Static Software Analysis, in the Large

Patrick Cousot

Distinguished Lecture Series at the MPI for Software Systems

Max Planck Institut für Informatik, Saarbrücken, August 26th, 2008

Abstract

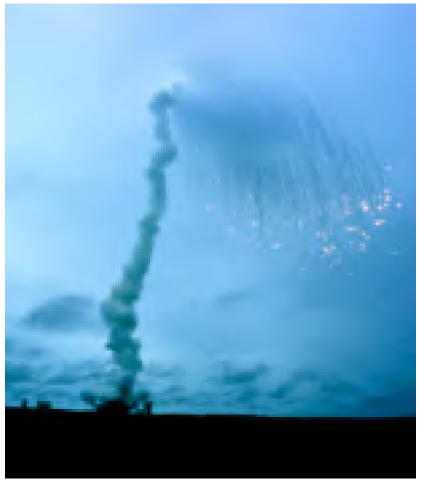
Static software analysis has known brilliant successes in the small, by proving complex program properties of programs of a few dozen or hundreds of lines, either by systematic exploration of the state space or by interactive deductive methods. To scale up is a definite problem. Very few static analyzers are able to scale up to millions of lines without sacrificing soundness and/or precision. Unsound static analysis may be useful for bug finding but is less useless in safety critical applications where the absence of bugs, at least of some categories of common bugs, should be formally verified.

After recalling the basic principles of abstract interpretation including the notions of abstraction, approximation, soundness, completeness, false alarm, etc., we introduce the domain-specific static analyzer ASTRÉE (www.astree.ens.fr) for proving the absence of runtime errors in safety critical real time embedded synchronous software in the large. The talk emphasizes soundness (no runtime error is ever omitted), parametrization (the ability to refine abstractions by options and analysis directives), extensibility (the easy incorporation of new abstractions to refine the approximation), precision (few or no false alarms for programs in the considered application domain) and scalability (the analyzer scales to millions of lines).

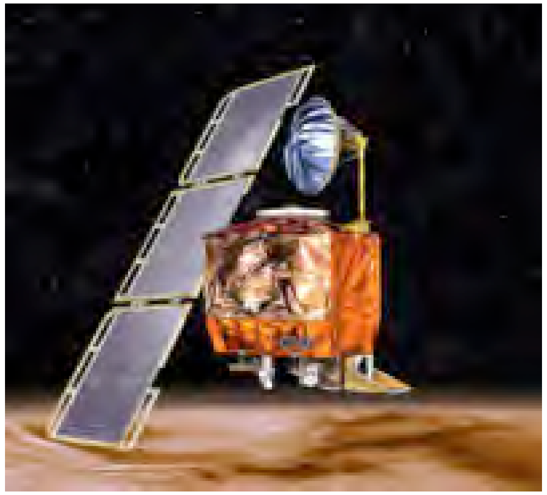
In conclusion, present-day software engineering methodology, which is based on the control of the design, coding and testing processes should evolve in the near future, to incorporate a systematic control of final software product thanks to domain-specific analyzers that scale up

1. Bugs

All Computer Scientists Have Experienced Bugs


Ariane 5.01 failure (overflow)

Patriot failure (float rounding)


Mars orbiter loss (unit error)

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

Principle of program verification

– Define a semantics of the language (that is the effect of executing programs of the language)

– Define a specification (example: absence of runtime errors such as division by zero, arithmetic overflow, etc)

– Make a formal proof that the semantics satisfies the specification

– Use a computer to automate the proof

– By undecidability (1), some form of approximation is inevitable!

(1) there are infinitely many programs for which a computer cannot prove them in finite time even with an infinite memory.

2. Abstract Interpretation

Reference

[1] P. Cousot. Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique de programmes. Thèse d'État ès sciences mathématiques. Université scientifique et médicale de Grenoble. 1978.

The Theory of Abstract Interpretation

- A theory of sound approximation of mathematical structures, in particular those involved in the description of the behavior of computer systems
- Systematic derivation of sound methods and algorithms for approximating undecidable or highly complex problems in various areas of computer science
- Main practical application is on the safety and security of complex hardware and software computer systems
- Abstraction: extracting information from a system description that is relevant to proving a property

Applications of Abstract Interpretation

- Static Program Analysis [CC77], [CH78], [CC79] including Dataflow Analysis; [CC79], [CC00], Set-based Analysis [CC95], Predicate Abstraction [Cou03], ...
- Grammar Analysis and Parsing [CC03];
- Hierarchies of Semantics and Proof Methods [CC92b], [Cou02];
- Typing & Type Inference [Cou97];
- (Abstract) Model Checking [CC00];
- Program Transformation (including program optimization, partial evaluation, etc) [CC02];

Applications of Abstract Interpretation (Cont'd)

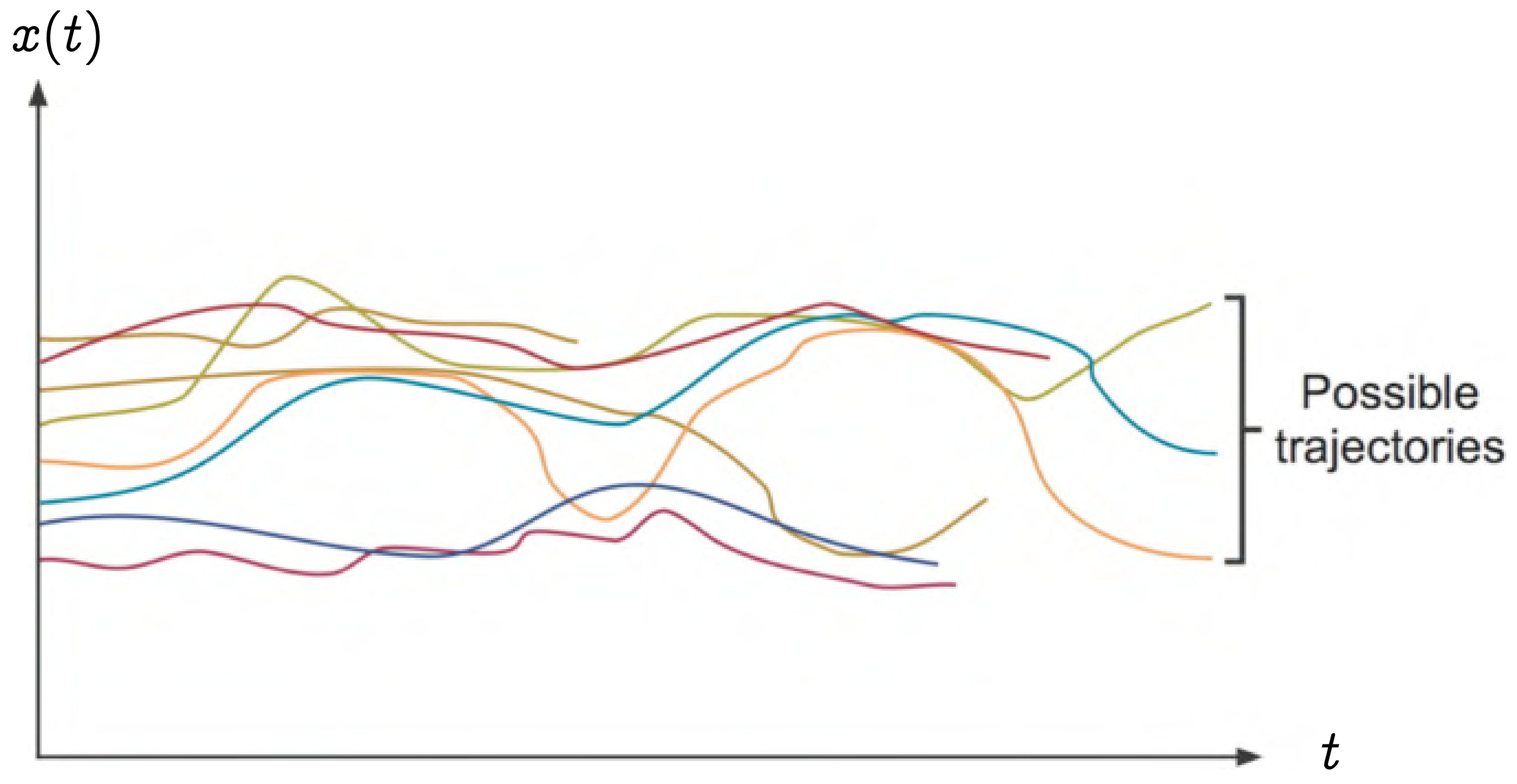
- Software Watermarking [CC04];
- Bisimulations [RT04, RT06];
- Language-based security [GM04];
- Semantics-based obfuscated malware detection [PCJD07].
- Databases [AGM93, BPC01, BS97]
- Computational biology [Dan07]
- Quantum computing [JP06, Per06]

All these techniques involve sound approximations that can be formalized by abstract interpretation

In this talk we concentrate on **— Sound Static Analysis —**

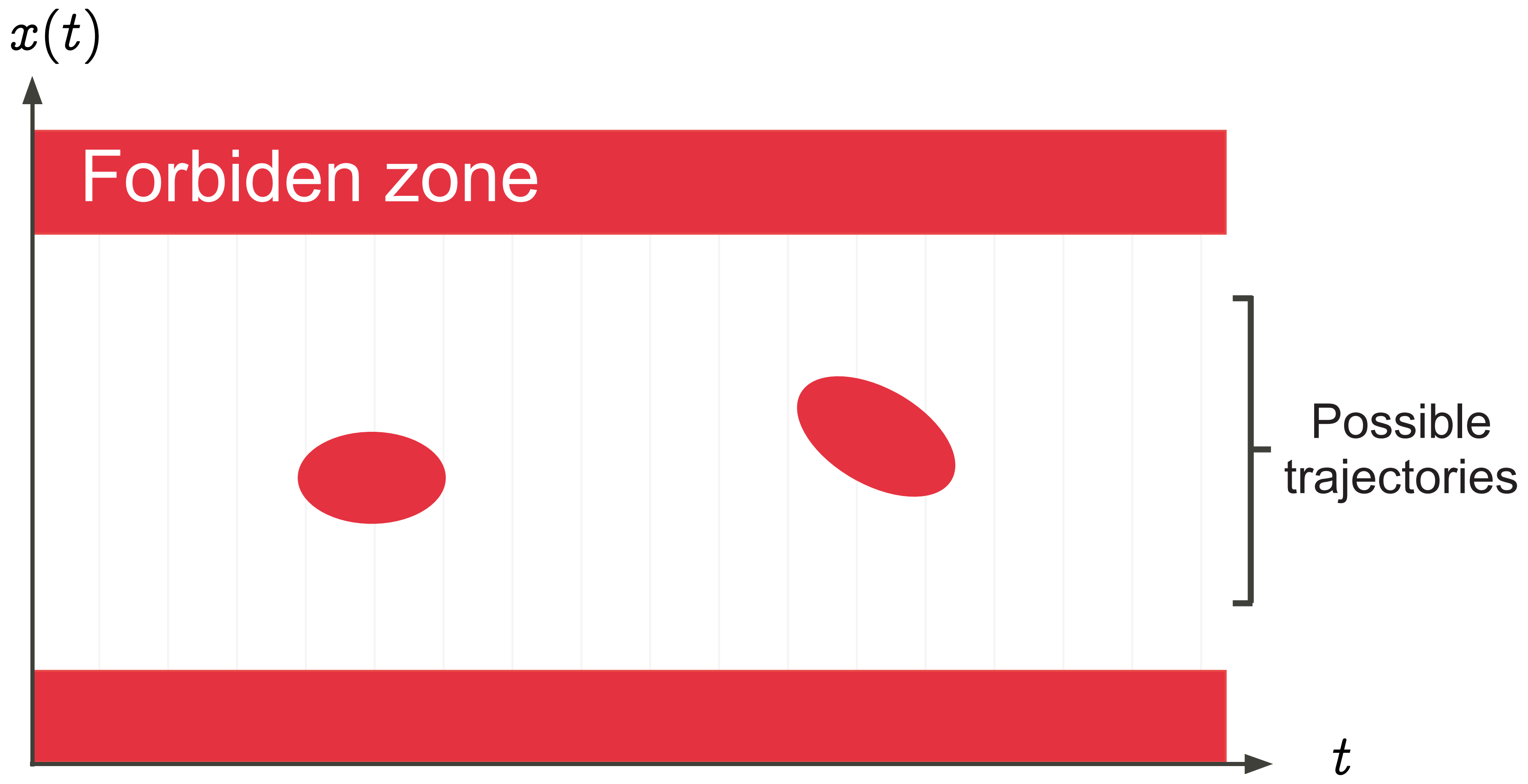
3. Principle of Static Analysis by Abstract Interpretation

