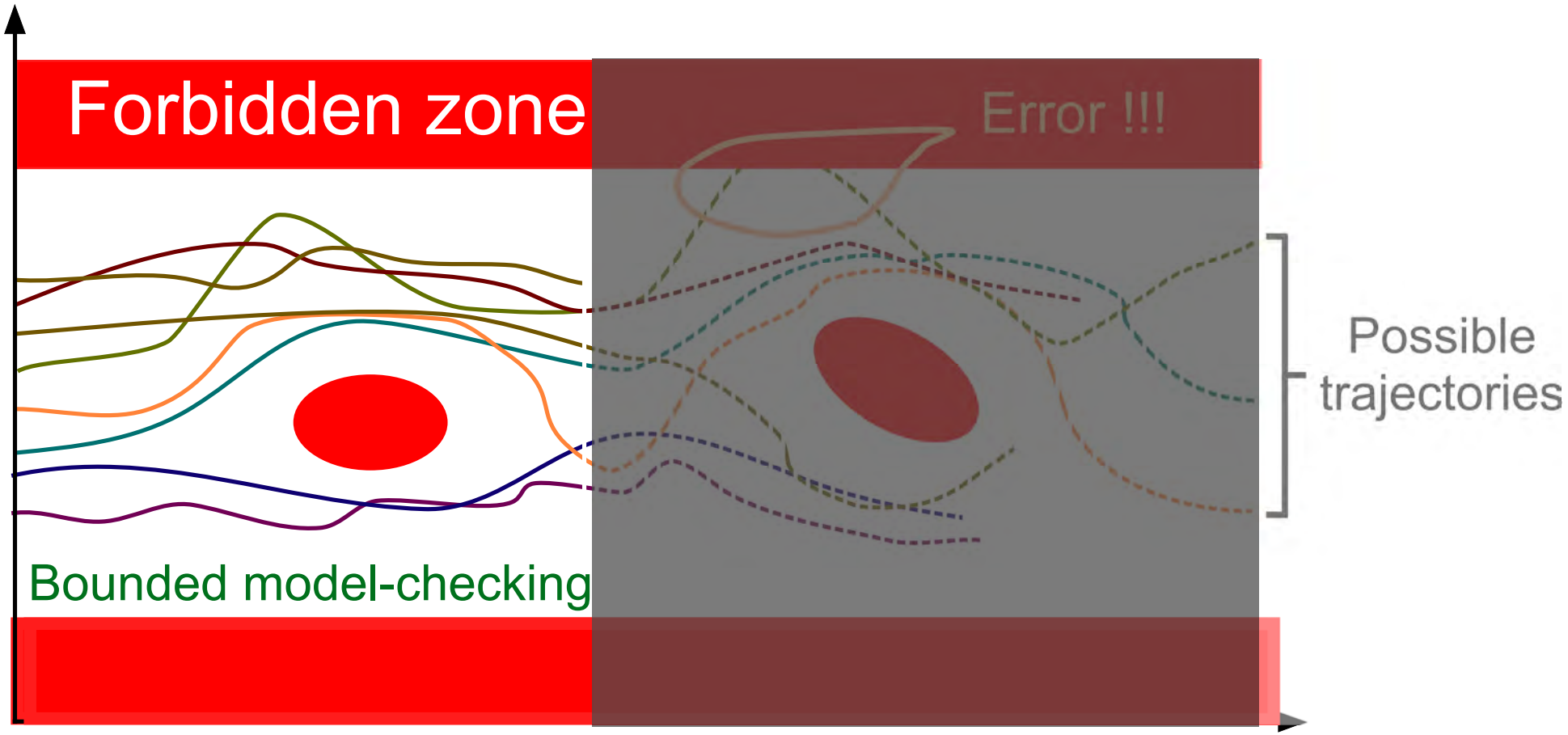
# Unsound validation: testing

*Try a few cases*



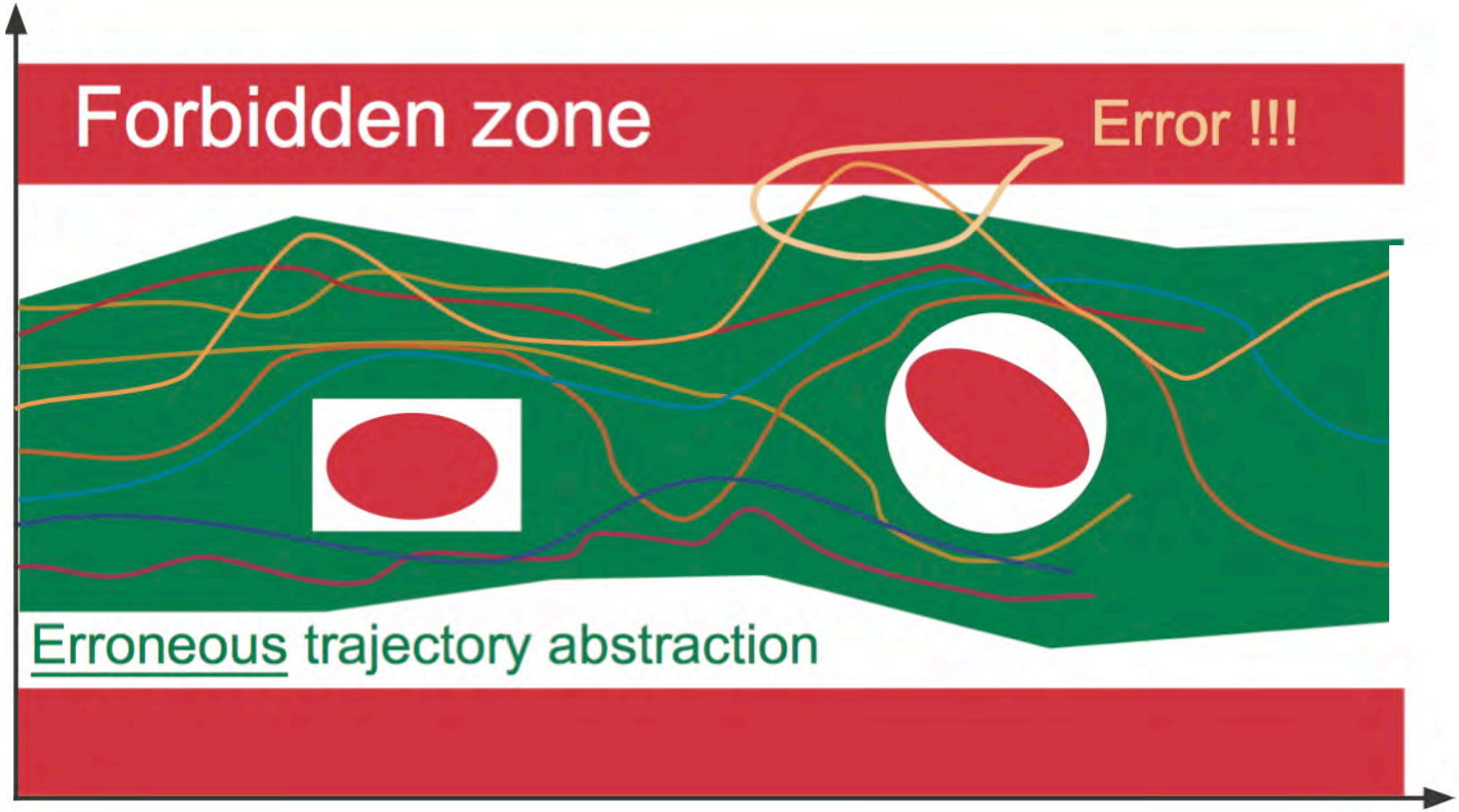
# Unsound validation: bounded model-checking

