Software Verification by Abstract Interpretation: Current Trends and Perspectives

Patrick COUSOT

École Normale Supérieure
45 rue d'Ulm
75230 Paris cedex 05, France
Patrick.Cousot@ens.fr
www.di.ens.fr/~cousot

Jerome C. Hunsaker Visiting Professor
Massachusetts Institute of Technology
Department of Aeronautics and Astronautics
cousot@mit.edu
www.mit.edu/~cousot

Pratt & Whitney, East Hartford
Friday May 6th, 2005

All Computer Scientists Have Experienced Bugs

It is preferable to verify that safety-critical programs do not go wrong before running them.

Static Analysis by Abstract Interpretation

Static analysis: analyze the program at compile-time to verify a program runtime property (e.g. the absence of some categories of bugs)

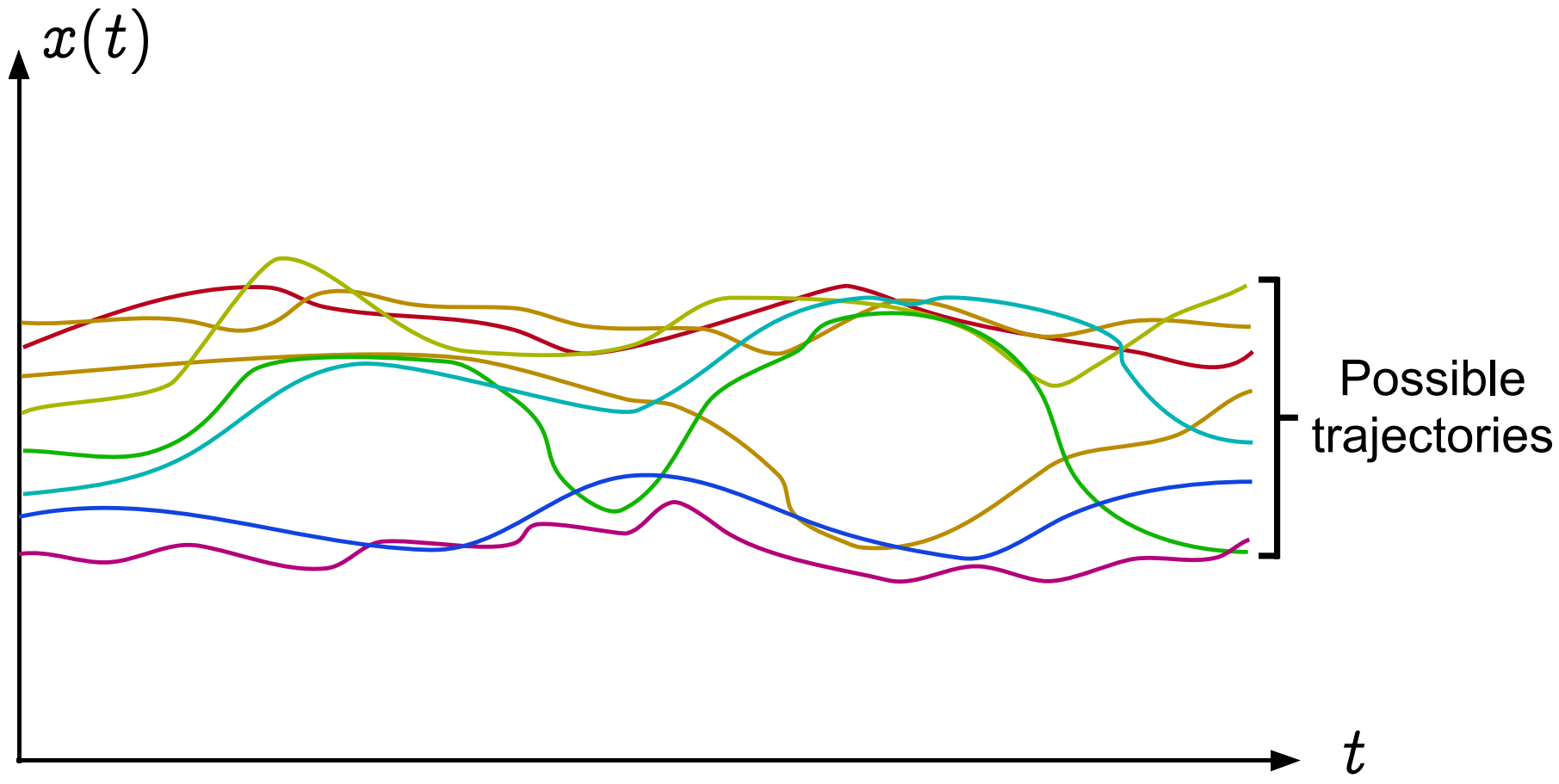
Undecidability $\longrightarrow$

Abstract interpretation: effectively compute an abstraction/sound approximation of the program semantics,

- which is precise enough to imply the desired property, and
- coarse enough to be efficiently computable.