Operational semantics of program P



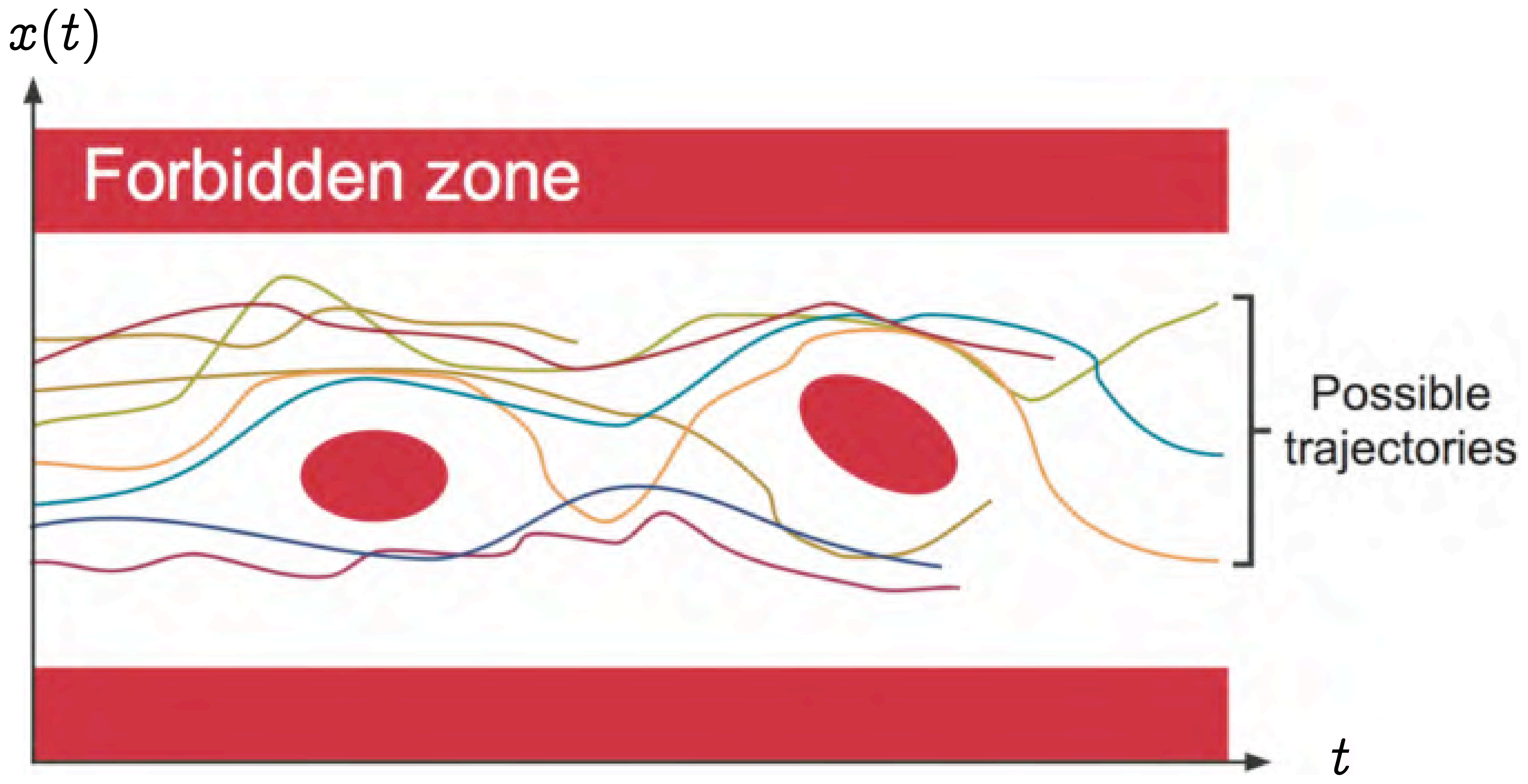
Semantics$[\![P]\!]$

Specification of program P



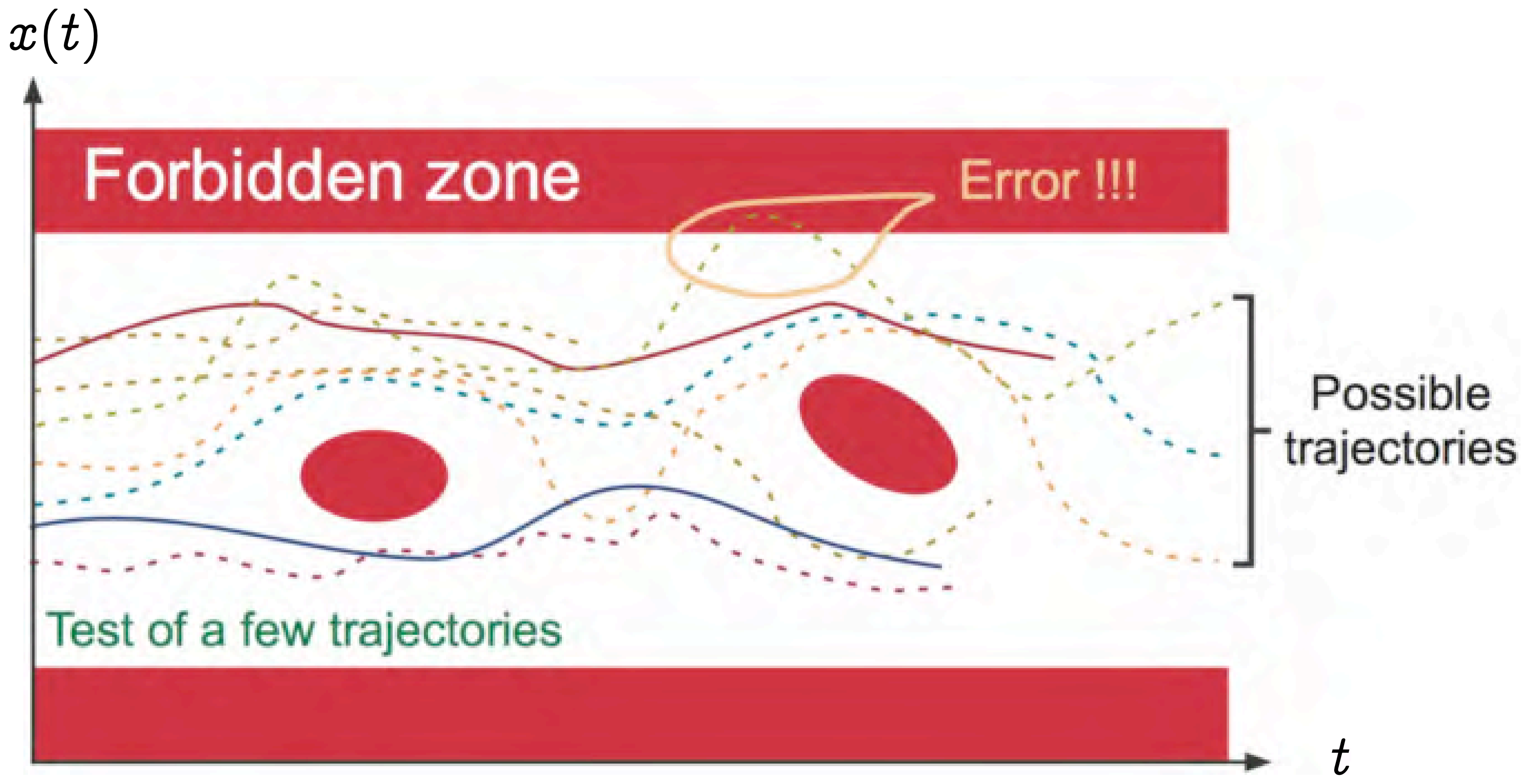
Specification$[\![P]\!]$

Formal proof of program P

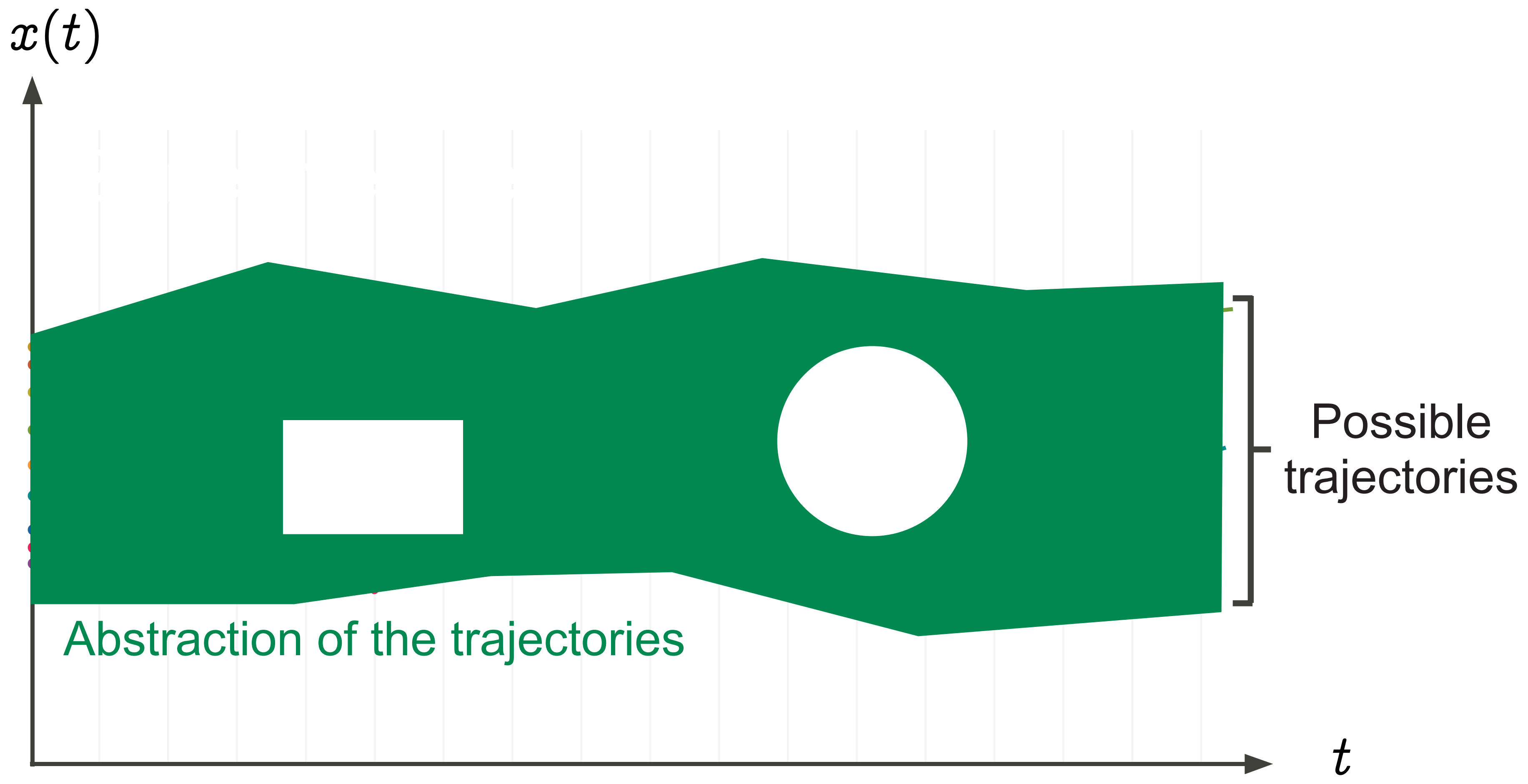


$$\text{Semantics}[\![P]\!] \subseteq \text{Specification}[\![P]\!]$$

Testing is incomplete

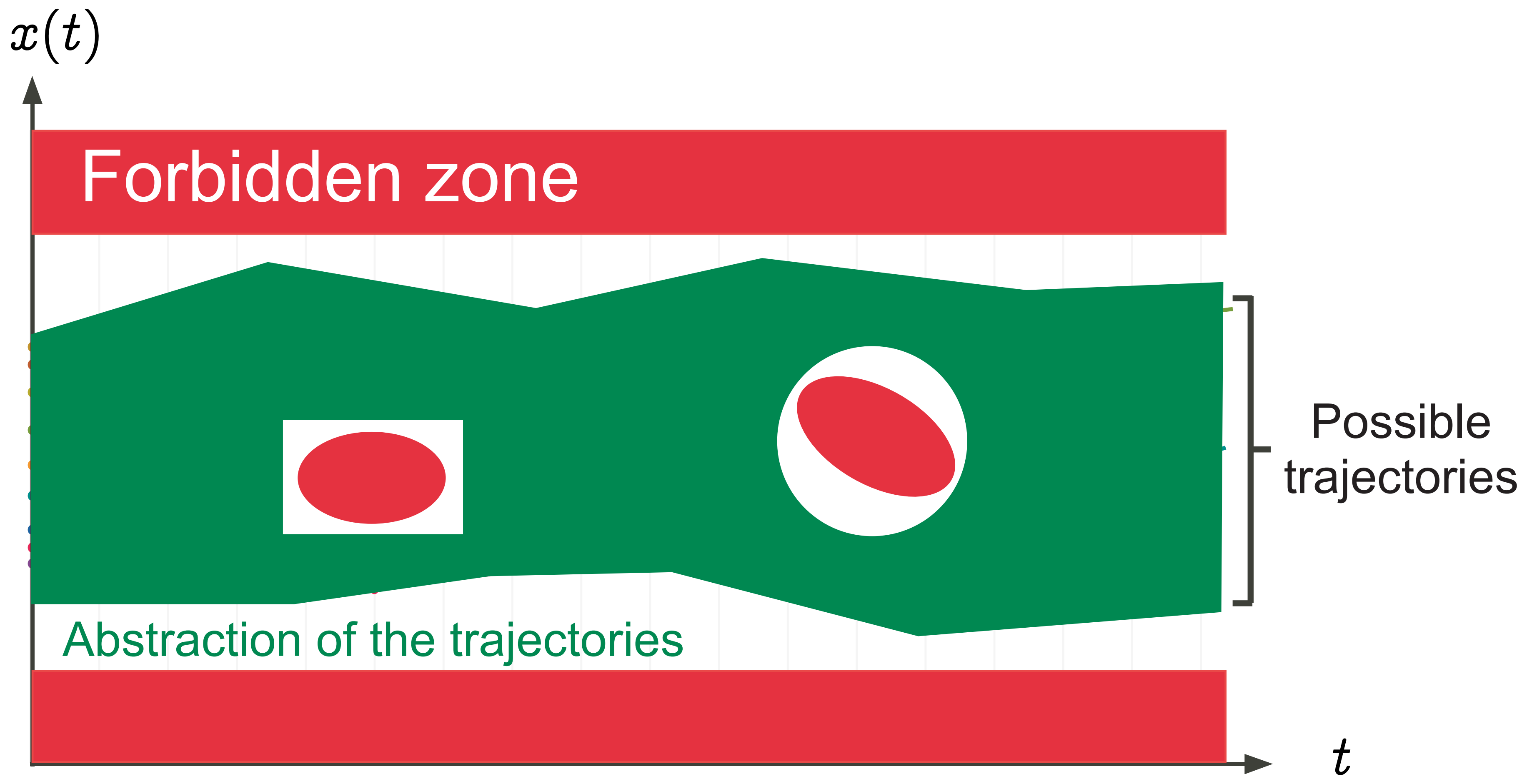


Abstraction of program P



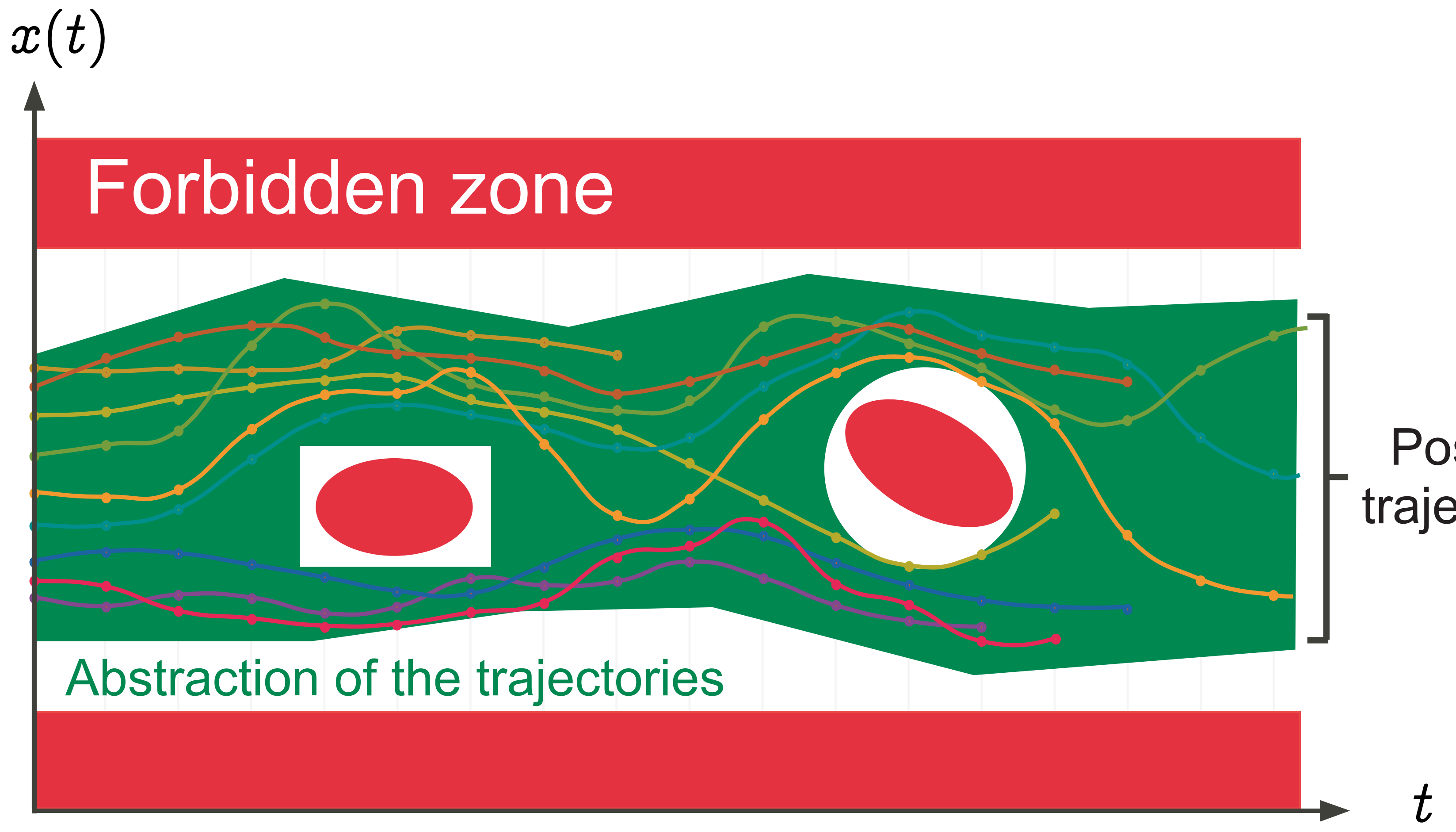
$\text{Abstraction}(\text{Semantics}[\![P]\!])$