*Simulate the beginning of all executions*



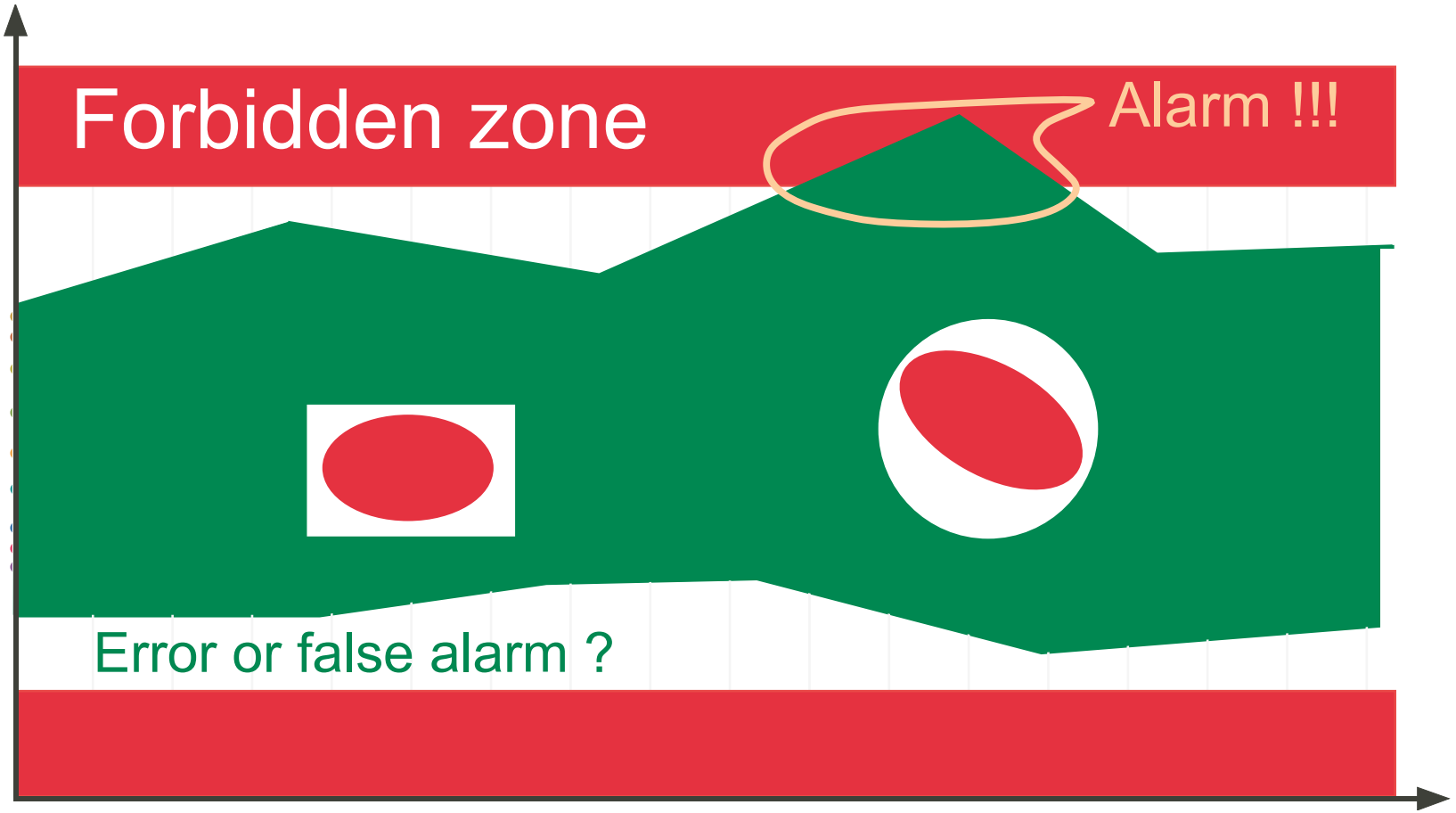
# Unsound validation: static analysis

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



# Incompleteness

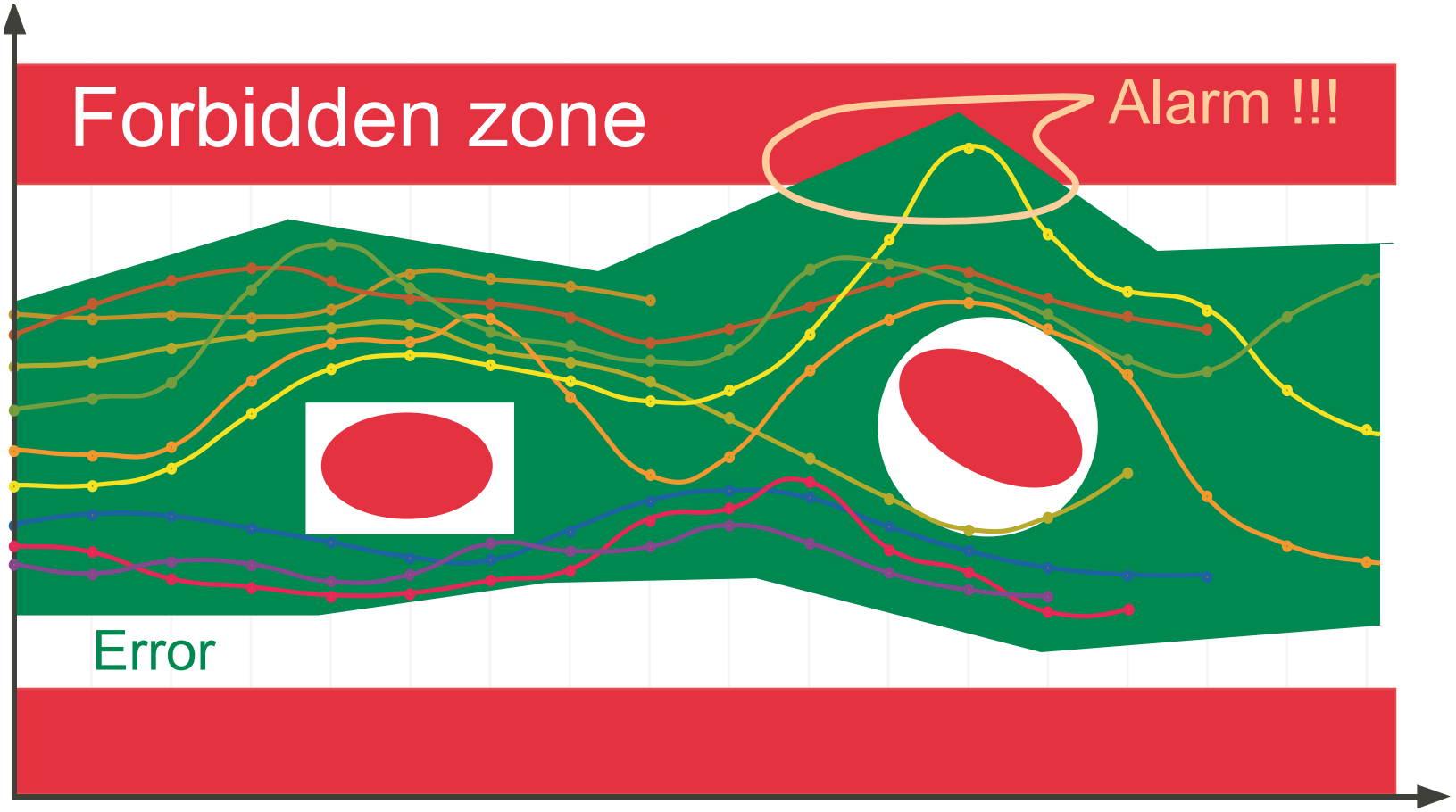
*When abstract proofs may fail while concrete proofs would succeed*