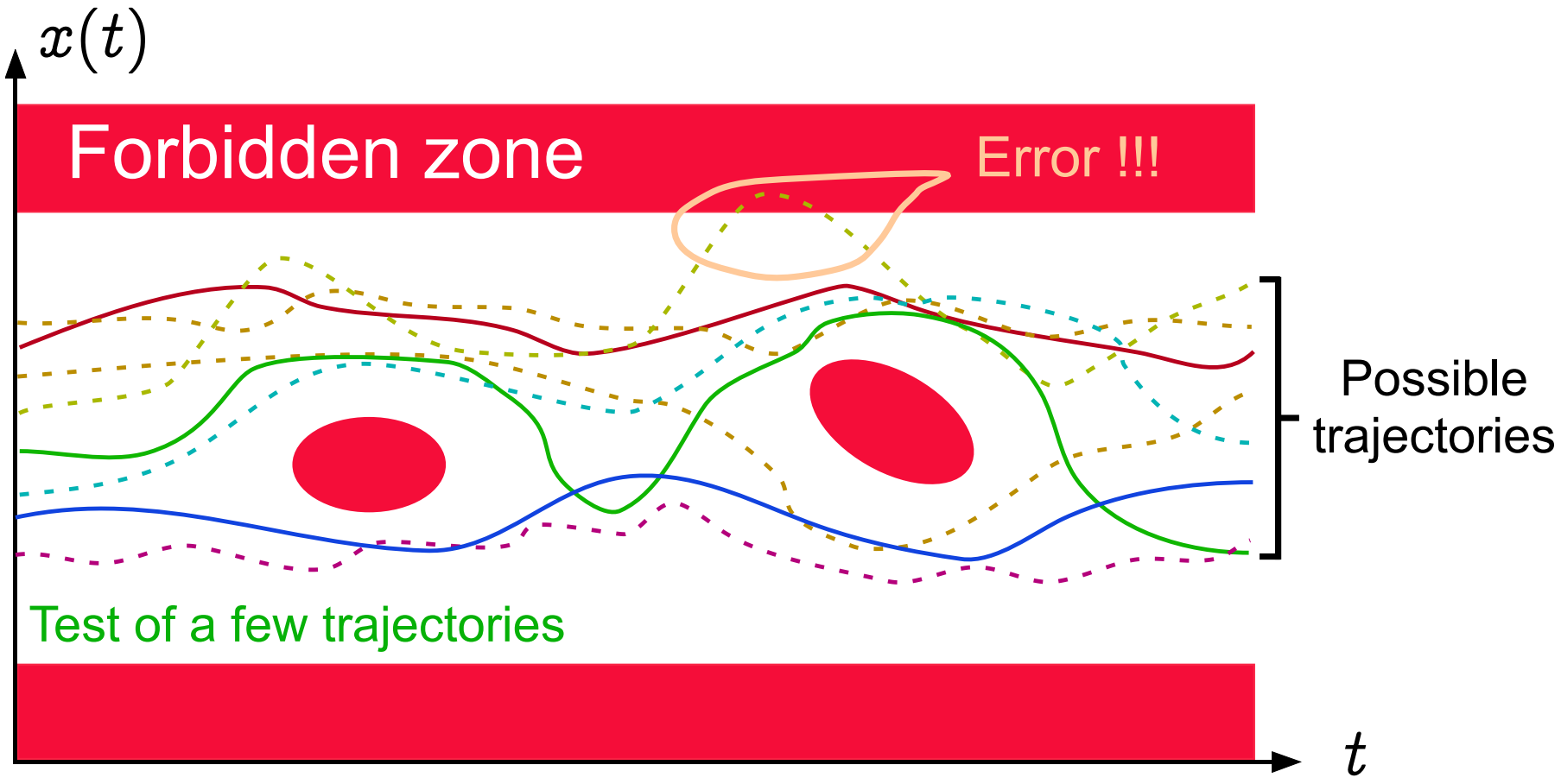
Abstract Interpretation, Informally

Operational Semantics

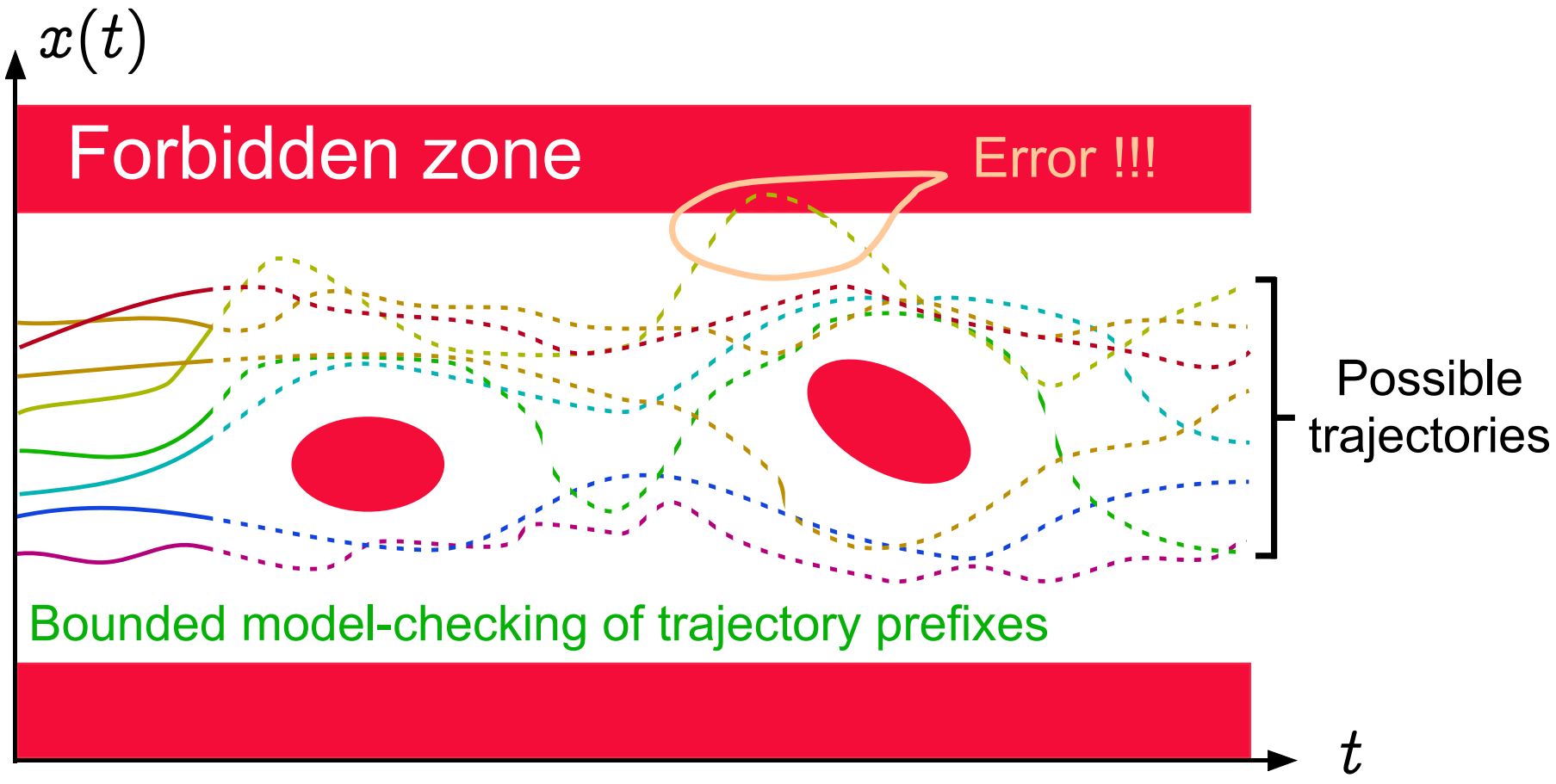


Safety property

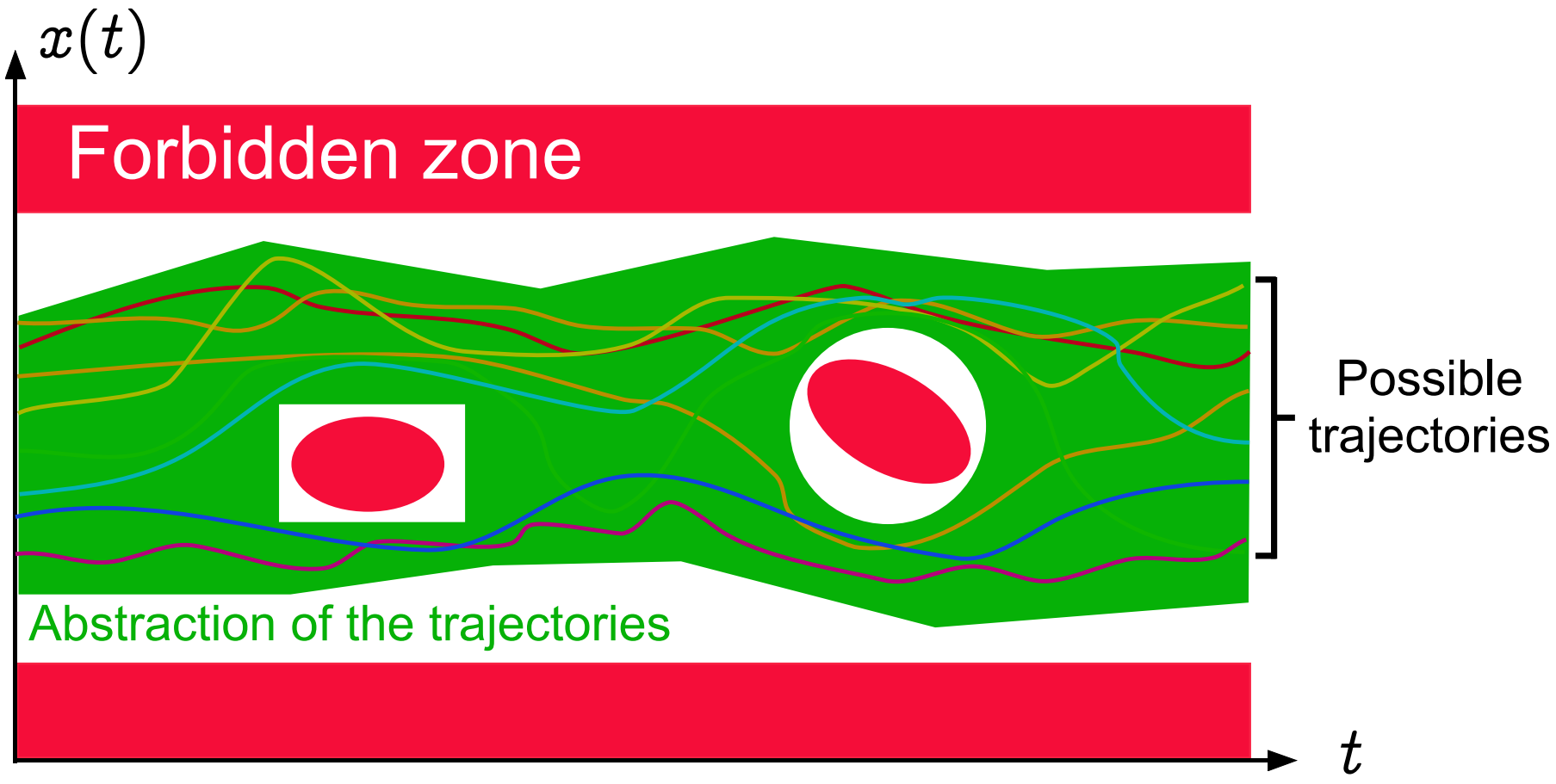
Test/Debugging is Unsafe



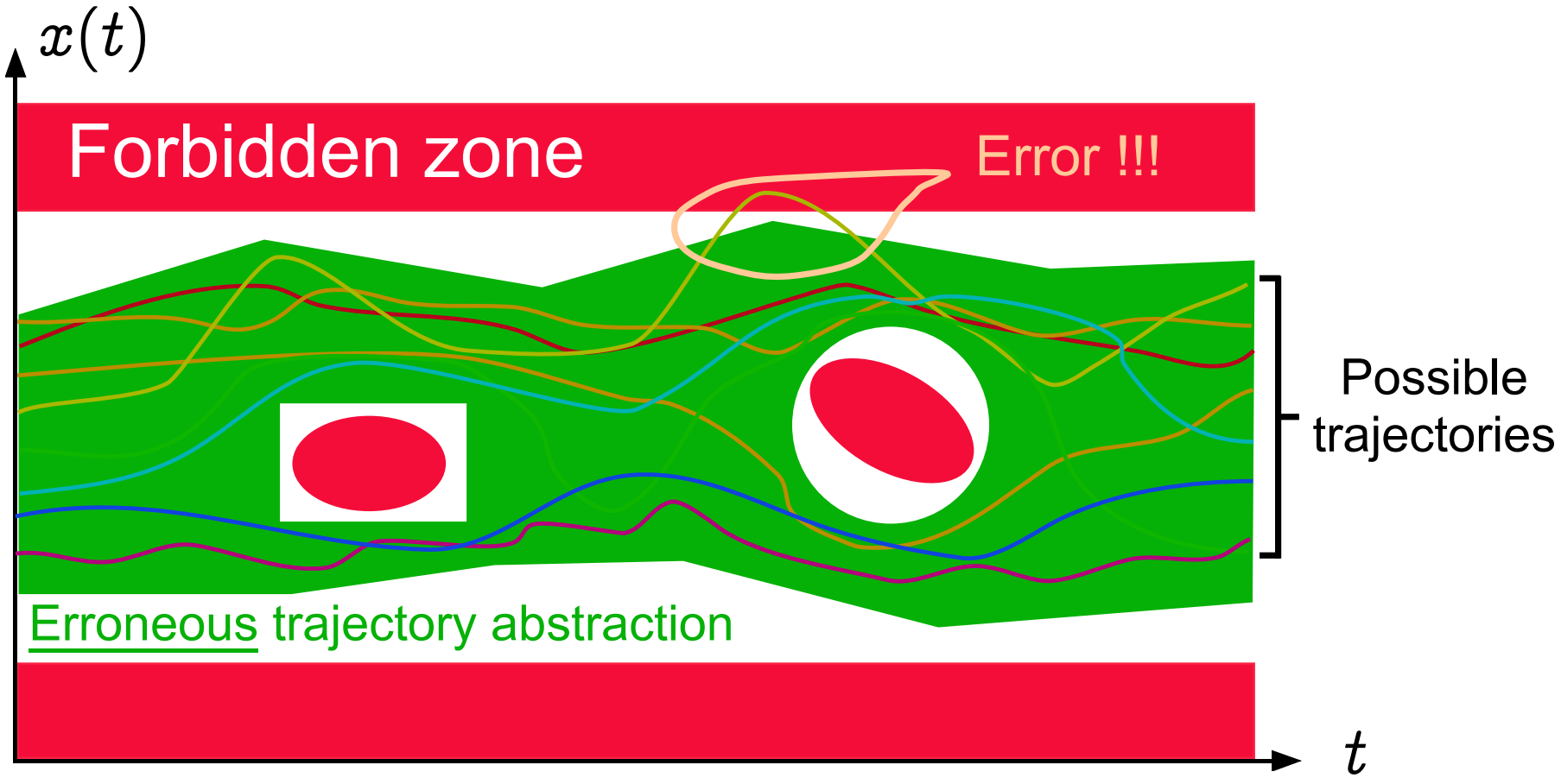
Bounded Model Checking is Unsafe



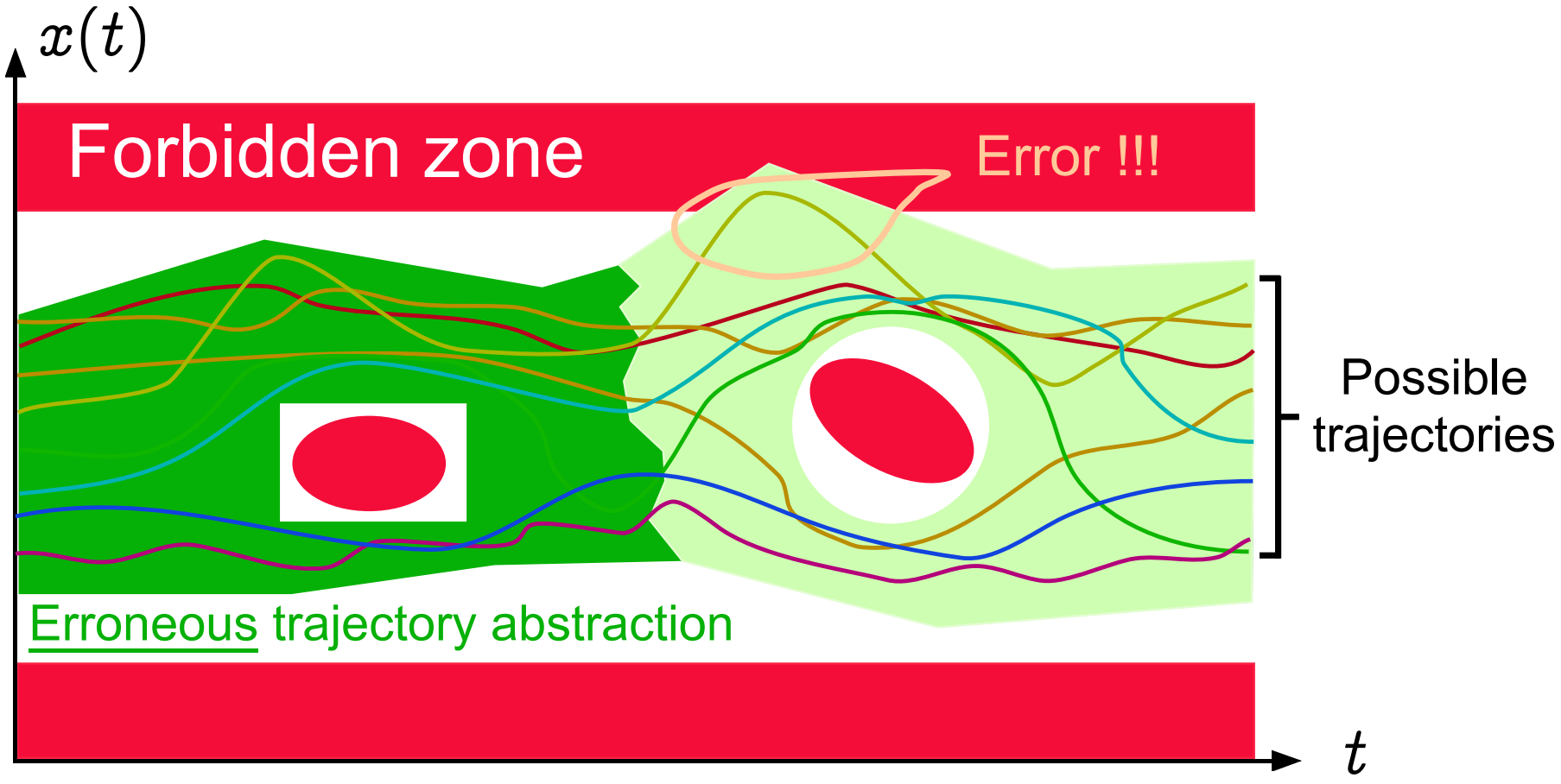
Abstract Interpretation



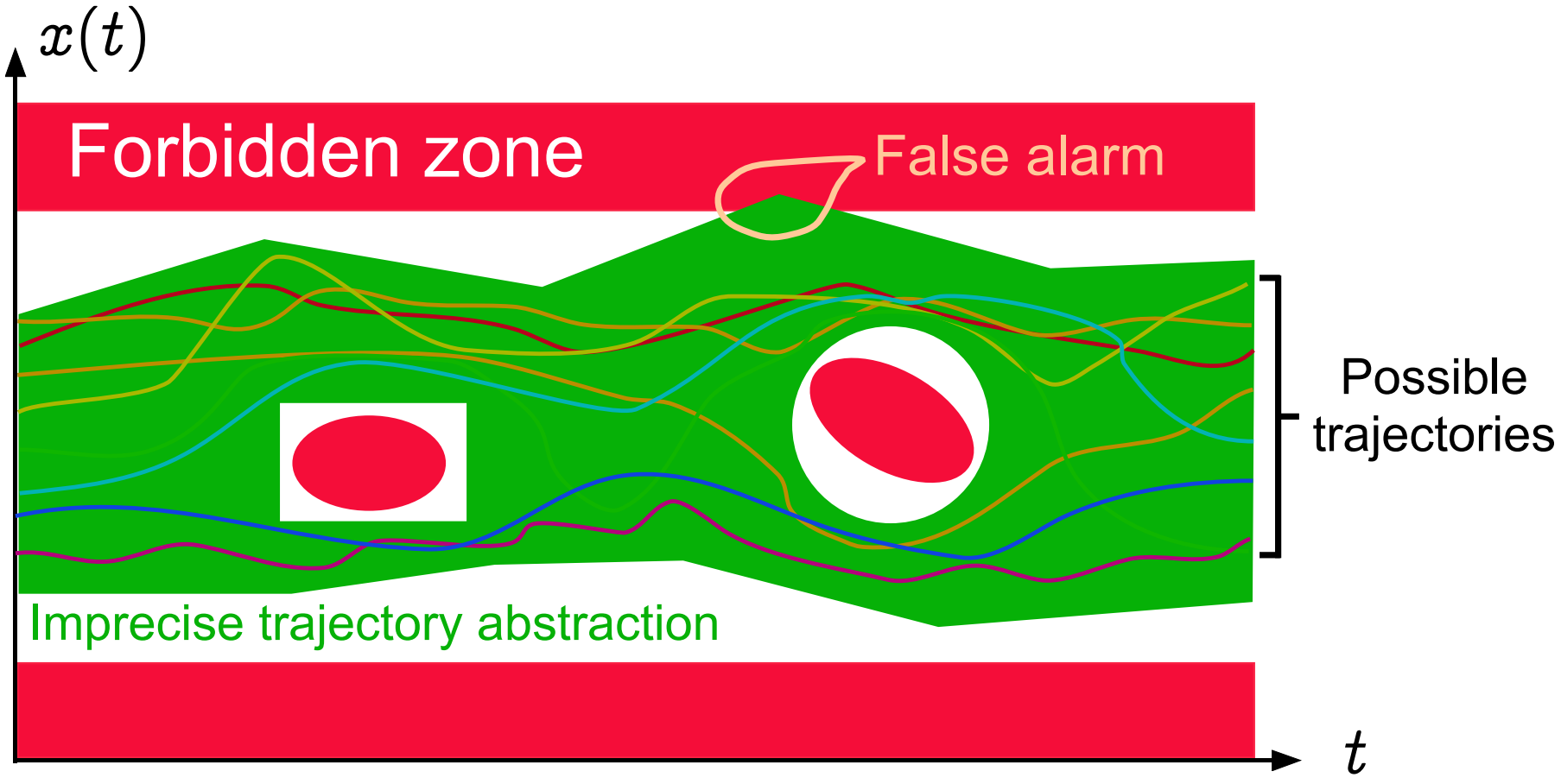
Soundness: Erroneous Abstraction — I



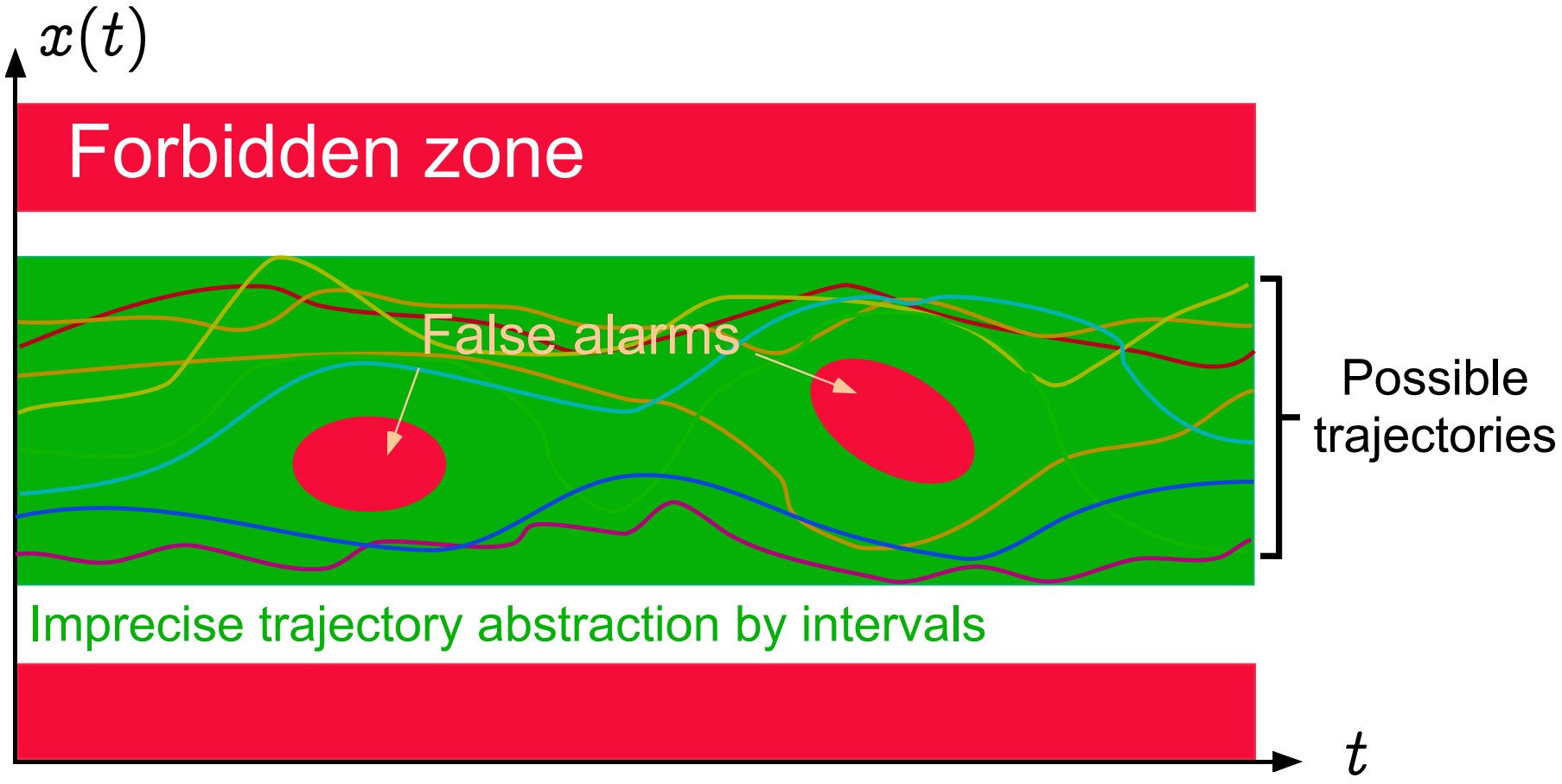
Soundness: Erroneous Abstraction — II



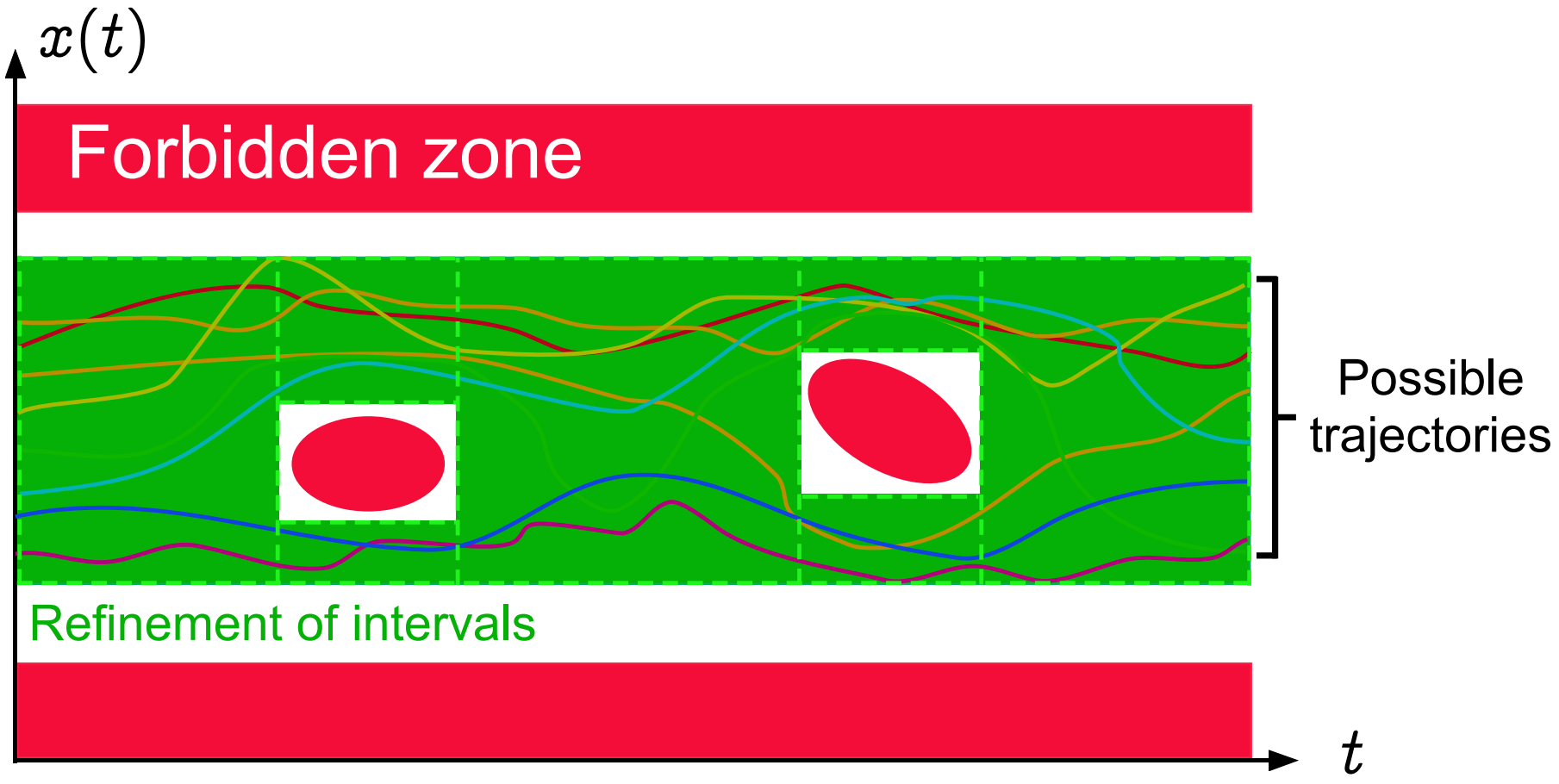
Imprecision $\Rightarrow$ False Alarms