Proof by abstraction



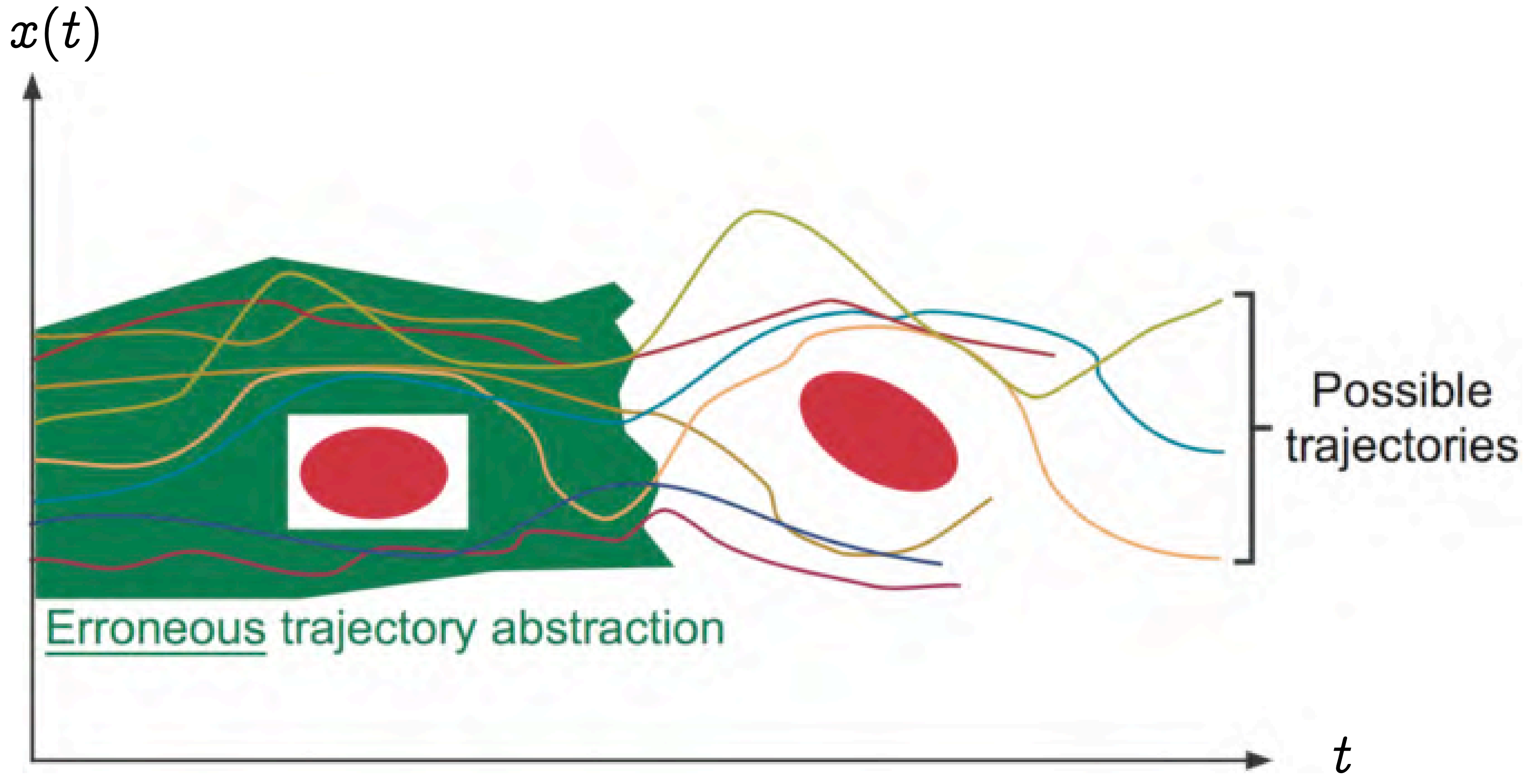
$$\text{Abstraction}(\text{Semantics}[\![P]\!]) \subseteq \text{Specification}[\![P]\!]$$

Abstract interpretation is sound



$$\text{Semantics}[\![P]\!] \subseteq \text{Abstraction}(\text{Semantics}[\![P]\!])$$

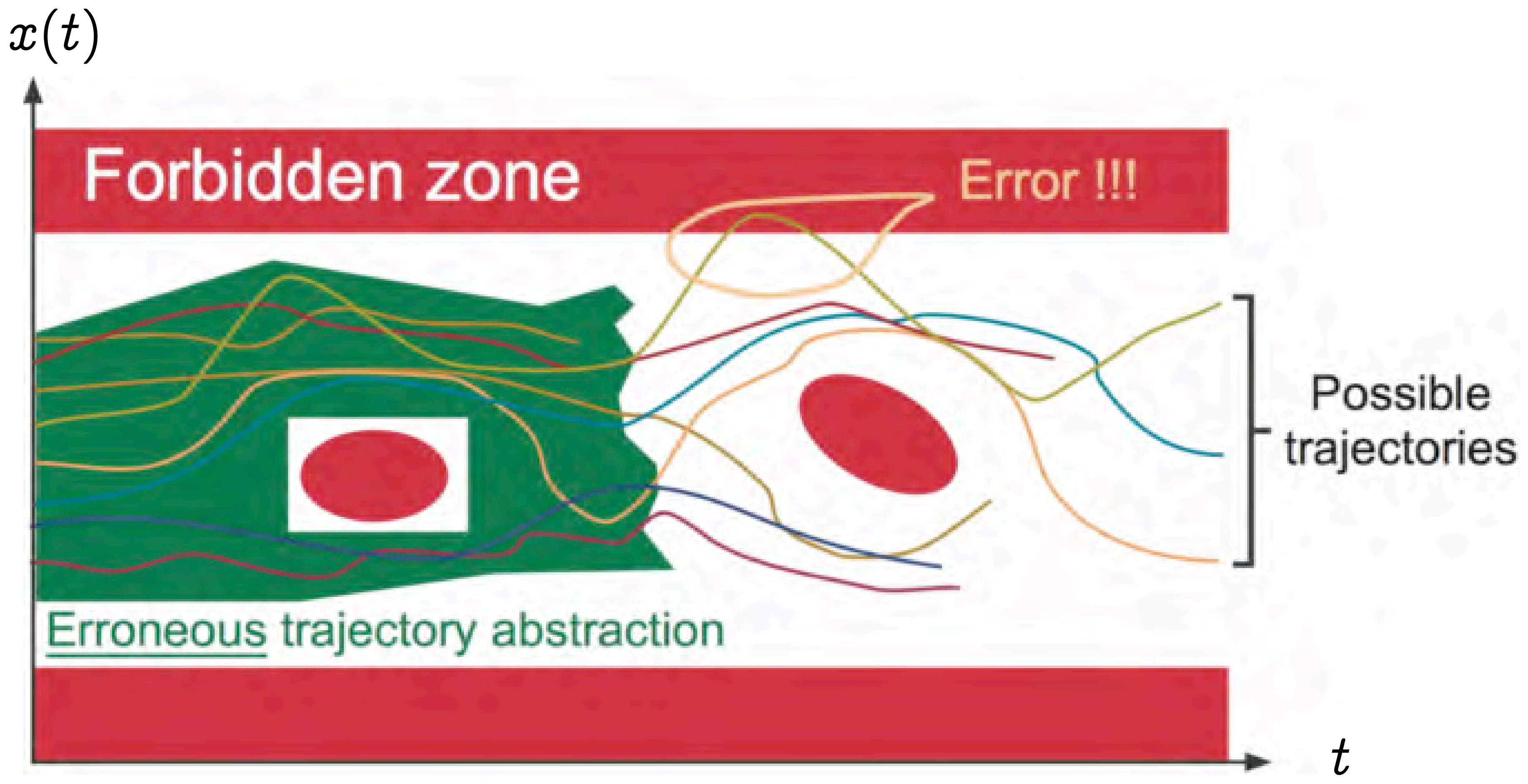
Example of unsound abstraction [2]



---

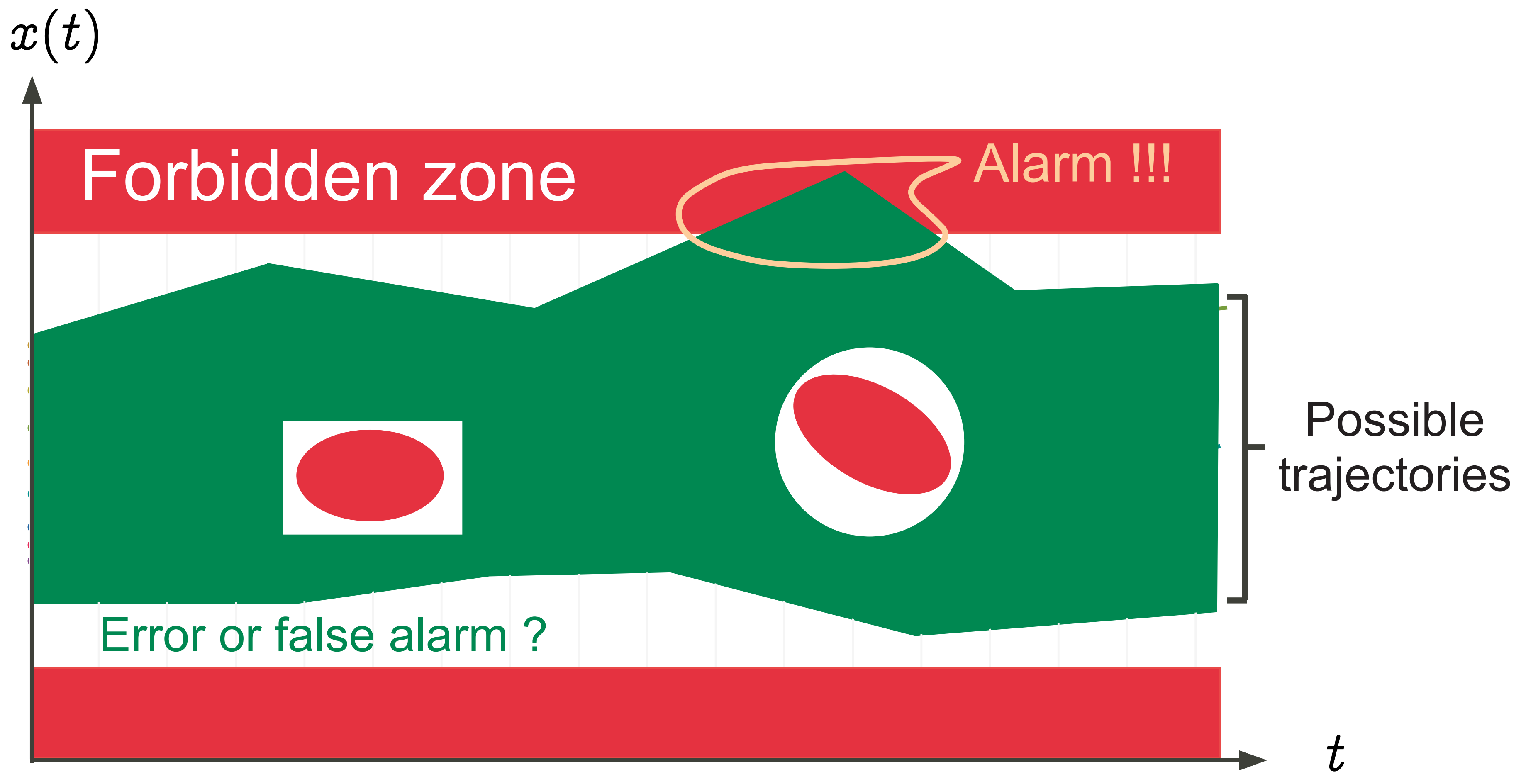
[2] Unsoundness is always excluded by abstract interpretation theory.

Unsound abstractions are inconclusive (false negatives) (2)



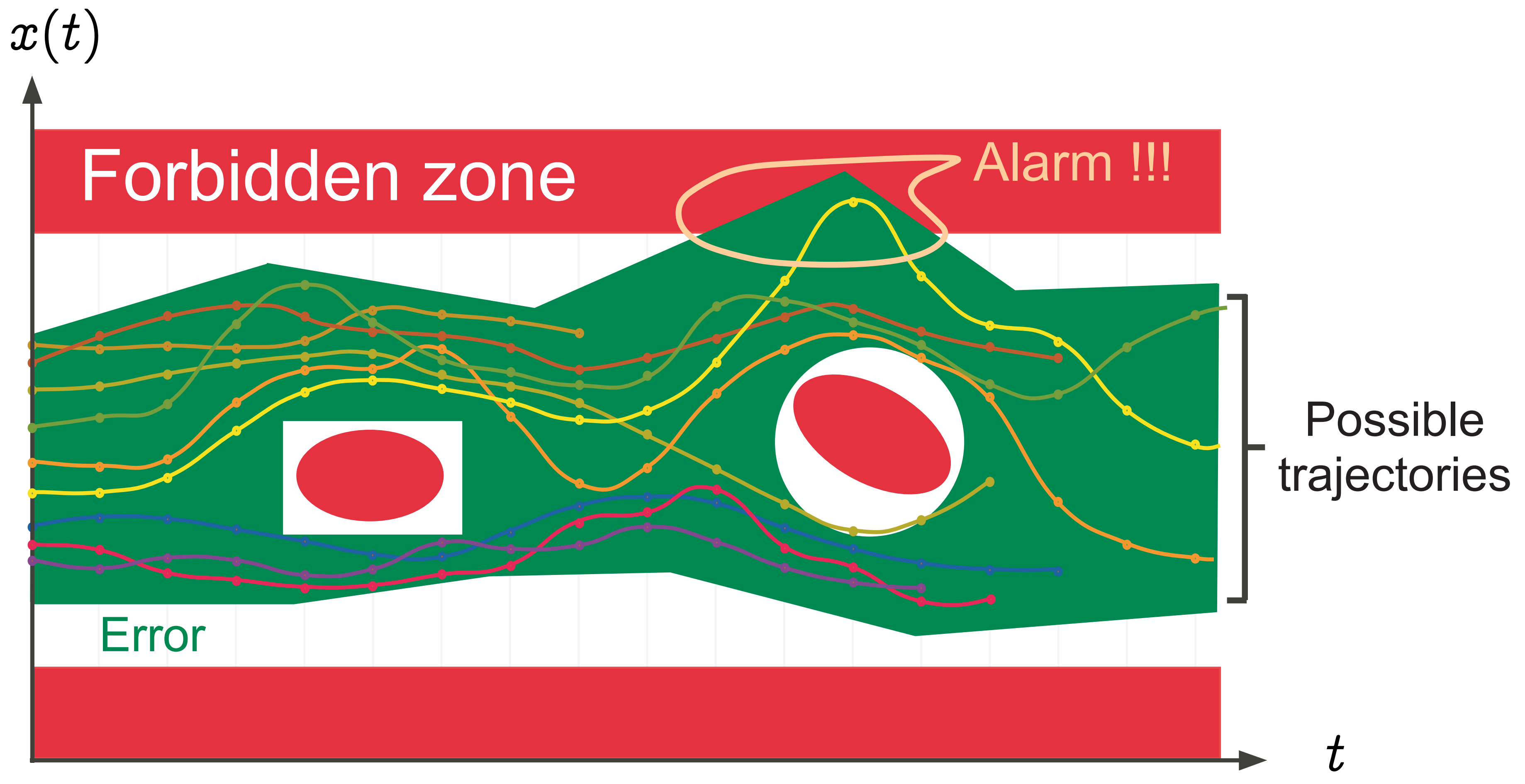
(2) Unsoundness is <u>always excluded</u> by abstract interpretation theory.