*By soundness an alarm must be raised for this overapproximation!*

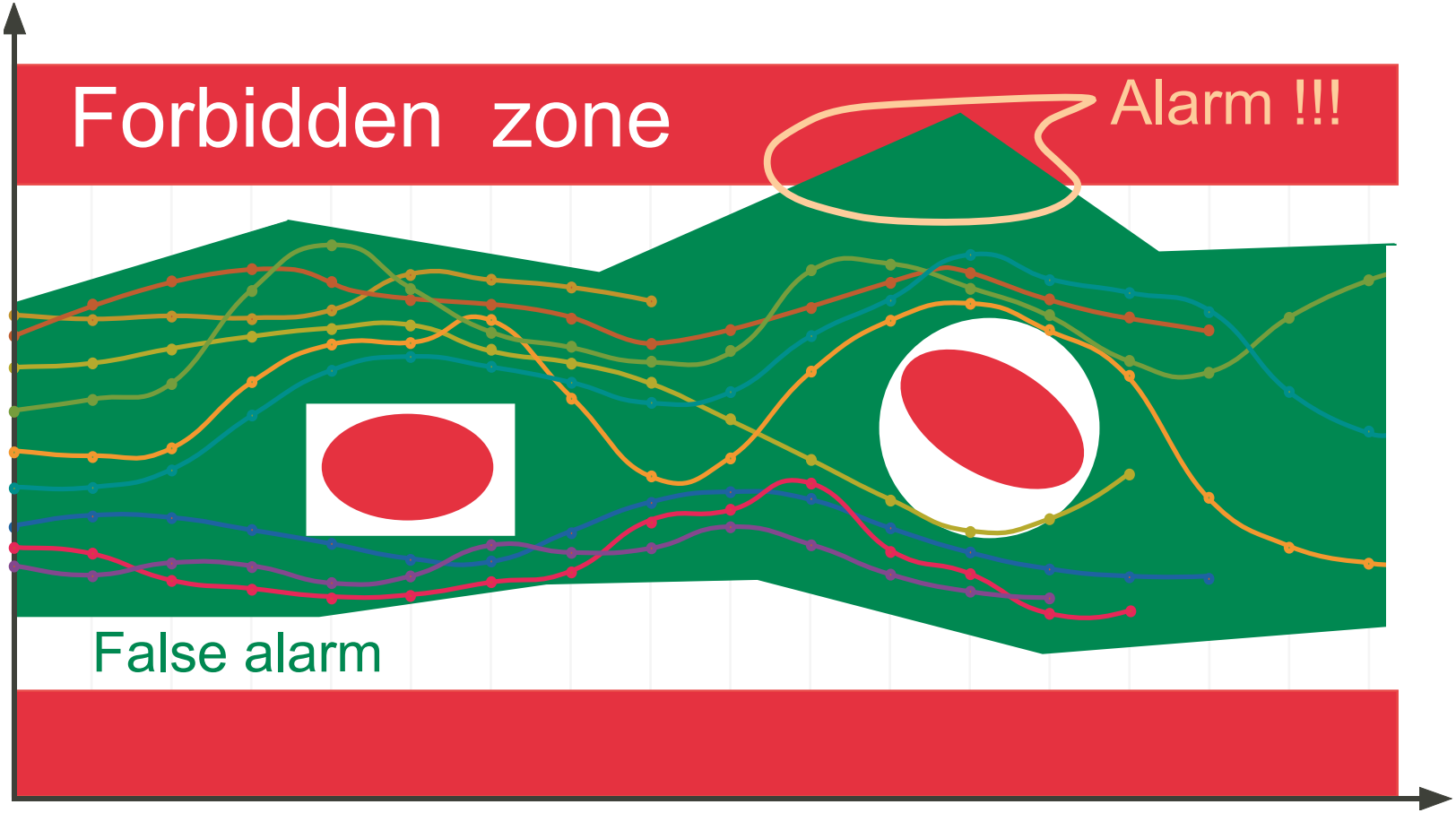
# True error

*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 about false alarms?

- Automatic refinement: inefficient and may not terminate (Gödel)
- Domain-specific abstraction:
  - Adapt the abstraction to the *programming paradigms* typically used in given *domain-specific applications*
  - e.g. *synchronous control/command*: no recursion, no dynamic memory allocation, maximum execution time, etc.

# Just a bit less informally ...

Patrick Cousot & Radhia Cousot. Basic Concepts of Abstract Interpretation. In *Building the Information Society* , René Jacquard (Ed.), Kluwer Academic Publishers, pp. 359—366, 2004. (IFIP WCC 2004 Toulouse, Topical Day on Abstract Interpretation, Tuesday 24 August 2004).

# Semantics and specification

- Small-step operational semantics (of a program)

  $\langle \Sigma,\ \tau,\ \mathsf{Init} \rangle$

- Reachable states

  $\tau^*$ — transitive closure

  $\mathsf{post}[\![r]\!]X \triangleq \{s' \mid \exists s \in X : r(s, s')\}$ — post-image

  $\mathsf{post}[\![\tau^*]\!]\mathsf{Init}$ — reachable states

  $= \mathsf{lfp}^{\subseteq}\, \lambda X \bullet \mathsf{Init} \cup \mathsf{post}[\![\tau]\!]X$ — fixpoint characterization

- Absence of run-time errors

  $\mathsf{Bad}$ — erroneous states

  $\mathsf{post}[\![\tau^*]\!]\mathsf{Init} \subseteq \Sigma \setminus \mathsf{Bad}$ — no erroneous state is reachable from initial states

# Fixpoint abstraction and approximation

**Theorem** *If $\langle L, \leqslant, \bot\rangle$ is a complete lattice or a cpo, $F \in L \to L$ is increasing, $\langle \overline{L}, \sqsubseteq\rangle$ is a poset, $\alpha \in L \to \overline{L}$ is continuous*[6],[7]*, $\overline{F} \in \overline{L} \to \overline{L}$ commutes (resp. semi-commutes) with $F$ that is $\alpha \circ F = \overline{F} \circ \alpha$ (resp. $\alpha \circ F \sqsubseteq \overline{F} \circ \alpha$) then $\alpha(\mathsf{lfp}^{\leqslant}_{\bot} F) = \mathsf{lfp}^{\sqsubseteq}_{\alpha(\bot)} \overline{F}$ (resp. $\alpha(\mathsf{lfp}^{\leqslant}_{\bot} F) \sqsubseteq \mathsf{lfp}^{\sqsubseteq}_{\alpha(\bot)} \overline{F}$).*

Example

$\mathbb{1}_\Sigma$ — identity

$\tau^* = \mathsf{lfp}^{\subseteq} \lambda R \bullet \mathbb{1}_\Sigma \cup R \circ \tau$ — transitive closure

$\lambda R \bullet \mathsf{post}[\![R]\!]\mathsf{Init} \circ \lambda R \bullet \mathbb{1}_\Sigma \cup R \circ \tau] = \lambda R \bullet \mathsf{Init} \cup \mathsf{post}[\![\tau]\!](\mathsf{post}[\![R]\!]\mathsf{Init})$ — commutativity

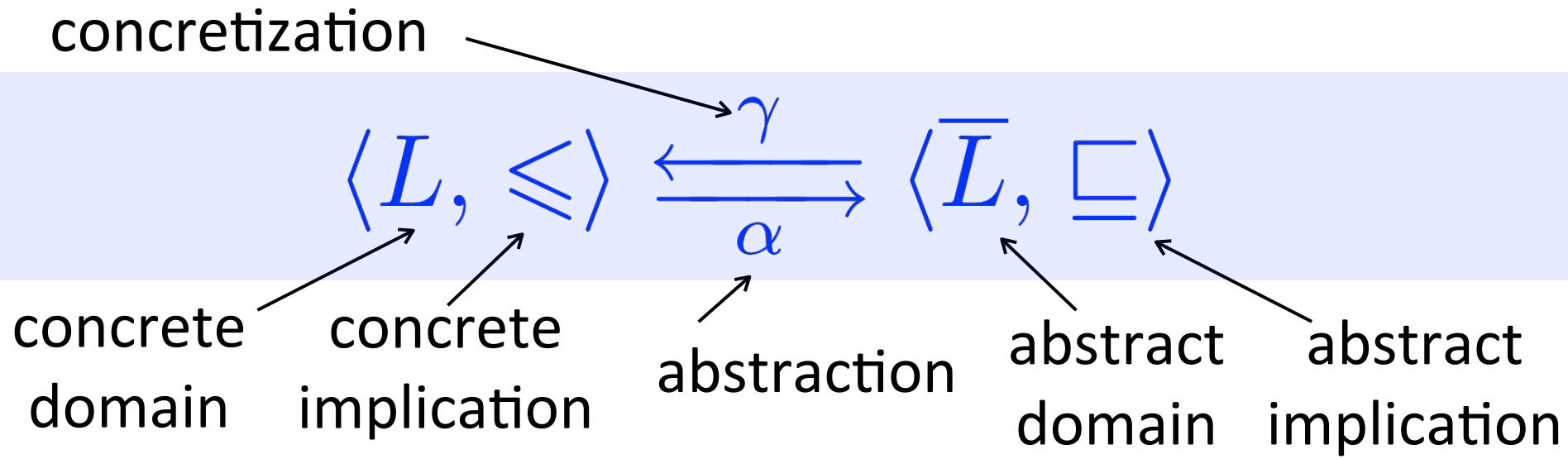
$\Rightarrow\ \mathsf{post}[\![\tau^*]\!]\mathsf{Init} = \mathsf{lfp}^{\subseteq} \lambda X \bullet \mathsf{Init} \cup \mathsf{post}[\![\tau]\!]X$ — reachable states

---

(6) $\alpha$ is *continuous* if and only if it preserves existing lubs of increasing chains.