Interval Abstraction $\Rightarrow$ False Alarms



Refinement by Partitioning



A Practical Application of Abstract Interpretation to the Verification of Safety Critical Embedded Control-Command Software

Reference

[1] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. *The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, LNCS 2566, pages 85–108. Springer, 2002.

[2] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. PLDI'03, San Diego, June 7–14, ACM Press, 2003.

ASTRÉE: A Sound, Automatic, Specializable, Domain-Aware, Parametric, Modular, Efficient and Precise Static Program Analyzer

www.astree.ens.fr

Implicit Specification: Absence of Runtime

- No violation of the norm of C (e.g. array index out of bounds)
- No implementation-specific undefined behaviors (e.g. maximum short integer is 32767)
- No violation of the programming guidelines (e.g. static variables cannot be assumed to be initialized to 0)
- No violation of the programmer assertions (must all be statically verified).

C language

<u>with</u>

- pointers (including on functions), structures and arrays
- floating point computations
- tests, loops and function calls
- limited branching (forward `goto`, `break`, `continue`)

<u>without</u>

- `union`
- dynamic memory allocation

– recursive function calls

– backward branching

– conflicting side effects [1]

– C libraries

[1] The ASTRÉE analyzer checks the absence of ambiguous side effects since otherwise the semantics of the C program would not be defined deterministically.

Operational semantics

- International norm of C (ISO/IEC 9899:1999)
- *restricted by* implementation-specific behaviors depending upon the machine and compiler (e.g. representation and size of integers, IEEE 754-1985 norm for floats and doubles)
- *restricted by* user-defined programming guidelines (such as no modular arithmetic for signed integers, even though this might be the hardware choice)