Alarm

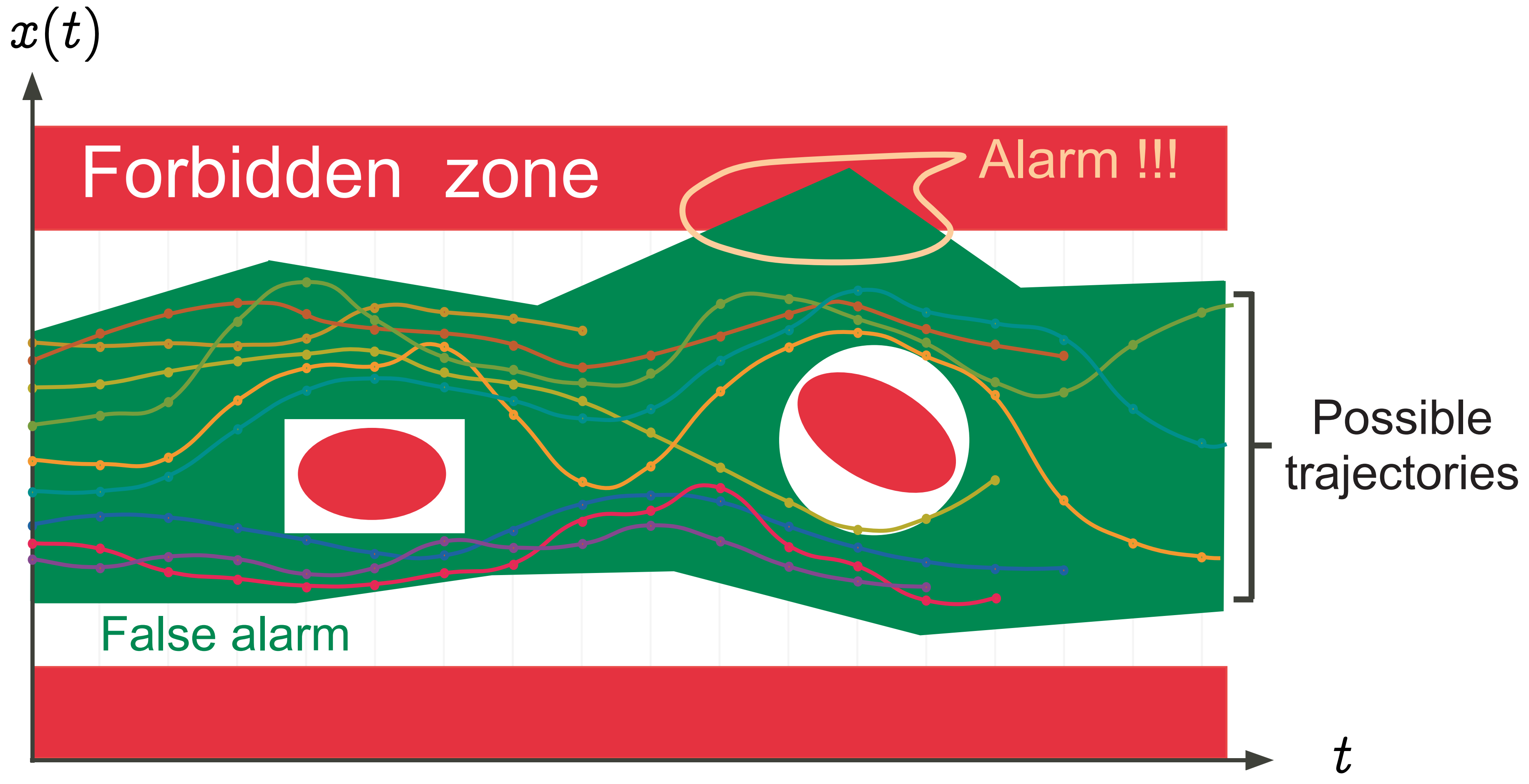


$x(t)$
Forbidden zone
Alarm !!!
Possible trajectories
Error or false alarm ?
t

An alarm can originate from an error



An alarm can originate from an over-approximation (false positive)



Abstraction is incomplete, a refinement is indispensable

Abstraction/Refinement

– The thorough design of a sound, precise and scalable abstraction is extremely difficult, even for a domain-specific family of programs

– We can proceed iteratively, starting from general abstractions

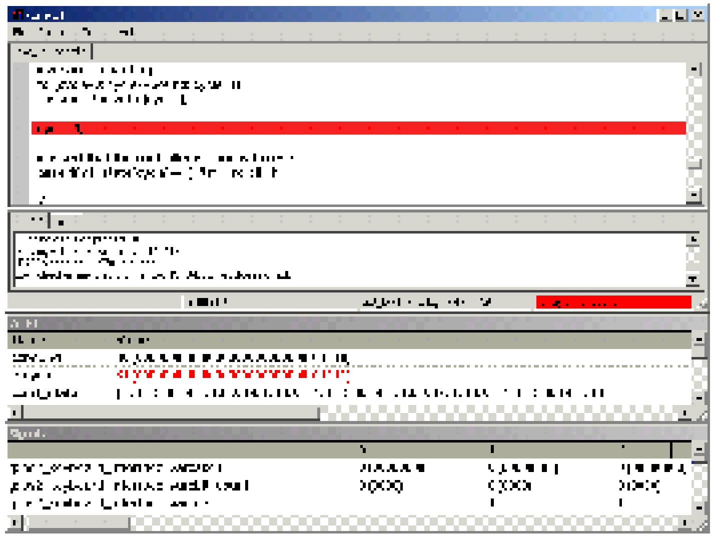
– In case of false alarm, the abstractions must be refined to be more precise

– The static analyzer must be designed to allow for easy incorporation of refined abstractions

4. Varieties of Static Analyzers

Example 1: CBMC

- CBMC is a Bounded Model Checker for ANSI-C programs (started at CMU in 1999).
- Allows verifying array bounds (buffer overflows), pointer safety, exceptions and user-specified assertions.
- Aimed for embedded software, also supports recursion and dynamic memory allocation using `malloc`.
- Done by unwinding the loops in the program and passing the resulting equation to a SAT solver.
- **Problem (a.o.): does not scale up!**



Example 2: Coverity Prevent™ Static Analysis

- Coverity Prevent™ Static Analysis offers (dixit) "the most precise static source code analysis solution available today" (started at Stanford by Dawson Engler around 2000).
- "Average false positive (FP) rate of about 15%, with some users reporting FP rates of as low as 5%."
- Integers overflows, arrays & pointer errors, memory leaks, deadlocks, race conditions, etc.
- Bug finding by local pattern matching, condition checking by SAT solver, and showing up the most probable errors.
- **Problem (a.o.): not sound, imprecise and endless!**

Example 3: Astrée

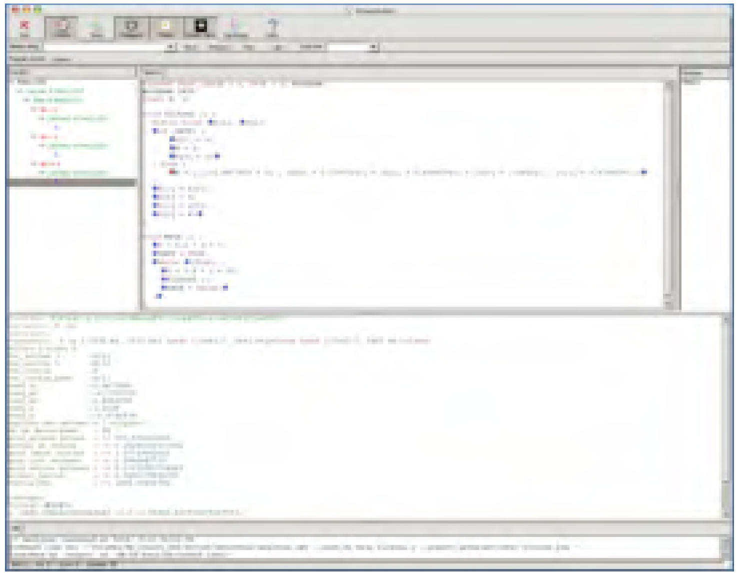
– Astrée is an abstract interpretation-based static analyzer for ANSI-C programs (started at ENS in 2001).

– Allows verifying array bounds (buffer overflows), pointer safety, exceptions and user-specified assertions.

– Aimed for embedded software, does not support recursion and dynamic memory allocation.

– Done by abstracting the reachability fixpoint equations for the program operational semantics.

– **Advantage (a.o.): sound, precise, and <u>does</u> scale up but domain-specific!**



5. Precision

Required Precision

- Coverity Prevent™ Static Analyzer has "an average FP rate of about 15%, with some users reporting FP rates of as low as 5%" [`www.coverity.com/html/prevent-for-c-features.html`]
- Consider a 1.000.000 LOCS control/command safety critical program, with 1 potential error per line (often much more)
- 5% FP = 5.000 false positives
- In safety critical software, false alarms must be justified for certification
- False/true alarms can take hours to days to be solved $\Longrightarrow$ the cost is several man $\times$ years!

6. Scaling up

Undecidability and complexity

– The mathematical proof problem is undecidable

– Even assuming finite states, the complexity is much too high for combinatorial exploration to succeed

– Example: 1.000.000 lines $\times$ 50.000 variables $\times$ 64 bits $\simeq 10^{27}$ states

– Exploring 10^{15} states per seconde, one would need 10^{12} s $>$ 300 centuries (and a lot of memory)!

A typical small control/command program ...

```
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;
  /* P in [-1325.4522, 1325.4522] */
}
int main () {
  int i = 1; X = 5.0; INIT = TRUE;
  while (i < 3600000) {/* simulated 10ms clock tick for 10 hours */
    X = 0.9 * X + 35; /* simulated filter input */
    filter (); INIT = FALSE; i++; }
}
```

… Analysis with CBMC

```
Script started on Tue Jul 29 23:44:00 2008
% time ./cbmc filter.c
...
Starting Bounded Model Checking
Unwinding loop 1 iteration 1
Unwinding loop 1 iteration 2
...
Unwinding loop 1 iteration 95479
cbmc(34799) malloc: *** mmap(size=2097152) failed (error code=12)
*** error: can't allocate region
*** set a breakpoint in malloc_error_break to debug
terminate called after throwing an instance of 'std::bad_alloc'
  what():  St9bad_alloc
...
Abort
29668.051u 101.916s 8:20:41.88 99.0% 0+0k 1+10io 2680pf+0w
% ^Dexit
Script done on Wed Jul 30 09:08:58 2008
```

… Analysis with ASTRÉE

```
% diff -U1 filter.c filter-a.c
-- filter.c 2008-07-30 11:33:13.000000000 +0200
+++ filter-a.c 2008-07-30 12:22:26.000000000 +0200
@@ -8,2 +8,3 @@
   E[1] = E[0]; E[0] = X; S[1] = S[0]; S[0] = P;
+  __ASTREE_log_vars((P));
   /* P in [-1325.4522, 1325.4522] */
```

– Fast:

```
% (time astree -exec-fn main filter-a.c) |& egrep "WARN|pf+"
0.710u 0.085s 0:01.47 53.7% 0+0k 7+7io 840pf+0w
%
```

– Precise:

```
% astree -exec-fn main filter-a.c |& grep "P in" | tail -n1
direct = <float-interval: P in [-1325.4522, 1325.4522] >
```

The difficulty of scaling up

– The abstraction must be coarse enough to be effectively computable with reasonable resources

– The abstraction must be precise enough to avoid false alarms

– Abstractions to *infinite domains with widenings* are more expressive than abstractions to *finite domains* [3] (when considering the analysis of a programming language) [CC92a]

– Abstractions are ultimately incomplete (even intrinsically for some semantics and specifications [CC00])

[3] e.g. predicate abstraction which always abstract to a finite domain.

A common believe 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 ($P \in [-1325.4522, 1325.4522]$)
- 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.

7. Soundness

Is the virtue of soundness a myth?

Why bother about soundness since automatic static analyzers cannot prove total correctness anyway? Finding as many bugs as possible is the most direct approximation! [3]

– We can focus on a well-defined category of bugs (e.g. runtime errors, time overrun, etc)

– And ensure no bug is left in this category

– And, more importantly, know when the verification should be stopped for that category of bugs (contrary to unsound methods like testing/bug finding)

Reference

[3] Madanlal Musuvathi and Dawson R. Engler. Some lessons from using static analysis and software model checking for bug finding. SoftMC 2003 workshop. July 18–19, 2002, Boulder, CO, USA.