(7) The continuity hypothesis for $\alpha$ can be restricted to the iterates of the least fixpoint of $F$.

# Abstraction by Galois connections




$\Leftarrow$ best abstraction

$$\forall x \in L, y \in \overline{L} : \alpha(x) \sqsubseteq y \Leftrightarrow x \leqslant \gamma(y)$$

$\Rightarrow$ soundness

Example:

$$\langle \wp(\Sigma \times \Sigma), \subseteq \rangle \underset{\lambda R \bullet \mathsf{post}[\![R]\!]\mathsf{Init}}{\overset{\lambda X \bullet \{\langle s, s' \rangle | s \in \mathsf{Init} \Rightarrow s' \in X\}}{\leftrightarrows}} \langle \wp(\Sigma), \subseteq \rangle$$

# Convergence acceleration

- Fixpoints can be computed iteratively

$$\mathrm{lfp}^{\sqsubseteq}_{\bot} F = \bigsqcup_{n \geqslant 0} F^n(\bot)$$

May not finitely converge

- Accelerate convergence by a widening $\nabla$

$$F^{\uparrow 0} \triangleq \bot, \quad \ldots, \quad F^{\uparrow n+1} \triangleq F^{\uparrow n} \nabla F(F^{\uparrow n}) \quad \text{until} \quad F(F^{\uparrow \ell}) \sqsubseteq F^{\uparrow \ell}$$

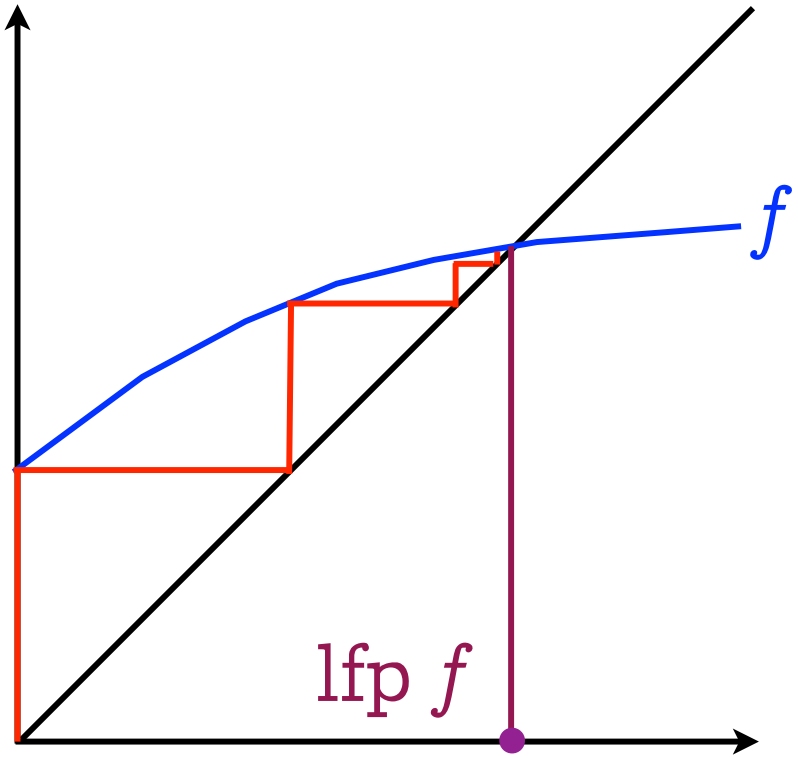
- followed by a narrowing $\Delta$

$$F^{\downarrow 0} \triangleq F^{\uparrow \ell}, \quad \ldots, \quad F^{\downarrow n+1} \triangleq F^{\downarrow n} \Delta F(F^{\downarrow n}) \quad \text{until} \quad F(F^{\downarrow \ell}) = F^{\downarrow \ell}$$

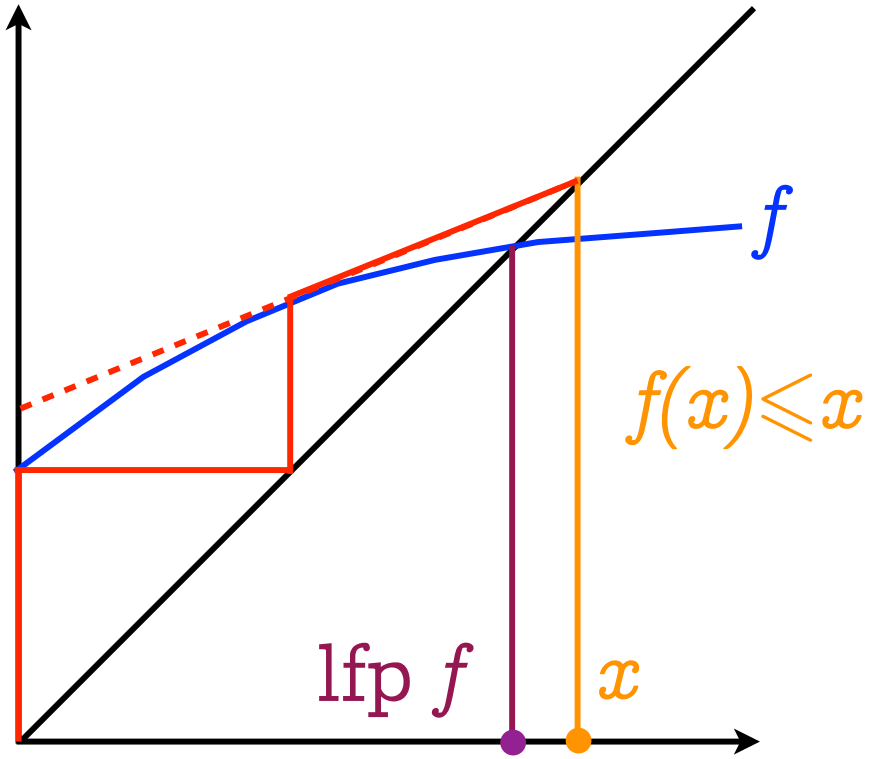
- to get an over-approximation

$$\mathrm{lfp}^{\sqsubseteq}_{\bot} F \sqsubseteq F^{\downarrow \ell} \sqsubseteq F^{\uparrow \ell}$$

# Intuition for Widening




Iteration




Iteration with widening (using the derivative as in Newton-Raphson method)

# Soundness and (In)completeness

- Soundness: We effectively compute $F^{\downarrow\ell}$ such that

$$\alpha(\mathsf{post}[\![\tau^*]\!]\mathsf{Init}) = \alpha(\mathsf{lfp}^{\subseteq}\, \lambda X \bullet \mathsf{Init} \cup \mathsf{post}[\![\tau]\!]X) \sqsubseteq \mathsf{lfp}^{\sqsubseteq}_{\bot}\, F \sqsubseteq F^{\downarrow\ell} \sqsubseteq F^{\uparrow\ell}$$

  and check that $F^{\downarrow\ell} \cap \mathsf{Bad} = \emptyset$ in the abstract,

  proving $\mathsf{post}[\![\tau^*]\!]\mathsf{Init} = \mathsf{lfp}^{\subseteq}\, \lambda X \bullet \mathsf{Init} \cup \mathsf{post}[\![\tau]\!]X \subseteq \Sigma \setminus \mathsf{Bad}$

- [In]completeness: may produce false alarms (e.g. take $\mathsf{Bad} = \neg\mathsf{post}[\![\tau^*]\!]\mathsf{Init}$ so that no abstraction is possible)

- In practice, can be made complete for domain-specific programs (such as synchronous control/ command) with adequate abstractions.