– *restricted by* program specific user requirements (e.g. `assert`)
– *restricted by* a volatile environment as specified by a *trusted* configuration file.

Example application

– Primary flight control software of the Airbus A340 family/A380 fly-by-wire system

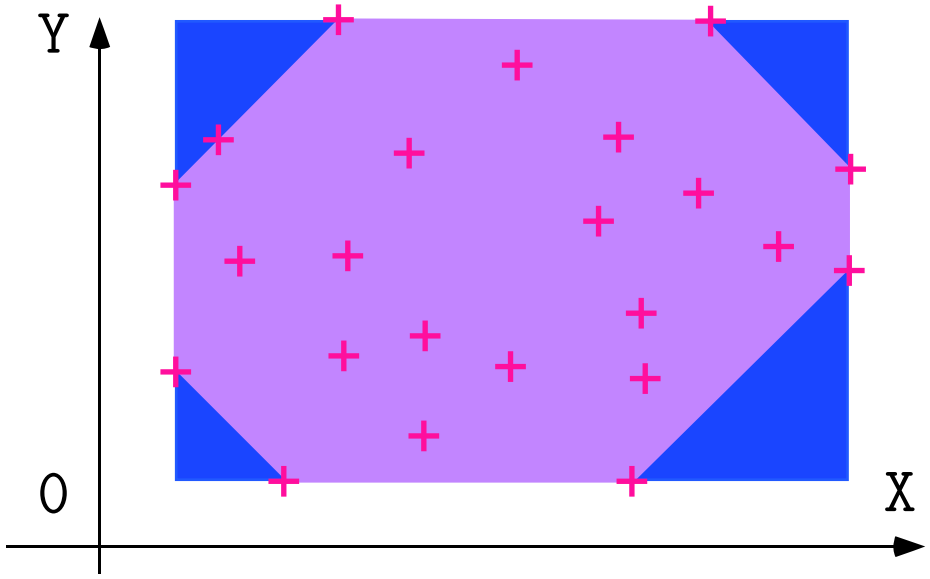
– C program, automatically generated from a proprietary high-level specification (à la Simulink/SCADE)

– A340 family: 132,000 lines, 75,000 LOCs after preprocessing, 10,000 global variables, over 21,000 after expansion of small arrays

– A380: $\times$ 3 $\Rightarrow$ No false alarm!