8. Abstraction Completion / Refinement

Abstraction completion

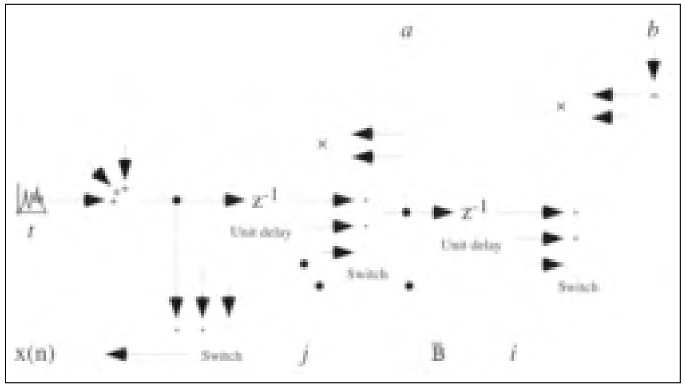
- Completion is the process of refining an abstraction of a semantics until a specification can be proved [CC79, GRS00]
- In theory, always possible by an infinite fixpoint computation *in the concrete!* [Cou00, GRS00]
- In complicated cases, the most abstract complete refined abstraction is *identity* (in which case the refinement ultimately amounts to computing the collecting semantics)
- Examples of *refinement semi-algorithms*:
    - counter-example-guided abstraction refinement [CGJ$^+$00]
    - fixpoint abstraction refinement [CGR07]

The limits of fixpoint abstraction completion

– Abstraction completion algorithms have misunderstood *severe limits*:
  - the refinement may be useless (corrected in [CGR07])
  - may not terminate (by ultimately computing in the infinite collecting semantics)
  - cannot to pass to the limit
  - cannot invent
    - efficient data representations of refined abstract properties (rely on state enumeration)
    - effective abstract transformer algorithms (rely on set of states transformers)

Example of Refinement: Ellipsoid Abstract Domain for Filters

2^d Order Digital Filter:

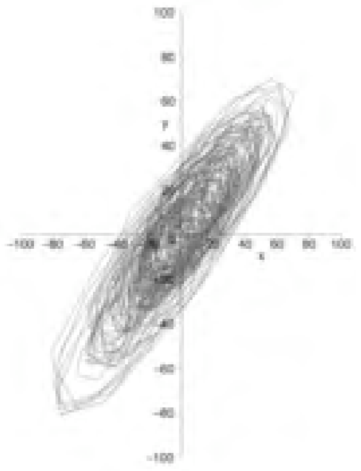


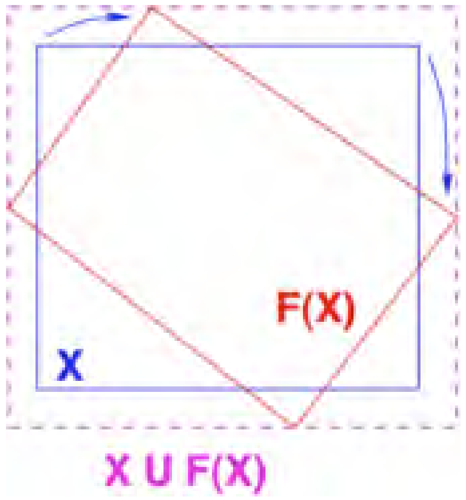
– Computes $X_n = \begin{cases} \alpha X_{n-1} + \beta X_{n-2} + Y_n \\ I_n \end{cases}$

– The concrete computation is bounded, which must be proved in the abstract.

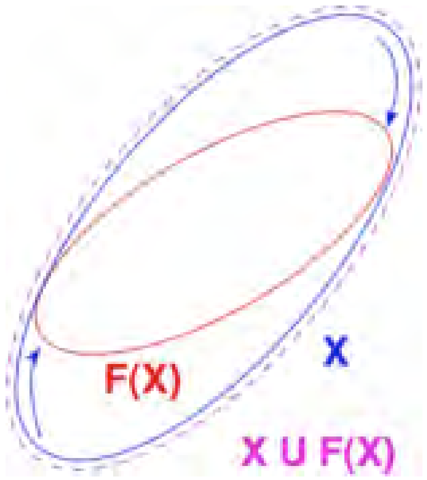
– There is no stable interval or octagon.

– The simplest stable surface is an ellipsoid.


execution trace



unstable interval



stable ellipsoid

9. (Concrete) Semantics

The Semantics of C is Hard (Ex. 1: Floats)

"*Put* $\mathtt{x}$ *in* $[\mathtt{m}, \mathtt{M}]$ *modulo* $(\mathtt{M} - \mathtt{m})$":

```
y = x - (int) ((x-m)/(M-m))*(M-m);
```

- The programmer thinks $\mathtt{y} \in [\mathtt{m}, \mathtt{M}]$
- But with $\mathtt{M} = 4095$, $\mathtt{m} = -\mathtt{M}$, IEEE double precision, and $\mathtt{x}$ is the greatest float strictly less than $\mathtt{M}$, then $\mathtt{x'} = \mathtt{m} - \epsilon$ (ϵ small).

```
% cat -n modulo.c
#include <stdio.h>
#include <math.h>
int main () {
   float m, M, x, y; M = 4095.0; m = -M;
   x = 4094.9997558593750; /* largest float strictly less than M */
   y = x - (int) ((x-m)/(M-m))*(M-m);
printf("%.20f\n",y);
}
% gcc modulo.c; ./a.out
-4095.00024414062500000000
%
```

Analysis by ASTRÉE

```
% cat modulo-a.c
int main () {
   float m, M, x, y;
M = 4095.0; m = -M;
x = 4094.9997558593750; /* largest float strictly less than M */
   y = x - (int) ((x-m)/(M-m))*(M-m);
__ASTREE_log_vars((y));
}
% astree -exec-fn main -print-float-digits 25 modulo-a.c |& grep "y in"
direct = <float-interval: y in [-4095.000244140625, 4094.999755859375] >
%
```

The Semantics of C is Hard (Ex. 2: Runtime Errors)