# In practice ...

- Proceed by structural induction in the syntax of programs
- Use chaotic/asynchronous iterations within loops
- Use many abstractions combined in an approximate *reduced product* with partial iterative reduction
  - Example:
    - $x \in [1,13]$
    - even(x)

  reduction:

    - $x \in [2,12]$

# Applications of abstract interpretation to the static analysis of aerospace control systems

Julien Bertrane, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, Xavier Rival. Static Analysis and Verification of Aerospace Software by Abstract Interpretation. AIAA Infotech@Aerospace 2010, AIAA 2010-3385, 20–22 April 2010, Atlanta, GA, `http://pdf.aiaa.org/preview/2010/CDReadyMIAA10_2358/PV2010_3385.pdf`

Patrick Cousot. Integrating Physical Systems in the Static Analysis of Embedded Control Software. The Third Asian Symposium on Programming Languages and Systems (APLAS'05), Tsukuba, Japan, November 3—5, 2005. Lecture Notes in Computer Science, volume 3780, © Springer, Berlin, pp. 135—138.

# ASTRÉE

Patrick Cousot, Radhia Cousot, Jérôme Feret, Antoine Miné, David Monniaux, Laurent Mauborgne, Xavier Rival. The ASTRÉE Analyzer. ESOP 2005: The European Symposium on Programming, Edinburgh, Scotland, April 2—10, 2005. Lecture Notes in Computer Science 3444, © Springer, Berlin, pp. 21—30.

# Target language and applications

- C programming language
  - Without recursion, longjump, dynamic memory allocation, conflicting side effects, backward jumps, system calls (stubs)
  - With all its horrors (union, pointer arithmetics, etc)
  - Reasonably extending the standard (e.g. size & endianess of integers, IEEE 754-1985 floats, etc)
- Synchronous control/command
  - e.g. generated from Scade

# The semantics of C implementations is very hard to define

What is the effect of out-of-bounds array indexing?

```
% cat unpredictable.c
#include <stdio.h>
int main () { int n, T[1];
 n = 2147483647;
 printf("n = %i, T[n] = %i\n", n, T[n]);
}
```

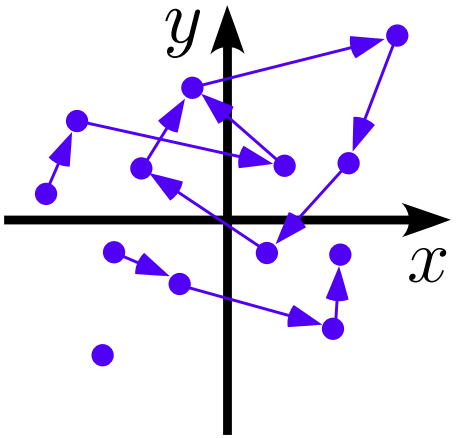
Yields different results on different machines:

| | |
|---|---|
| `n = 2147483647, T[n] = 2147483647` | Macintosh PPC |
| `n = 2147483647, T[n] = -1208492044` | Macintosh Intel |
| `n = 2147483647, T[n] = -135294988` | PC Intel 32 bits |
| `Bus error` | PC Intel 64 bits |

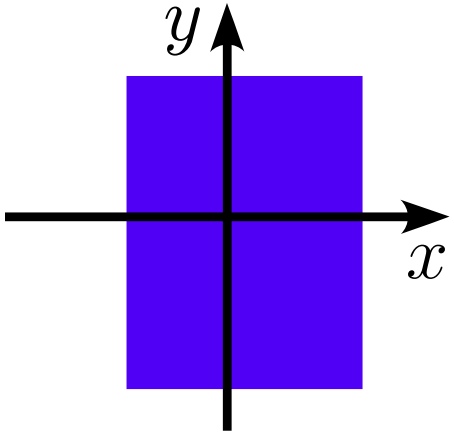
# Implicit specification

- Absence of runtime errors: overflows, division by zero, buffer overflow, null & dangling pointers, alignment errors, …
- Semantics of runtime errors:
  - Terminating execution: stop (e.g. floating-point exceptions when traps are activated)
  - Predictable outcome: go on with worst case (e.g. signed integer overflows result in some integer, some options: e.g. modulo arithmetics)
  - Unpredictable outcome: stop (e.g. memory corruption)

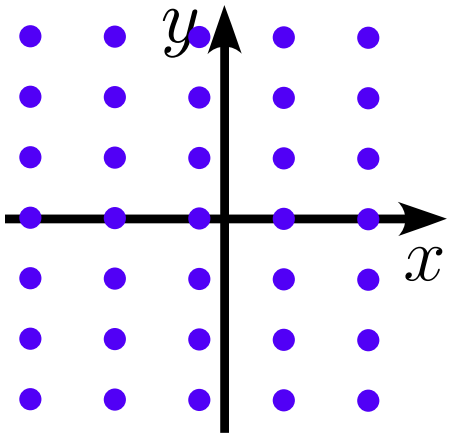
# Abstractions



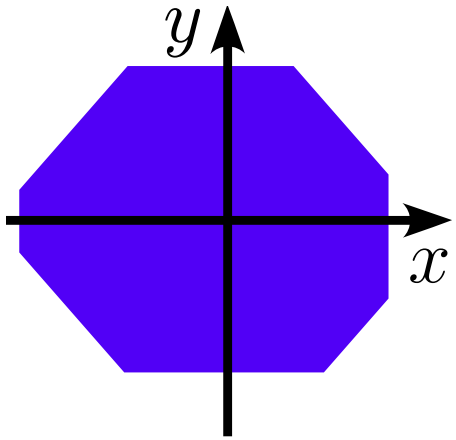

Collecting semantics:
partial traces



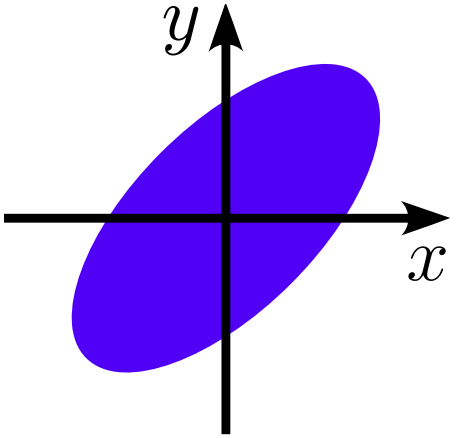

Intervals:
$\mathtt{x} \in [a, b]$



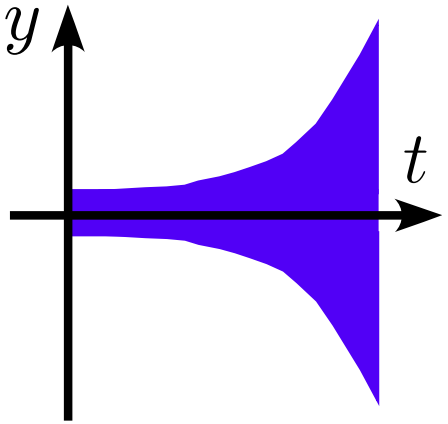

Simple congruences:
$\mathtt{x} \equiv a[b]$




Octagons:
$\pm\mathtt{x} \pm \mathtt{y} \leqslant a$




Ellipses:
$\mathtt{x}^2 + b\mathtt{y}^2 - a\mathtt{xy} \leqslant d$




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

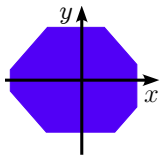
# Example of general purpose abstraction: octagons

- Invariants of the form $\pm \mathtt{x} \pm \mathtt{y} \leq \mathtt{c}$, with $\mathcal{O}(\mathbf{N^2})$ memory and $\mathcal{O}(\mathbf{N^3})$ time cost.
- Example:

```
while (1) {
  R = A-Z;
  L = A;
  if (R>V)
    { ★ L = Z+V; }
  ★
}
```