Examples

General-purpose abstract domains: intervals and octagons



Intervals:

$$\begin{cases} 1 \leq x \leq 9 \\ 1 \leq y \leq 20 \end{cases}$$

Octagons [?]:

$$\begin{cases} 1 \leq x \leq 9 \\ x + y \leq 75 \\ 1 \leq y \leq 20 \\ x - y \leq 04 \end{cases}$$

Difficulties: many global variables, IEEE 754 floating-point arithmetic (in program and analyzer)

Floating-Point Computations

– Code Sample:

```
/* float-error.c */
int main () {
  float x, y, z, r;
  x = 1.000000019e+38;
  y = x + 1.0e21;
  z = x - 1.0e21;
  r = y - z;
  printf("%f\n", r);
} % gcc float-error.c
% ./a.out
0.000000
```

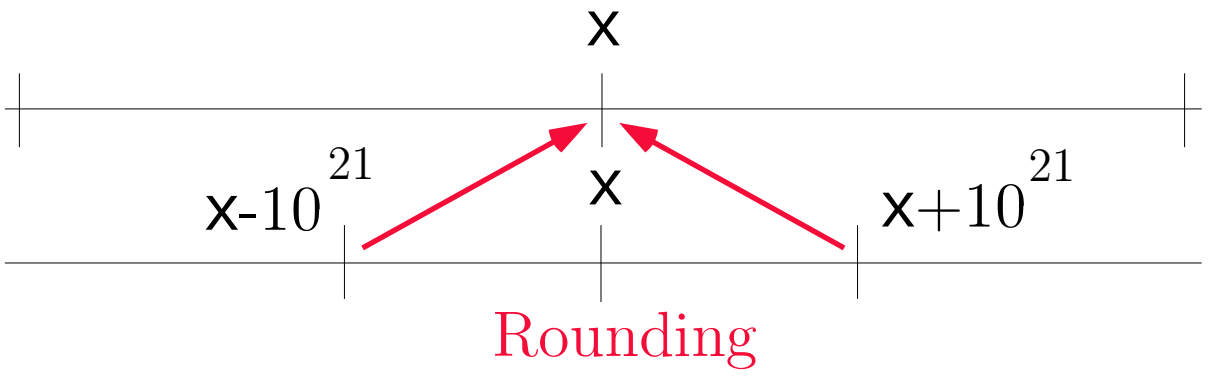
$$(x + a) - (x - a) \neq 2a$$

```
/* double-error.c */
int main () {
double x; float y, z, r;
/* x = ldexp(1.,50)+ldexp(1.,26); */
x = 1125899973951488.0;
y = x + 1;
z = x - 1;
r = y - z;
printf("%f\n", r);
}
% gcc double-error.c
% ./a.out
134217728.000000
```


Explanation of the huge rounding errors

(1) Floats

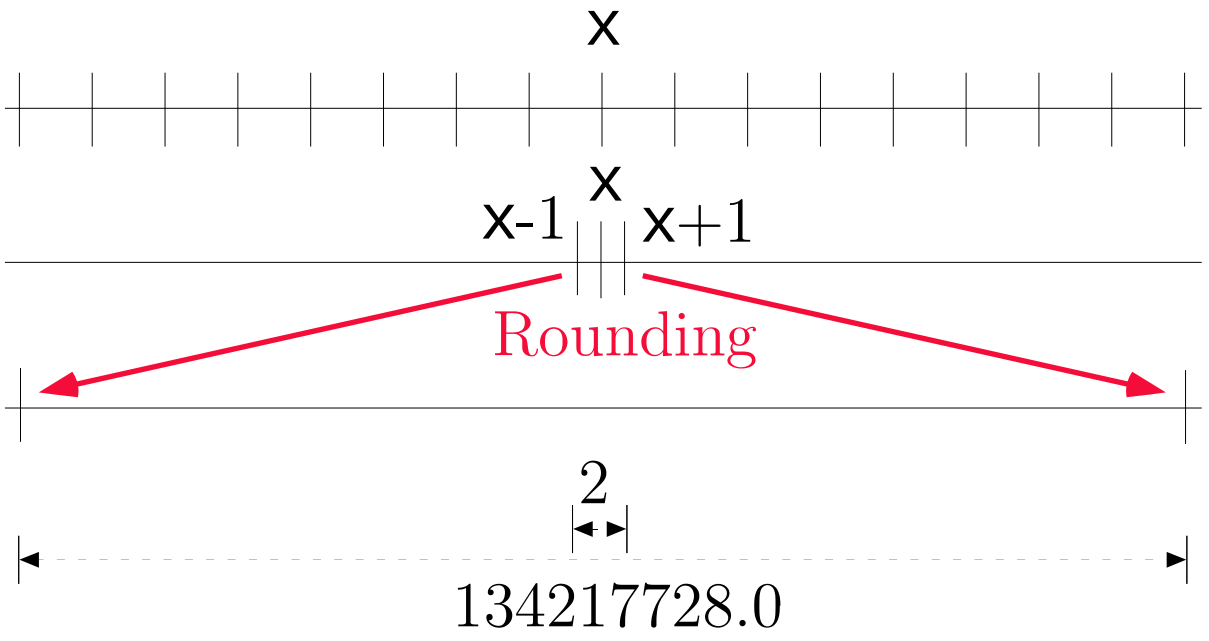
Reals



(2) Doubles

Reals

Floats



Clock abstract domain for counters

– Code Sample:

```
R = 0;
while (1) {
  if (I)
    { R = R+1; }
  else
    { R = 0; }
  T = (R>=n);
  wait_for_clock ();
}
```

- Output T is true iff the volatile input I has been true for the last n clock ticks.
- The clock ticks every s seconds for at most h hours, thus R is bounded.
- To prove that R cannot overflow, we must prove that R cannot exceed the elapsed clock ticks *(impossible using only intervals)*.