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
```

Analysis by Astrée

```
% cat -n unpreditable-a.c
     1  const int false = 0;
     2  int main () { int n, T[1], x;
     3  n = 1;
     4  x = T[n];
     5  __ASTREE_assert((false));
     6  }
% astree -exec-fn main unpreditable-a.c |& grep "WARN"
unpreditable-a.c:4.4-8::[call#main@2:]: WARN: invalid dereference: dereferencing
4 byte(s) at offset(s) [4;4] may overflow the variable T of byte-size 4
%
```

No alarm on `assert(false)` because execution is assumed to stop after a definite runtime error with unpredictable results [(4)].

---

(4) Equivalent semantics if no alarm.

Different Classes of Run-time Errors

1. Errors terminating the execution [5]. ASTRÉE warns and continues by taking into account only the executions that did not trigger the error.
2. Errors not terminating the execution with predictable outcome [6]. ASTRÉE warns and continues with worst-case assumptions.
3. Errors not terminating the execution with <u>unpredictable</u> outcome [7]. ASTRÉE warns and continues by taking into account only the executions that did not trigger the error.

$\Rightarrow$ ASTRÉE is sound with respect to C standard, unsound with respect to C implementation, <u>unless</u> no false alarm of type 3.

(5) floating-point exceptions e.g. (invalid operations, overflows, etc.) when traps are activated

(6) e.g. overflows over signed integers resulting in some signed integer.

(7) e.g. memory corruptionss.

10. Specification

Implicit Specification: Absence of Runtime Errors

The static analyzer should definitely guarantee the absence of

- violations of the norm of C (e.g. array index out of bounds, division by zero, nil/dangling pointer dereferencing)
- implementation-specific undefined behaviors (e.g. maximum short integer is 32767, NaN)
- violations of the programming guidelines (e.g. no modulo arithmetics for signed integers)
- violations of the programmer assertions (must all be statically verified).

for all reachable states during any execution [8]

[8] May be restricted by hypotheses on a few inputs and timing given in a *configuration file*

11. The design of ASTRÉE for soundness, precision, scalability, and refinability

Modular refinable abstraction

The abstract semantics is decomposed into:

– A structural fixpoint iterator (by composition on the program syntax)

– A collection of parametric abstract domains with:

  - parameters to adjust the expressivity of the abstraction
  - parametric convergence acceleration (parameters to adjust the frequence and precision of widenings/narrowings)
  - analysis directives (to locally adjust the choice of abstractions)

– A reduction performing the conjunction of the abstractions

$\Rightarrow$ Easily refinable by parameter/directive adjustment and extendable by addition of new abstract domains!

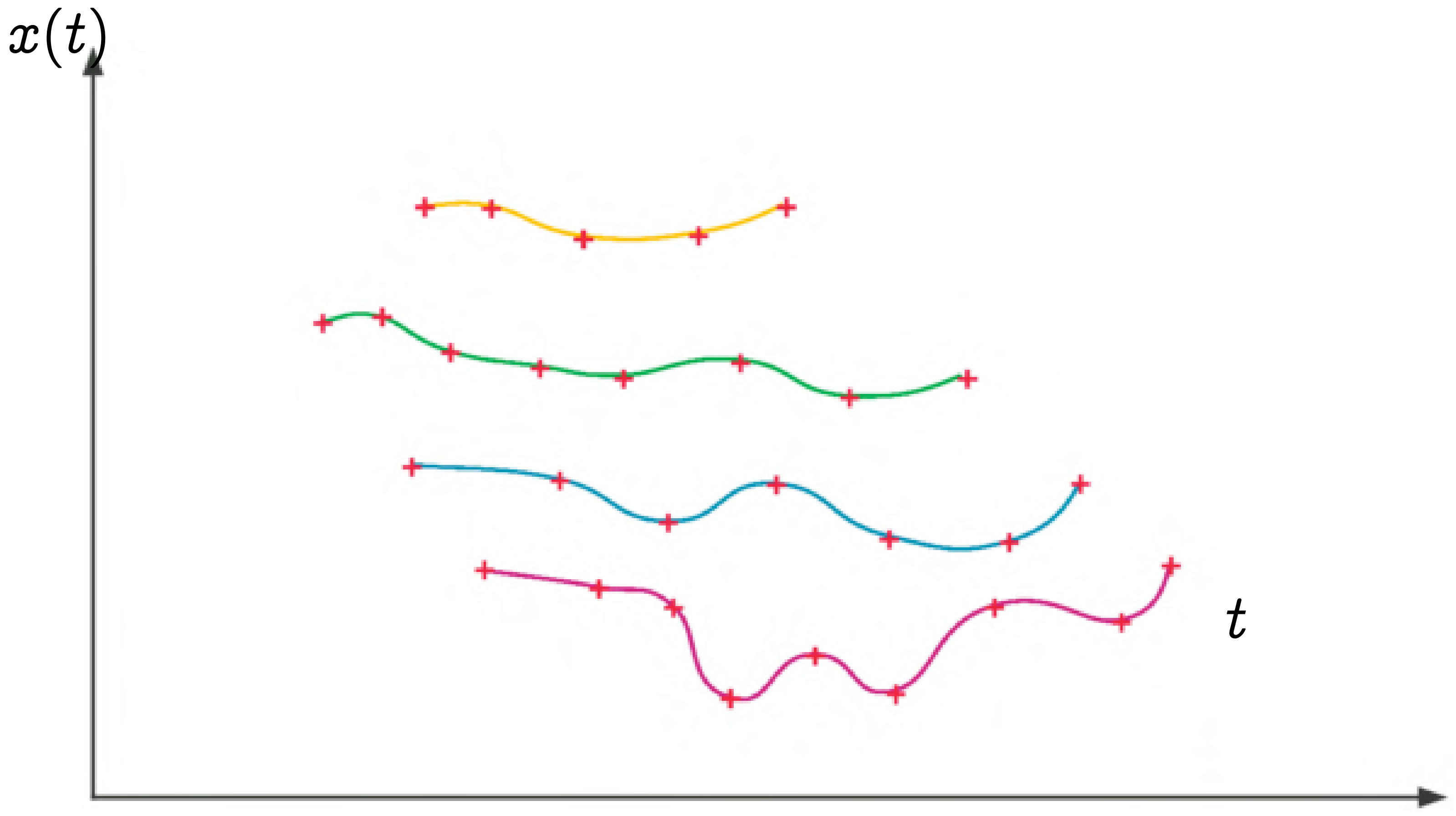
12. Iterator

Characterization of the iterator

- structural (by induction on the program syntax)
- flow sensitive (the execution order of statements is taken into account)
- path sensitive (distinguishes between feasible paths through a program)
- context sensitive (function calls are analyzed differently for each call site)
- interprocedural (function bodies are analyzed in the context of each respective call site)

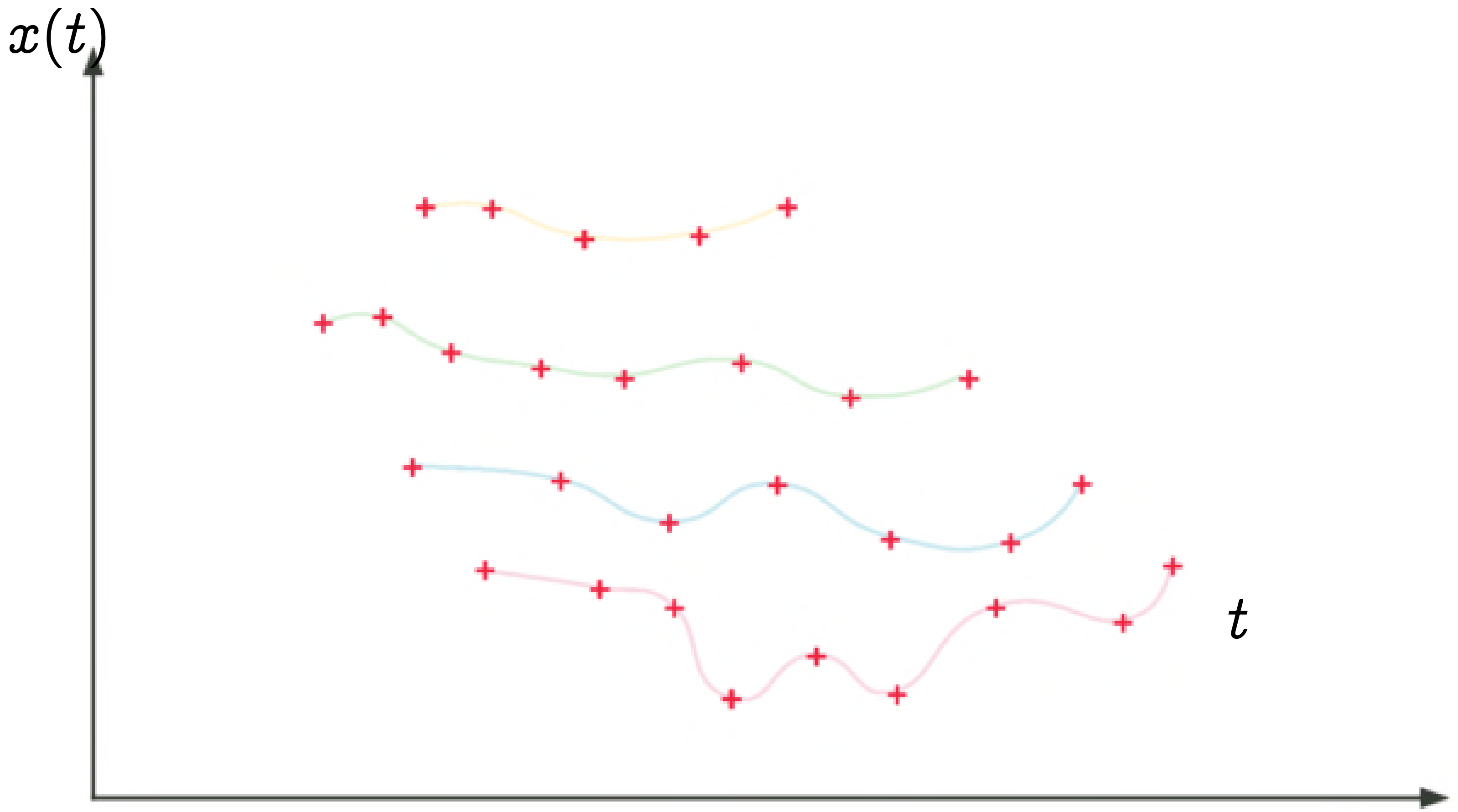
13. General Abstract Domains

Semantics



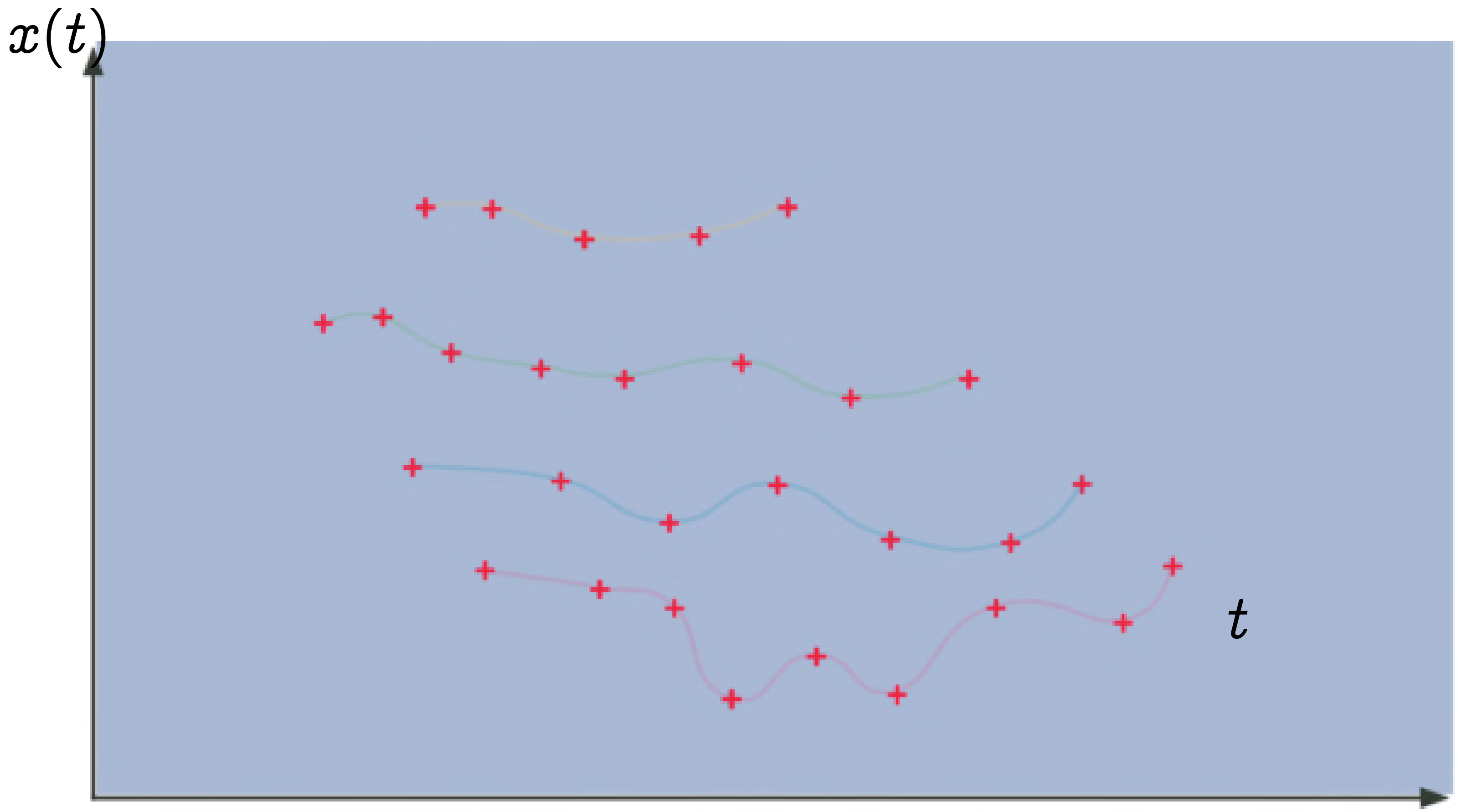
(Infinite) set of traces (finite ou infinite)

Abstraction to a set of states (invariant)



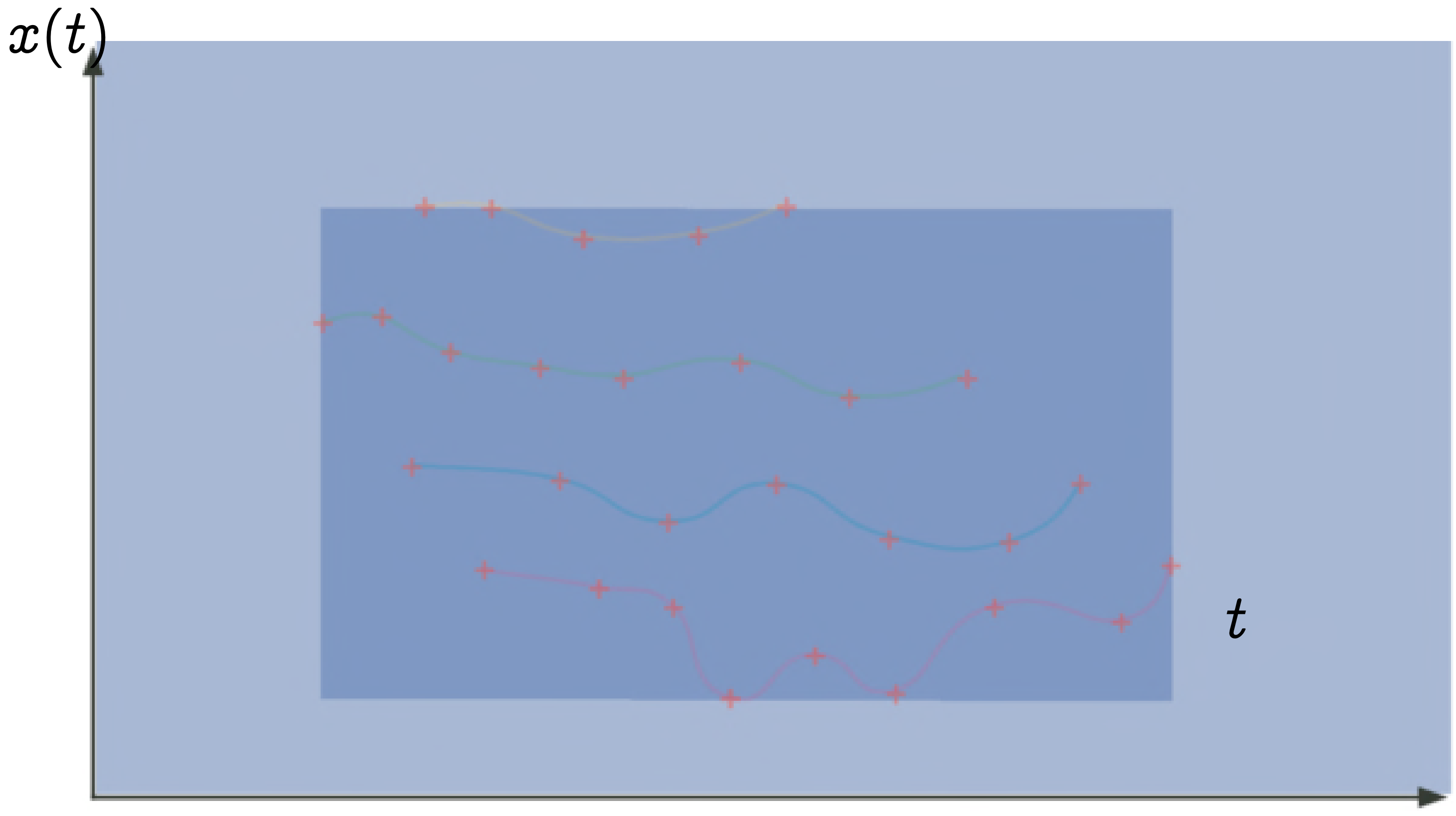
Set of points $\{(x_i, y_i) : i \in \Delta\}$, Floyd/Hoare/Naur invariance proof method [Cou02]

Abstraction by signs



Signs $x \geq 0$, $y \geq 0$ [CC79]

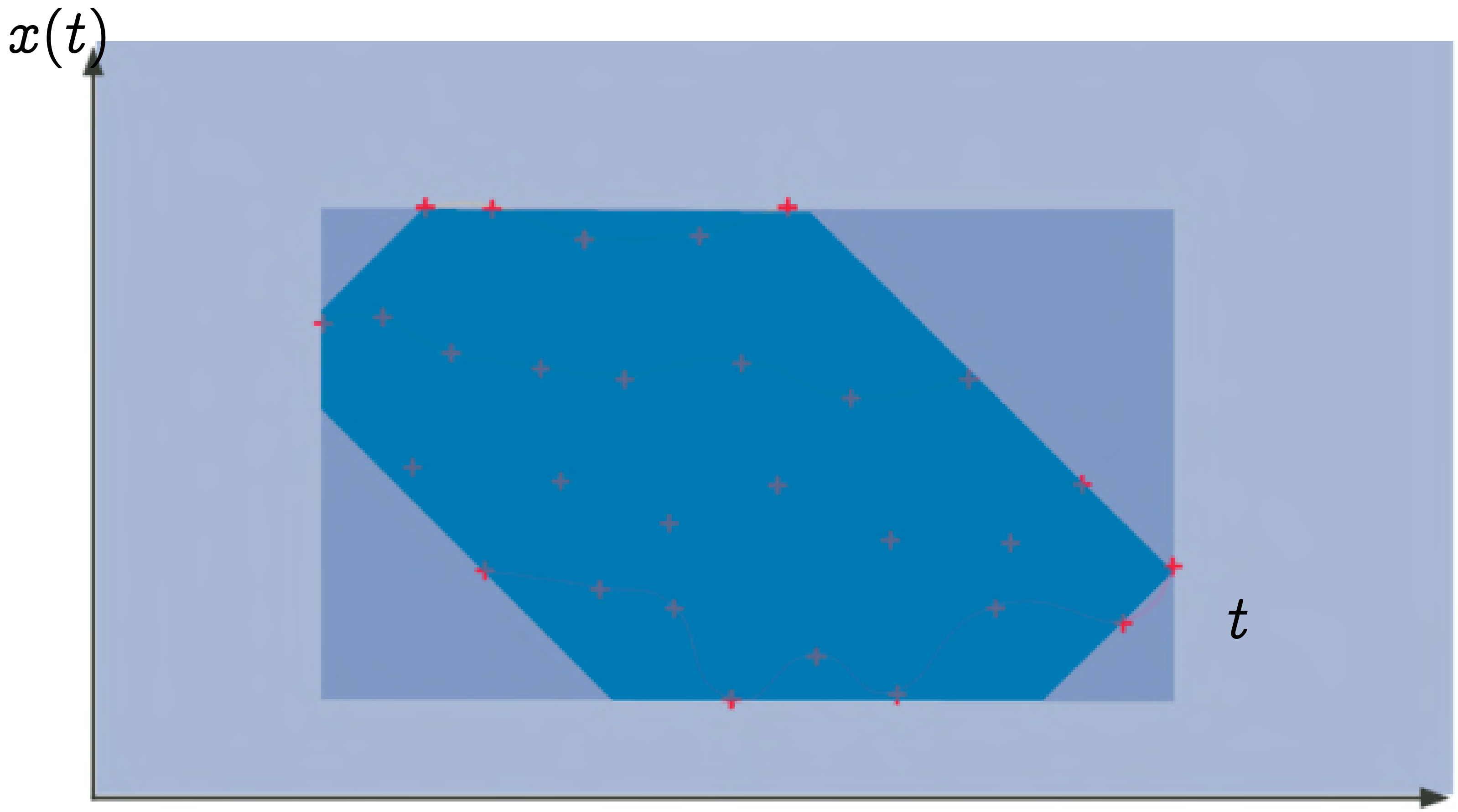
Abstraction by intervals



Intervals $a \leq x \leq b$, $c \leq y \leq d$ [CC77]

Sound implementation with floats!

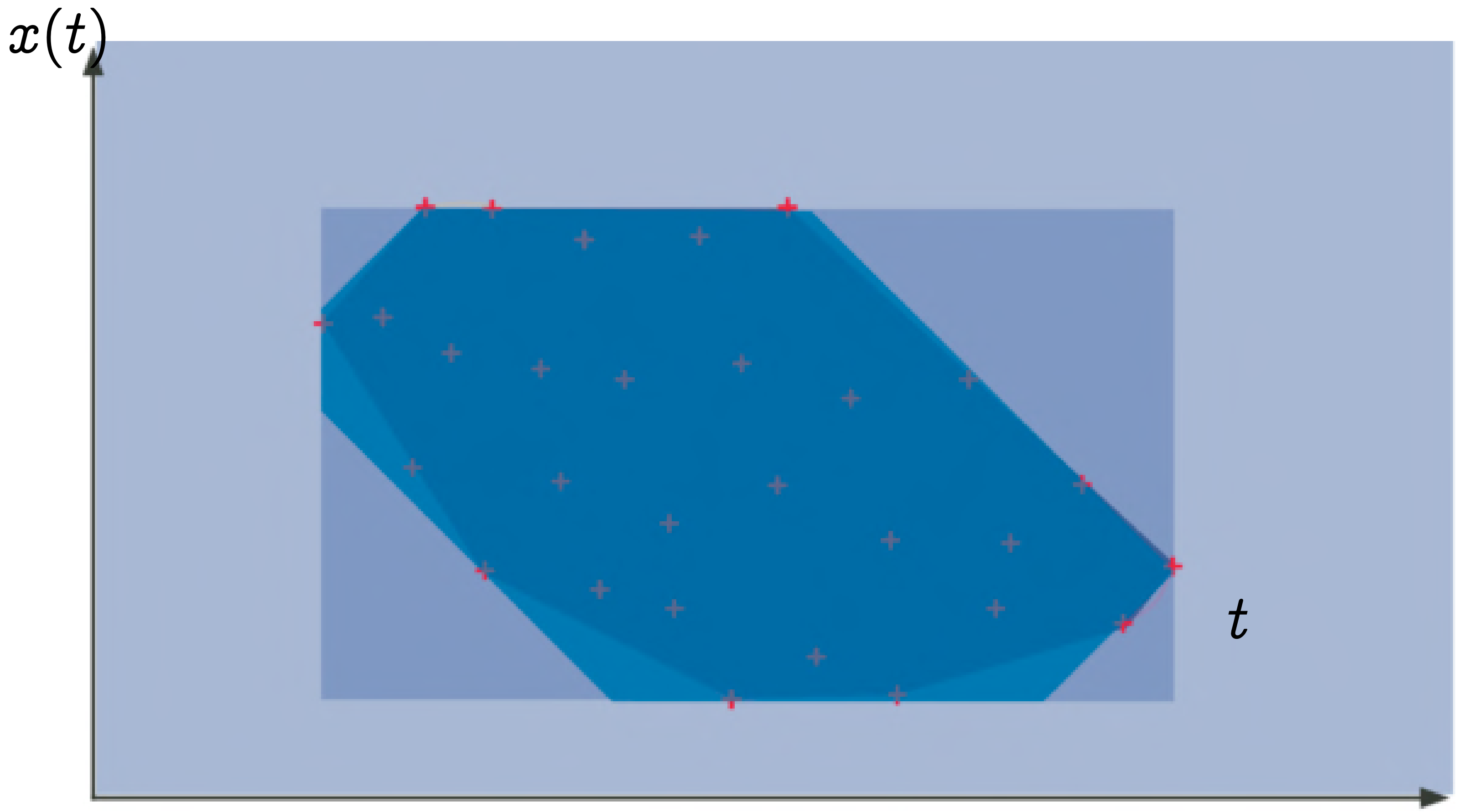
Abstraction by octagons



Octagons $x - y \leq a,\ x + y \leq b$ [Min06]

Sound implementation with floats!

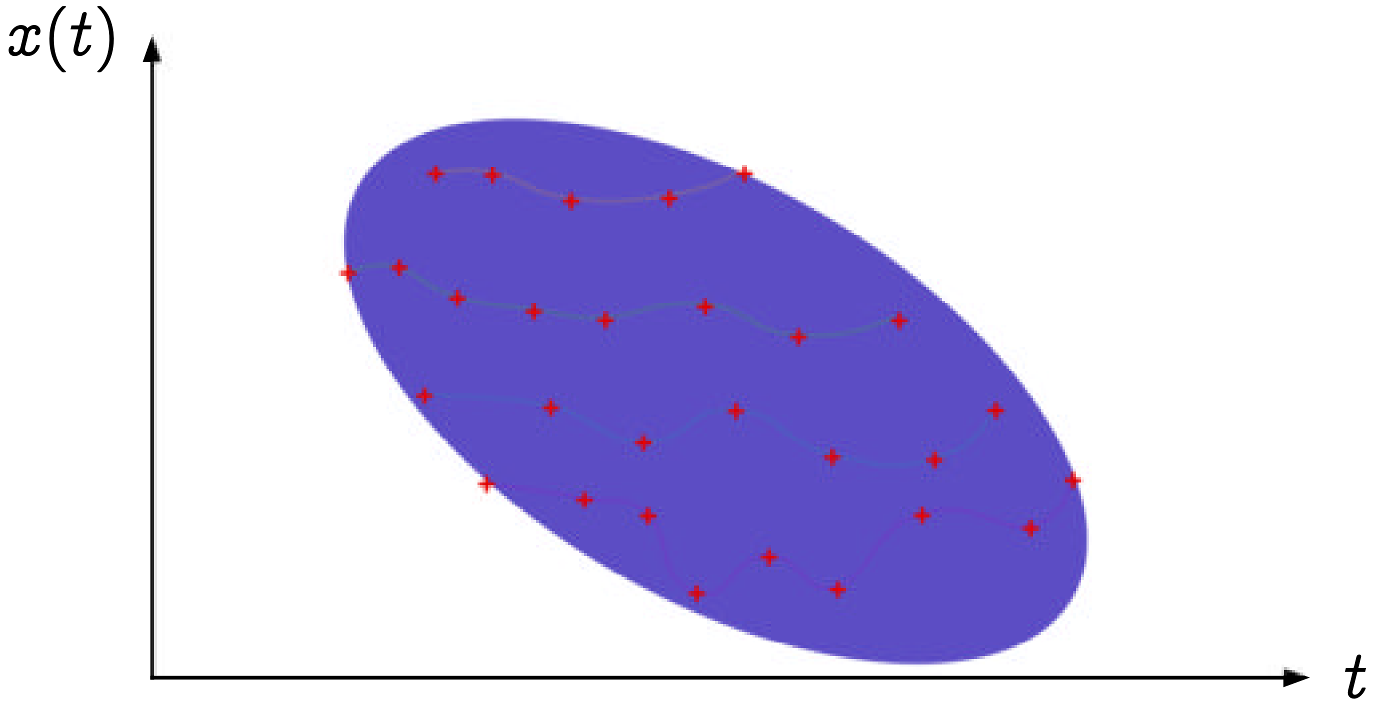
Abstraction by polyedra



Polyedra $a.x + b.y \leq c$ [CH78]

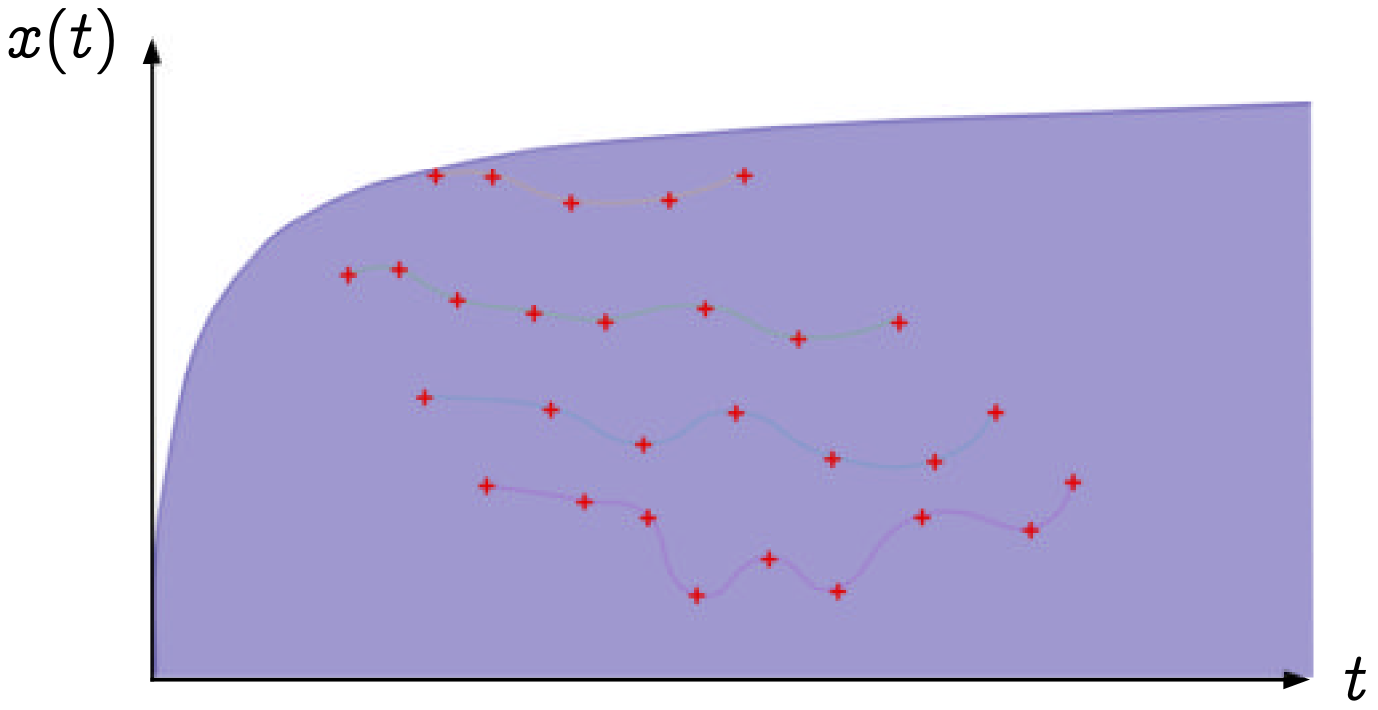
NEW Sound implementation with floats! [CMC08]

Abstraction by ellipsoid for filters



Ellipsoids $(x-a)^2+(y-b)^2 \leq c$ [Fer05b]

Abstraction by exponentials



Exponentials $a^x \leq y$ [Fer05a]

Abstraction by floating-point linearization [Min04a, Min04b]

– Approximate arbitrary expressions in the form
  $[a_0, b_0] + \sum_k([a_k, b_k] \times V_k)$
– Example:
  `Z = X - (0.25 * X)` is linearized as
  $\mathtt{Z} = ([0.749\cdots, 0.750\cdots] \times \mathtt{X}) + (2.35\cdots 10^{-38} \times [-1, 1])$
– Allows simplification even in the interval domain
  if `X` $\in$ [-1,1], we get $|\mathtt{Z}| \leq 0.750\cdots$ instead of $|\mathtt{Z}| \leq 1.25\cdots$
– Allows using a relational abstract domain (octagons)
– Example of good compromize between cost and precision

14. Trace partitioning

Paths versus reachable states analysis

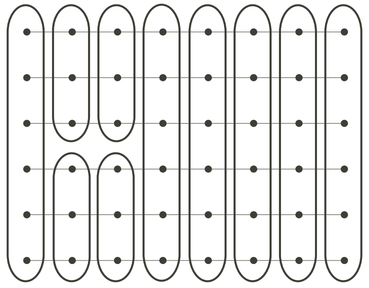
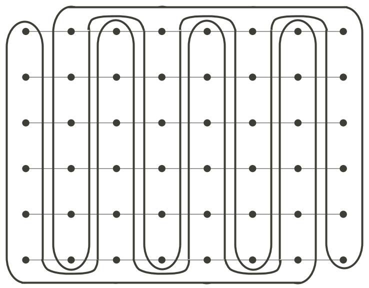
- The merge over all paths analysis is more precise than fixpoint reachable states analysis for non-distributive abstract domains (but more costly)
- The merge over all paths can be obtained in fixpoint form by disjunctive completion of the abstract domain [CC79]
- The disjunctive completion is costly (a terminating analysis such as constant propagation can become non terminating)

Reference

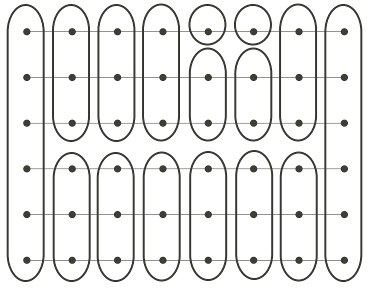
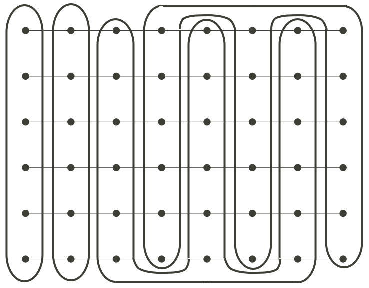
[CC79] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In 6th *POPL*, pages 269–282, San Antonio, TX, 1979. ACM Press.

Trace partitioning

– State partitionning by program points [9]

– Trace partitionning [10]

– Trace partitionning abstract interpretation combines the effects of case analysis and symbolic execution [MR05, RM07]

[9] all reachable states corresponding to a given program point are over-approximated by a local invariant on memory states reachable at that program points

[10] portions of traces starting at a given program point for given memory values and finishing at a given program point are analyzed by an overapproximating abstract execution

Example of trace partitioning

Principle:

– Semantic equivalence:

```
   if (B) { C1 } else { C2 }; C3
                  ⇓
if (B) { C1; C3 } else { C2; C3 };
```

– More precise in the abstract: concrete execution paths are merged later.

Application:

```
if (B)
  { X=0; Y=1; }
else
  { X=1; Y=0; }
R = 1 / (X-Y);
```

cannot result in a division by zero

Scalability of trace partitioning

```
% cat -n explode.c
     1 void main() {
     2    /* uninitialized local variables */
     3    unsigned int a1, a2, a3, a4, a5, a6; int r;
     4    while(a1<20) { a1++; }
     5    while(a2<20) { a2++; }
     6    while(a3<20) { a3++; }
     7    while(a4<20) { a4++; }
     8    while(a5<20) { a5++; }
     9 }
% (time astree -exec-fn main -partition-all explode.c) |& egrep
  "error:|WARN|pf\+"
*** error: can't allocate region
Fatal error: out of memory.
5072.204u 44.410s 1:29:31.01 95.2% 0+0k 0+0io 45661pf+0w
%
```

Trace partitionning must be performed locally (thanks to *analysis directives*)

– CBMC cannot make it either:

```
% time ./cbmc explode.c
...
Unwinding loop 1 iteration 1
Unwinding loop 1 iteration 2
...
Unwinding loop 1 iteration 8731
...
Abort
```

– But ASTRÉE succeeds (without partitionning `-partition-all`)

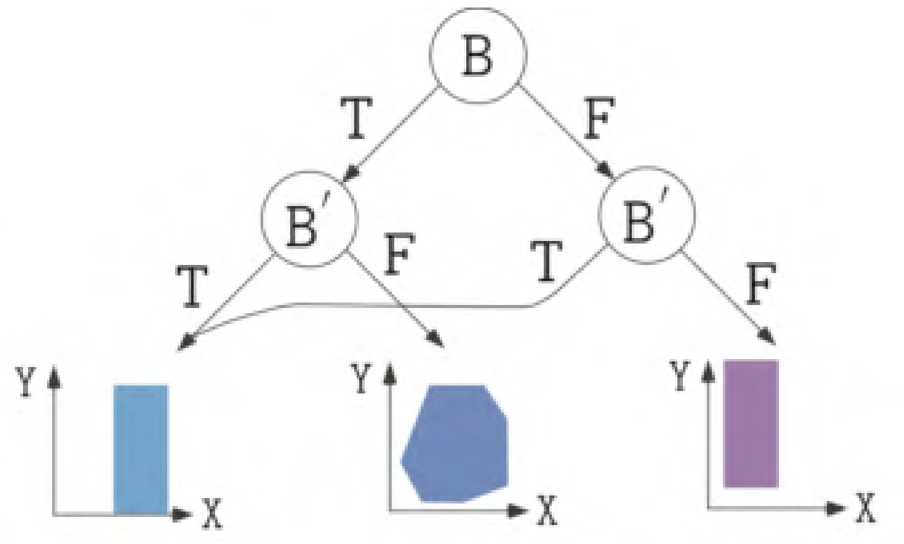
```
% (time astree -exec-fn main explode.c) |& egrep "error:|WARN|pf"
0.402u 0.064s 0:00.48 95.8% 0+0k 0+0io 0pf+0w
%
```

15. Abstract Domain Functors

Example of abstract domain functor in ASTRÉE: decision trees

– 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

16. Combination of Abstract Domains by Reduction

Example: typical combination of abstractions in ASTRÉE

```
/* Launching the forward abstract interpreter */
/* Domains:  Guard domain, and Boolean packs (based on Absolute
value equality relations, and Symbolic constant propagation
(max_depth=20), and Linearization, and Integer intervals, and
congruences, and bitfields, and finite integer sets, and Float
intervals), and Octagons, and High_passband_domain(10), and
Second_order_filter_domain (with real roots)(10), and
Second_order_filter_domain (with complex roots)(10), and
Arithmetico-geometric series, and new clock, and Dependencies
(static), and Equality relations, and Modulo relations, and
Symbolic constant propagation (max_depth=20), and Linearization,
and Integer intervals, and congruences, and bitfields, and
finite integer sets, and Float intervals.  */
```

Reduction [CC79, CCF⁺08]

Example: reduction of intervals [CC76] by simple congruences [Gra89]