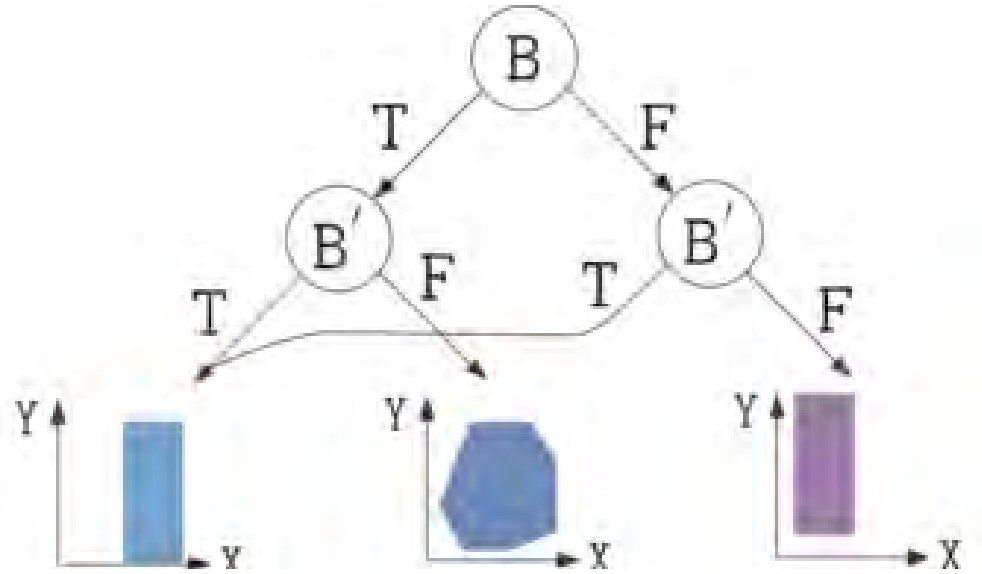
- At ★, the interval domain gives $\mathtt{L} \leq \max(\max \mathtt{A}, (\max \mathtt{Z}) + (\max \mathtt{V}))$.
- In fact, we have $\mathtt{L} \leq \mathtt{A}$.
- To discover this, we must know at ★ that $\mathtt{R} = \mathtt{A}\text{-}\mathtt{Z}$ and $\mathtt{R} > \mathtt{V}$.

- Here, $\mathtt{R} = \mathtt{A}\text{-}\mathtt{Z}$ cannot be discovered, but we get $\mathtt{L}\text{-}\mathtt{Z} \leq \max \mathtt{R}$ which is sufficient.
- We use many octagons on **small packs** of variables instead of a large one using all variables to cut costs.



# Example of general purpose abstraction: decision trees

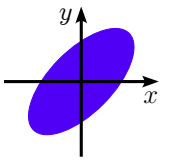
```
/* boolean.c */
typedef enum {F=0,T=1} BOOL;
BOOL B;
void main () {
  unsigned int X, Y;
  while (1) {
    ...
    B = (X == 0);
    ...
    if (!B) {
      Y = 1 / X;
    }
    ...
  }
}
```




The boolean relation abstract domain is parameterized by the height of the decision tree (an analyzer option) and the abstract domain at the leaves

# Example of domain-specific abstraction: ellipses

```
typedef enum {FALSE = 0, TRUE = 1} BOOLEAN;
BOOLEAN INIT; float P, X;
void filter () {
  static float E[2], S[2];
  if (INIT) { S[0] = X; P = X; E[0] = X; }
  else { P = (((((0.5 * X) - (E[0] * 0.7)) + (E[1] * 0.4))
             + (S[0] * 1.5)) - (S[1] * 0.7)); }
  E[1] = E[0]; E[0] = X; S[1] = S[0]; S[0] = P;
  /* S[0], S[1] in [-1327.02698354, 1327.02698354] */
}
void main () { X = 0.2 * X + 5; INIT = TRUE;
  while (1) {
    X = 0.9 * X + 35; /* simulated filter input */
    filter (); INIT = FALSE; }
}
```

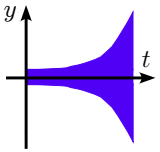


# Example of domain-specific abstraction (I): exponentials

```
% cat count.c
typedef enum {FALSE = 0, TRUE = 1} BOOLEAN;
volatile BOOLEAN I; int R; BOOLEAN T;
void main() {
  R = 0;
  while (TRUE) {
    __ASTREE_log_vars((R));
    if (I) { R = R + 1; }
    else { R = 0; }
    T = (R >= 100);
    __ASTREE_wait_for_clock(());
  }}
```

← potential overflow!

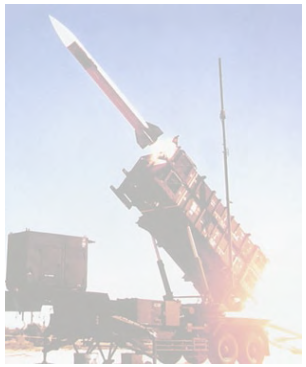
```
% cat count.config
__ASTREE_volatile_input((I [0,1]));
__ASTREE_max_clock((3600000));
% astree -exec-fn main -config-sem count.config count.c|grep '|R|'
|R| <= 0. + clock *1. <= 3600001.
```



# Example of domain-specific abstraction (II): exponentials

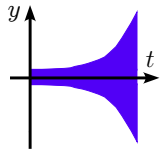
```
% cat retro.c
typedef enum {FALSE=0, TRUE=1} BOOL;
BOOL FIRST;
volatile BOOL SWITCH;
volatile float E;
float P, X, A, B;

void dev( )
{ X=E;
  if (FIRST) { P = X; }
  else
    { P =  (P - ((((2.0 * P) - A) - B)
             * 4.491048e-03)); };
  B = A;
  if (SWITCH) {A = P;}
  else {A = X;}
}
```

Cumulated rounding error



```
void main()
{ FIRST = TRUE;
  while (TRUE) {
    dev( );
    FIRST = FALSE;
    __ASTREE_wait_for_clock(());
  }}
% cat retro.config
__ASTREE_volatile_input((E [-15.0, 15.0]));
__ASTREE_volatile_input((SWITCH [0,1]));
__ASTREE_max_clock((3600000));
```

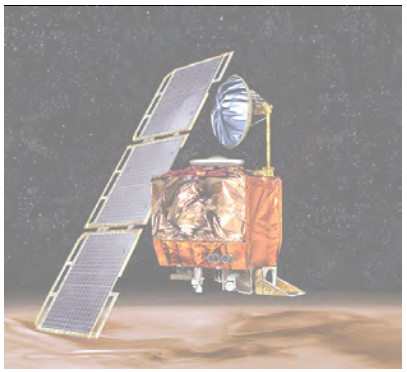
```
|P| <= (15.  + 5.87747175411e-39
/ 1.19209290217e-07) * (1 +
1.19209290217e-07)^clock - 5.87747175411e-39
/ 1.19209290217e-07 <= 23.0393526881
```



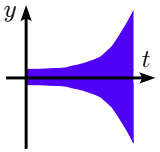
# Example of domain-specific abstraction (III): scaling

```
% cat -n scale.c
 1 int main () {
 2  float x; x = 0.70000001;
 3  while (1) {
 4   x = x / 3.0;
 5   x = x * 3.0;
 6   __ASTREE_log_vars((x));
 7   __ASTREE_wait_for_clock(());
 8  }
 9 }
```

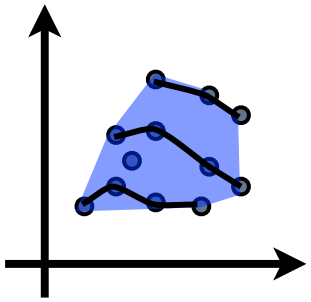
```
% gcc scale.c
% ./a.out
x = 0.699999988079071
```



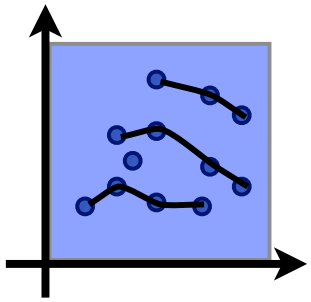
```
% cat scale.config
 __ASTREE_max_clock((1000000000));
% astree -exec-fn main -config-sem scale.config -unroll 0 scale.c\
 |& grep "x in" | tail -1
direct = <float-interval: x in [0.69999986887, 0.700000047684] >
%
```



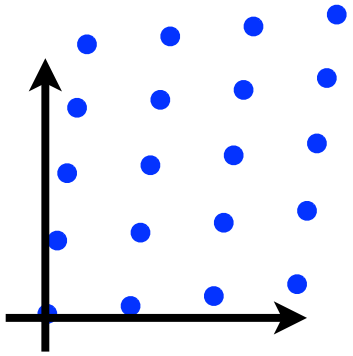
# Examples of abstractions not used by Astrée