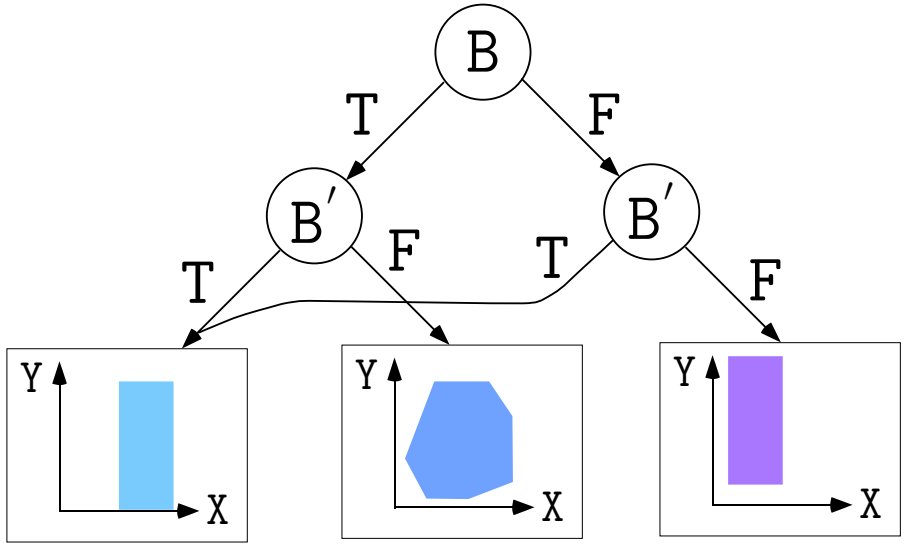
– Solution:

- Relate the value of variables X to the number clock of elapsed clock ticks.
- For example if X+clock or X-clock is bounded then so is X.

Boolean relations for boolean control

– Code Sample:

```
/* 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;
    }
    ...
  }
}
```



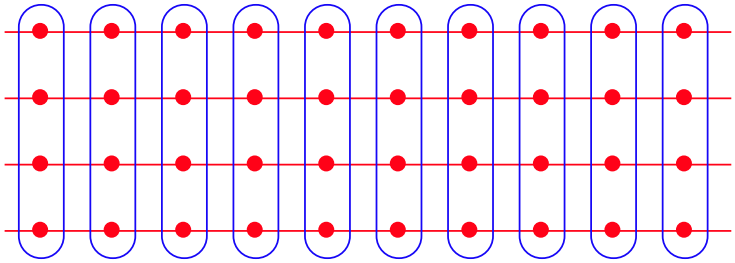
The boolean relation abstract domain is parameterized by the height of the decision tree (an analyzer option) and the abstract domain at the leafs

Control partitionning for case analysis

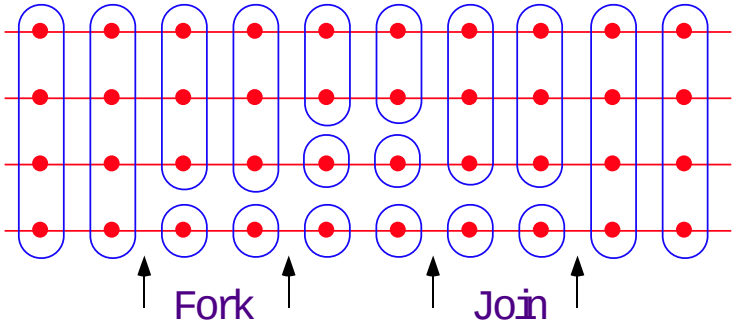
– Code Sample:

```
/* trace_partitionning.c */
void main() {
  float t[5] = {-10.0, -10.0, 0.0, 10.0, 10.0};
  float c[4] = {0.0, 2.0, 2.0, 0.0};
  float d[4] = {-20.0, -20.0, 0.0, 20.0};
  float x, r;
  int i = 0;

  ... found invariant −100 ≤ x ≤ 100 ...

  while ((i < 3) && (x >= t[i+1])) {
    i = i + 1;
  }
  r = (x - t[i]) * c[i] + d[i];
}
```

Control point partitionning:

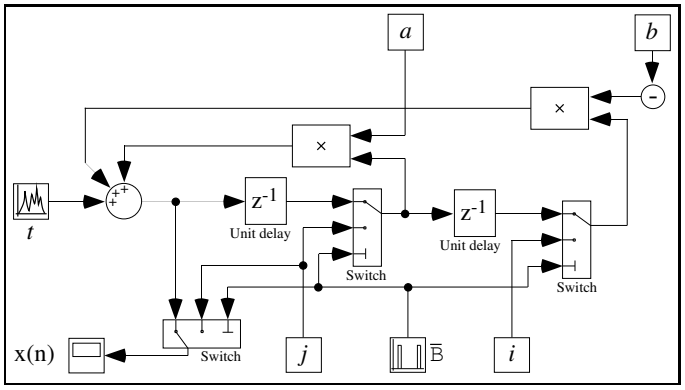


Trace partitionning:

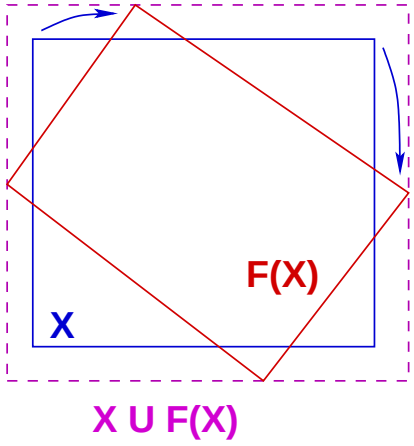


Ellipsoid abstract domain for filters

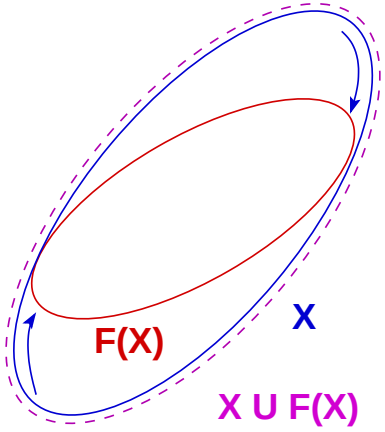
2^d Order Digital Filter:



- Computes $X_n = \begin{cases} \alpha X_{n-1} + \beta X_{n-2} + Y_n \\ I_n \end{cases}$
- The concrete computation is bounded, which must be proved in the abstract.
- There is no stable interval or octagon.
- The simplest stable surface is an ellipsoid.



unstable interval



stable ellipsoid

Arithmetic-geometric progressions

```
% 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;}
}
```

```
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
```


Conclusion

The Future & Grand Challenges

Forthcoming (1 year):

– More general memory model (`union`)

Future (5 years):

– Asynchronous concurrency (for less critical software)

– Functional properties (reactivity)

– Industrialization

Grand challenge:

– Verification from specifications to machine code (verifying compiler)

– Verification of systems (quasi-synchrony, distribution)

THE END, THANK YOU

More references at URL `www.di.ens.fr/~cousot`
`www.astree.ens.fr`.