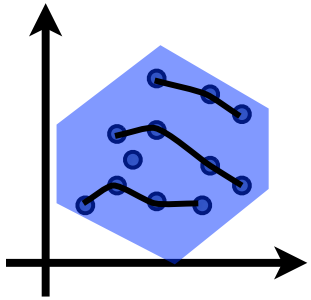
Polyhedra (too expensive)



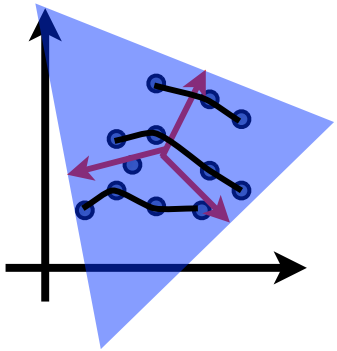
Signs (too imprecise)



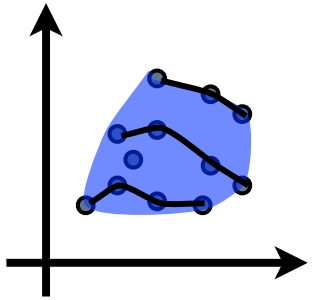
Linear congruences (too expensive)



Zonotopes (inclusion?)



Support functions (widening?)



Convex sets (algorithmics?)

# An erroneous common belief on static analyzers

"The properties that can be proved by static analyzers are often simple" [2]

Like in mathematics:

- May be simple to state (no overflow)
- But harder to discover (`S[0], S[1] in [-1327.02698354, 1327.02698354]`)
- And difficult to prove (since it requires finding a non trivial non-linear invariant for second order filters with complex roots [Fer04], which can hardly be found by exhaustive enumeration)

Reference

[2] Vijay D'Silva, Daniel Kroening, and Georg Weissenbacher. A Survey of Automated Techniques for Formal Software Verification. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 27, No. 7, July 2008.

# Industrial applications

# Examples of applications

- Verification of the absence of runtime-errors in
  - Fly-by-wire flight control systems

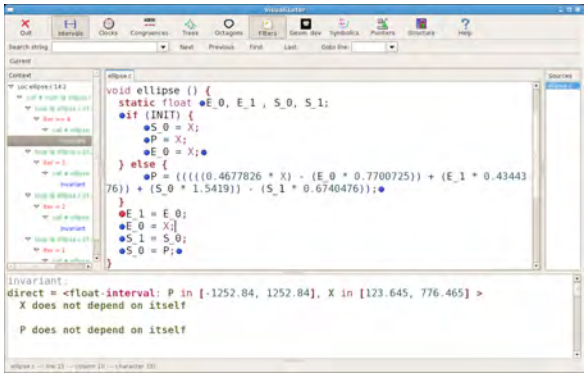

  - ATV docking system

  - Flight warning system (on-going work)

# Industrialization

- 8 years of research (CNRS/ENS/INRIA):

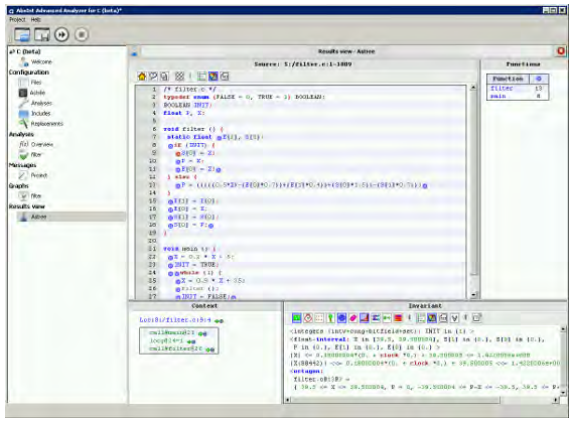
  www.astree.ens.fr

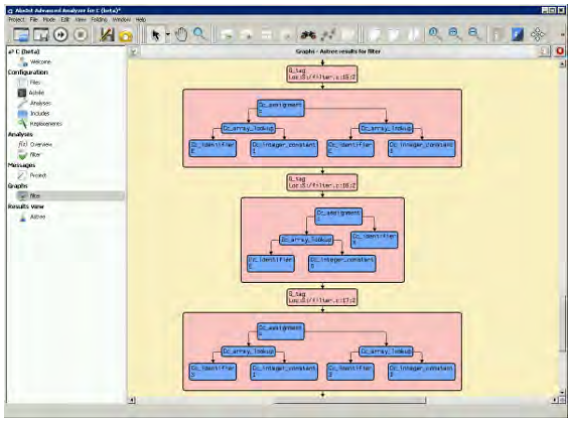



- Industrialization by AbsInt (since Jan. 2010):

  www.absint.com/astree/

AbsInt
Angewandte Informatik





# On-going work

Julien Bertrane, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, Xavier Rival. Static Analysis and Verification of Aerospace Software by Abstract Interpretation. AIAA Infotech@Aerospace 2010, AIAA 2010-3385, 20–22 April 2010, Atlanta, GA, `http://pdf.aiaa.org/preview/2010/CDReadyMIAA10_2358/PV2010_3385.pdf`

# Verification of target programs

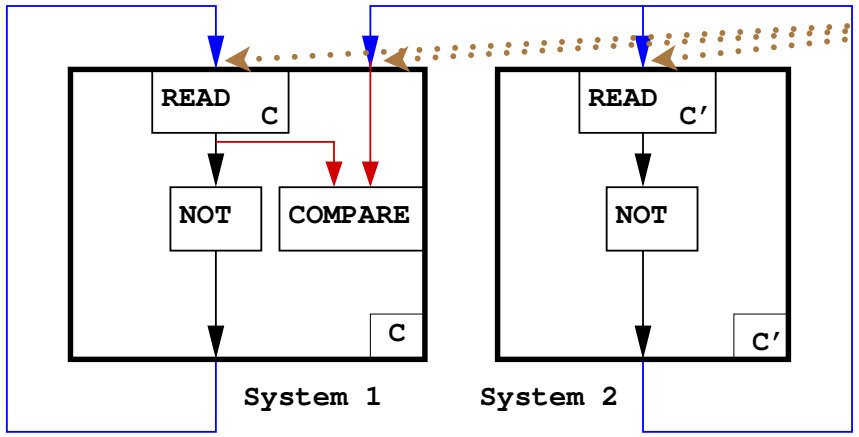
# Verification of compiled programs

- The valid source may be proved correct while the certified compiler is incorrect so the target program may go wrong
- Possible approaches:
  - Verification at the target level
  - Source to target proof translation and proof check on the target
  - ✳ Translation validation (local verification of equivalence of run-time error free source and target)
  - Formally certified compilers

# Verification of imperfectly clocked synchronous systems

# Imperfect synchrony

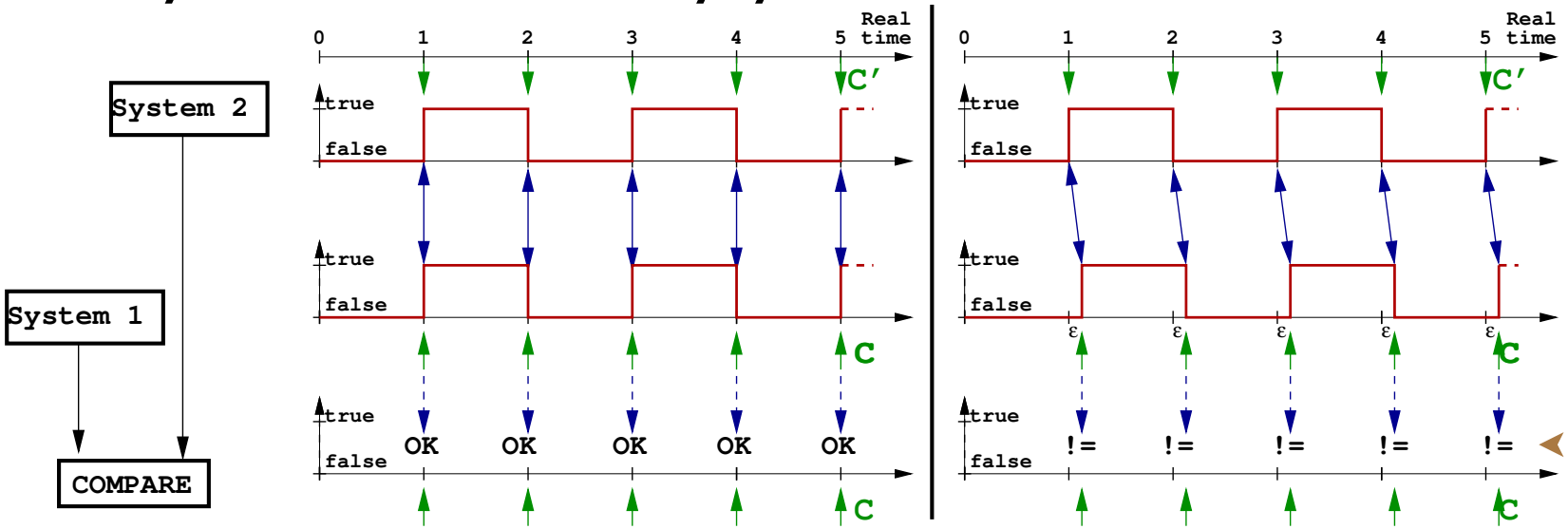
- Example of (buggy) communicating synchronous systems:




blackboard inputs

- negate previous input (on clocks C and C')
- compare inputs

- Synchronized and dysynchronized executions:



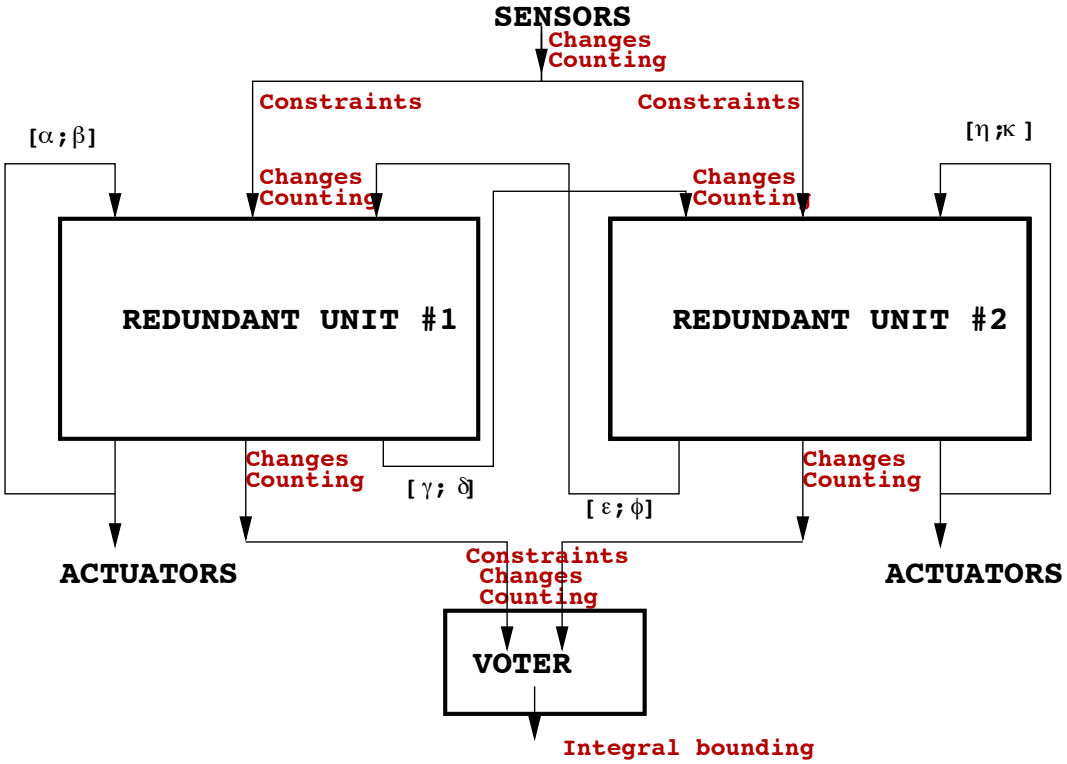

flawed alarms

# Semantics and abstractions

- Continuous semantics (value *s(t)* of signals *s* at any time *t*)
- Clock ticks and serial communications do happen in known time intervals $[l, h], l \leq h$
- Examples of abstractions:
  - $\forall t \in [a; b] : s(t) = x.$
  - $\exists t \in [a; b] : s(t) = x.$
  - change counting $(\leqslant k, a \blacktriangleright \blacktriangleleft b)$ and $(\geqslant k, a \blacktriangleright \blacktriangleleft b)$

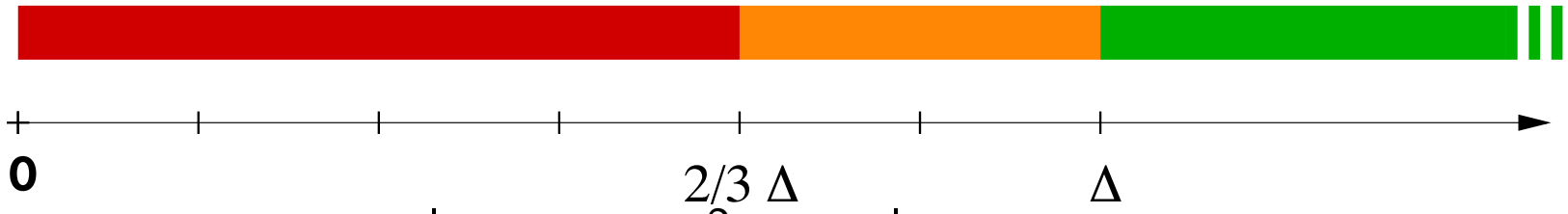
    (signal changes less (more) than *k* times in time interval [*a*, *b*])

# Example of static analysis




For how long should the input be stabilized before deciding on disagreement?

**Specification** : no alarm raised with a normal input




| input stability $< \Delta$ : | Between $\frac{2}{3} \times \Delta$ | input stability $> \Delta$ : the analyzer |
|---|---|---|
| counter-example | and $\Delta$ : ? | proves the specification |

# THÉSÉE: Verification of embedded real-time parallel C programs

# Parallel programs

- Bounded number of processes with shared memory, events, semaphores, message queues, blackboards,…
- Processes created at initialization only
- Real time operating system (ARINC 653) with fixed priorities (highest priority runs first)
- Scheduled on a single processor

# Verified properties

- Absence of runtime errors
- Absence of unprotected data races

# Semantics

- No memory consistency model for C
- Optimizing compilers consider sequential processes out of their execution context

init: flag1 = flag2 = 0

| process 1: | process 2: |
|---|---|
| `flag1 = 1;` | `flag2 = 1;` |
| `if (!flag2)` | `if (!flag1)` |
| `{` | `{` |
| `  /* critical section */` | `  /* critical section */` |

write to `flag1/2` and read of `flag2/1` are independent so can be reordered → error!

- We assume:
  - sequential consistency in absence of data race
  - for data races, values are limited by possible interleavings between synchronization points

# Abstractions

- Based on Astrée for the sequential processes
- Takes scheduling into account
- OS entry points (semaphores, logbooks, sampling and queuing ports, buffers, blackboards, …) are all stubbed (using Astrée stubbing directives)
- Interference between processes: flow-insensitive abstraction of the writes to shared memory and inter-process communications

# Example of application: FWS

- Degraded mode (5 processes, 100 000 LOCS)
  - 1h40 on 64-bit 2.66 GHz Intel server
  - 98 alarms
- Full mode (15 processes, 1 600 000 LOCS)
  - 50 h
  - 12 000 alarms in May, 7000 in December 2010
  - !!! more work to be done !!! (e.g. analysis of complex data structures, interference, logs, etc)

# Conclusion

# Cost-effective verification

- The rumor has it that:
  - Manuel validation (testing) is costly, unsafe, not a verification!
  - Formal proofs by theorem provers are extremely laborious hence costly
  - Model-checkers do not scale up
- Why not try abstract interpretation?
  - Domain-specific static analysis scales and can deliver no false alarm

# The End

Cited references are available online at URL *http://www.di.ens.fr/~cousot/COUSOTpapers.shtml*