```
% cat -n congruence.c
     1 /* congruence.c */
     2 int main()
     3 { int X;
     4   X = 0;
     5   while (X <= 128)
     6     { X = X + 4; };
     7   __ASTREE_log_vars((X));
     8 }
% astree congruence.c -no-relational -exec-fn main |& egrep "(WARN)|(X in)"
direct = <integers (intv+cong+bitfield+set): X in {132} >
```

Intervals : $X \in [129, 132]$ + congruences : $X = 0 \mod 4 \Longrightarrow X \in \{132\}$.

17. Refinement by Parametrization

Parameterized abstractions

– Parameterize the cost / precision ratio of abstractions in the static analyzer

– Examples:

  - array smashing: `--smash-threshold` n (400 by default) $\rightarrow$ smash elements of arrays of size $> n$, otherwise individualize array elements (each handled as a simple variable).

  - packing in octogons: (to determine which groups of variables are related by octagons and where)
    - `--fewer-oct`: no packs at the function level,
    - `--max-array-size-in-octagons` n: unsmashed array elements of size $> n$ don't go to octagons packs

Parameterized widenings

– Parameterize the rate and level of precision of widenings in the static analyzer

– Examples:

  - delayed widenings: `--forced-union-iterations-at-beginning` n (2 by default)
  - enforced widenings: `--forced-widening-iterations-after` n (250 by default)
  - thresholds for widening (e.g. for integers):

```
let widening_sequence =
 [ of_int 0; of_int 1; of_int 2; of_int 3; of_int 4; of_int 5;
   of_int 32767; of_int 32768; of_int 65535; of_int 65536;
   of_string "2147483647"; of_string "2147483648";
   of_string "4294967295" ]
```

18. Refinement by Analysis Directives

Analysis directives

– Require a local refinement of an abstract domain

– Example:

```
% cat repeat1.c
typedef enum {FALSE=0,TRUE=1} BOOL;
int main () {
  int x = 100; BOOL b = TRUE;

  while (b) {
    x = x - 1;
    b = (x > 0);
  }
}
% astree -exec-fn main repeat1.c |& egrep "WARN"
repeat1.c:5.8-13::[call#main@2:loop@4>=4:]: WARN: signed int arithmetic
range [-2147483649, 2147483646] not included in [-2147483648, 2147483647]
%
```

Example of directive (Cont'd)

```
% cat repeat2.c
typedef enum {FALSE=0,TRUE=1} BOOL;
int main () {
  int x = 100; BOOL b = TRUE;
 __ASTREE_boolean_pack((b,x));
  while (b) {
    x = x - 1;
    b = (x > 0);
  }
}
```

```
% astree -exec-fn main repeat2.c |& egrep "WARN"
%
```

The insertion of this directive could have been automated in ASTRÉE (if the considered family of programs had had "repeat" loops).

Automatic analysis directives

– The directives can be inserted automatically by static analysis

– Example:

```
% cat p.c
int clip(int x, int max, int min) {
 if (max >= min) {
  if (x <= max) {
   max = x;
  }
  if (x < min) {
   max = min;
  }
 }
 return max;
}
void main() {
 int m = 0; int M = 512; int x, y;
 y = clip(x, M, m);
  __ASTREE_assert(((m<=y) && (y<=M)));
}
% astree -exec-fn main p.c |& grep WARN
%
```

```
% astree -exec-fn main p.c -dump-partition
...
int (clip)(int x, int max, int min)
{
  if ((max >= min))
  {  __ASTREE_partition_control((0))
    if ((x <= max))
    {
      max = x;
    }
    if ((x < min))
    {
      max = min;
    }
    __ASTREE_partition_merge_last(());
  }
  return max;
}
...
%
```

19. Inexpressiveness

Inexpressivity

– The weakest invariant to prove the specification may be inexpressible with the current reduced abstractions, whatever parameters or analysis directives are used [(11)]

$\Rightarrow$ false alarms are unavoidable and cannot be solved

– No solution, but refining the current abstract domains!

– Done by extension of the abstract interpreter with a new abstract domain

(11) or their cost might me prohibitive like in exhaustive partitioning per data value!

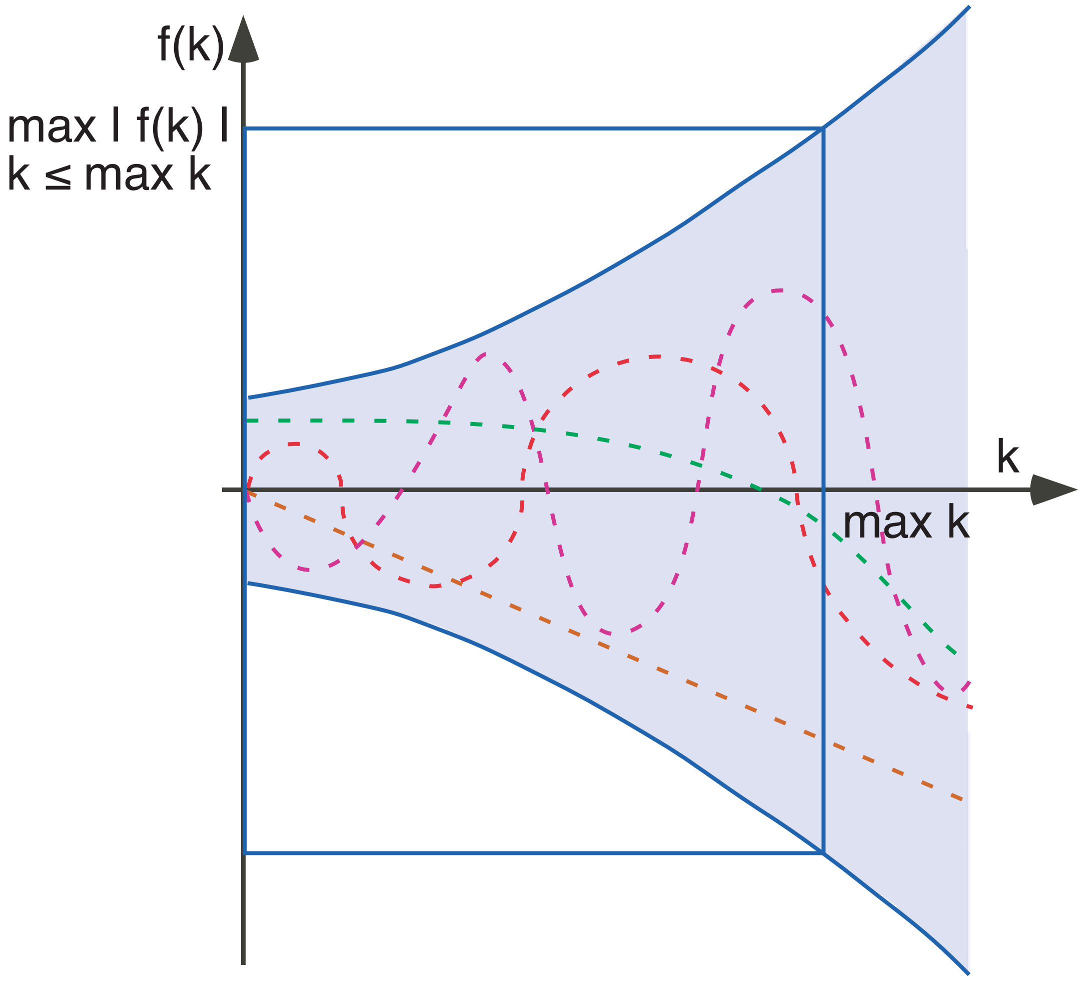
Adding new abstract domains

- Design the mathematical abstract domain
- Specify the concretization, and
- Implement:
  - the representation of the (parameterized) abstract properties
  - the abstract property transformers for language primitives
  - (parameterized) widening
  - reduction with other abstractions
- Examples : ellipsoids for filters [Fer05b], exponentials for accumulation of small rounding errors [Fer05a], quaternions, ...

20. Refinement by Extension

Example of abstract domain introduced in Astrée

Overapproximation with an arithmetico-geometric series:



Arithmetico-geometric series (12) [Fer05a]

– Abstract domain: $(R^{+})^{5}$

– Concretization:

$$\gamma \in (R^{+})^{5} \longmapsto \wp(N \mapsto R)$$

$$\gamma(M, a, b, a', b') =$$

$$\{f \mid \forall k \in N : |f(k)| \leq \left(\boldsymbol{\lambda} x \cdot ax + b \circ (\boldsymbol{\lambda} x \cdot a'x + b')^{k}\right)(M)\}$$

i.e. any function bounded by the arithmetic-geometric progression.

Reference

[4] J. Feret. The arithmetic-geometric progression abstract domain. In *VMCAI'05*, Paris, LNCS 3385, pp. 42–58, Springer, 2005.

(12) here in R but must be implemented in the floats by appropriate roundings!

Arithmetic-Geometric Progressions: Example 1

```
% 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
```

Arithmetic-geometric progressions (Example 2)

```
% 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; }                ← potential overflow!
    else { R = 0; }
    T = (R >= 100);
    __ASTREE_wait_for_clock(());
  }}
% 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.
```

More precise than the *clock domain* (intervals for X, $X + \texttt{clock}$, $X - \texttt{clock}$) which could therefore be suppressed!

21. Industrial applications of abstract interpretation

Industrial results obtained with ASTRÉE

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

– Software 1 : 132.000 lignes de C, 40mn sur un PC 2.8 GHz, 300 mégaoctets (nov. 2003)

– Software 2 : 1.000.000 de lignes de C, 34h, 8 gigaoctets (nov. 2005)

No false alarm

World premières !

22. Conclusion

Conclusion

- Static analysis by abstract interpretation does scale up for domain-specific industrial software
- In consequence, software engineering methodology should evolve in the near future:
  - From the present-day process-based methodology controlling the design, coding and testing processes
  - To a product-based methodology incorporating a systematic control of the final software product by static analyzers.

THE END

Thank you for your attention

23. Bibliography

Short bibliography

[AGM93] G. Amato, F. Giannotti, and G. Mainetto. Data sharing analysis for a database programming language via abstract interpretation. In R. Agrawal, S. Baker, and D.A.Bell, editors, *Proc. 19th Int. Conf. on Very Large Data Bases*, pages 405–415, Dublin, IE, 24–27 Aug. 1993. MORGANKAUFMANN.

[BCC+03] 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. In *Proc. ACM SIGPLAN '2003 Conf. PLDI*, pages 196–207, San Diego, CA, US, 7–14 June 2003. ACM Press.

[BPC01] J. Bailey, A. Poulovassilis, and C. Courtenage. Optimising active database rules by partial evaluation and abstract interpretation. In *Proc. 8th Int. Work. on Database Programming Languages*, LNCS 2397, pages 300–317, Frascati, IT, 8–10 Sep. 2001. Springer.

[BS97] V. Benzaken and X. Schaefer. Static integrity constraint management in object-oriented database programming languages via predicate transformers. In M. Aksit and S. Matsuoka, editors, *Proc. 11th European Conf. on Object-Oriented Programming, ECOOP '97*, LNCS 1241. Springer, Jyväskylä, FI, 9–13 June 1997.

[CC76] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proc. 2nd Int. Symp. on Programming*, pages 106–130, Paris, FR, 1976. Dunod.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th POPL*, pages 238–252, Los Angeles, CA, 1977. ACM Press.

[CC79] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *6th POPL*, pages 269–282, San Antonio, TX, 1979. ACM Press.

[CC92a] P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation, invited paper. In M. Bruynooghe and M. Wirsing, editors, *Proc. 4th Int. Symp. on PLILP '92*, Leuven, BE, 26–28 Aug. 1992, LNCS 631, pages 269–295. Springer, 1992.

[CC92b] P. Cousot and R. Cousot. Inductive definitions, semantics and abstract interpretation. In *19th POPL*, pages 83–94, Albuquerque, NM, US, 1992. ACM Press.

[CC95] P. Cousot and R. Cousot. Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In *Proc. 7th FPCA*, pages 170–181, La Jolla, CA, US, 25–28 June 1995. ACM Press.

[CC00] P. Cousot and R. Cousot. Temporal abstract interpretation. In *27th POPL*, pages 12–25, Boston, MA, US, Jan. 2000. ACM Press.

[CC02] P. Cousot and R. Cousot. Systematic design of program transformation frameworks by abstract interpretation. In *29th POPL*, pages 178–190, Portland, OR, US, Jan. 2002. ACM Press.

[CC03] P. Cousot and R. Cousot. Parsing as abstract interpretation of grammar semantics. *Theoret. Comput. Sci.*, 290(1):531–544, Jan. 2003.

[CC04] P. Cousot and R. Cousot. An abstract interpretation-based framework for software watermarking. In *31st POPL*, pages 173–185, Venice, IT, 14–16 Jan. 2004. ACM Press.

[CCF$^+$07] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Varieties of static analyzers: A comparison with ASTRÉE, invited paper. In M. Hinchey, He Jifeng, and J. Sanders, editors, *Proc. 1st TASE '07*, pages 3–17, Shanghai, CN, 6–8 June 2007. IEEE Comp. Soc. Press.

[CCF$^+$08] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Combination of abstractions in the ASTRÉE static analyzer. In M. Okada and I. Satoh, editors, *11th ASIAN 06*, pages 272–300, Tokyo, JP, 6–8 Dec. 2006, 2008. LNCS 4435, Springer.

[CGJ$^+$00] E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In E.A. Emerson and A.P. Sistla, editors, *Proc. 12th Int. Conf. CAV '00*, Chicago, IL, US, LNCS 1855, pages 154–169. Springer, 15–19 Jul. 2000.

[CGR07] P. Cousot, P. Ganty, and J.-F. Raskin. Fixpoint-guided abstraction refinements. In G. Filé and H. Riis-Nielson, editors, *Proc. 14th Int. Symp. SAS '07*, Kongens Lyngby, DK, LNCS 4634, pages 333–348. Springer, 22–24 Aug. 2007.

[CH78] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *5th POPL*, pages 84–97, Tucson, AZ, 1978. ACM Press.

[CMC08] L. Chen, A. Miné, and P. Cousot. A sound floating-point polyhedra abstract domain. To appear in The Sixth ASIAN SYMP on Programming Languages and Systems, APLAS 2008, Bangalore, India, 9–11 December, 2008, 2008.

[Cou97] P. Cousot. Types as abstract interpretations, invited paper. In *24th POPL*, pages 316–331, Paris, FR, Jan. 1997. ACM Press.

[Cou00] P. Cousot. Partial completeness of abstract fixpoint checking, invited paper. In B.Y. Choueiry and T. Walsh, editors, *Proc. 4th Int. Symp. SARA '2000*, Horseshoe Bay, TX, US, LNAI 1864, pages 1–25. Springer, 26–29 Jul. 2000.

[Cou02] P. Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Theoret. Comput. Sci.*, 277(1—2):47–103, 2002.

[Cou03] P. Cousot. Verification by abstract interpretation, invited chapter. In N. Dershowitz, editor, *Proc. Int. Symp. on Verification – Theory & Practice – Honoring Zohar Manna's 64th Birthday*, pages 243–268. LNCS 2772, Springer, Taormina, IT, 29 June – 4 Jul. 2003.

[Dan07] V. Danos. Abstract views on biological signaling. In *Mathematical Foundations of Programming Semantics, 23rd Annual Conf. (MFPS XXIII)*, 2007.

[DS07] D. Delmas and J. Souyris. ASTRÉE: from research to industry. In G. Filé and H. Riis-Nielson, editors, *Proc. 14th Int. Symp. SAS '07*, Kongens Lyngby, DK, LNCS 4634, pages 437–451. Springer, 22–24 Aug. 2007.

[Fer04] J. Feret. Static analysis of digital filters. In D. Schmidt, editor, *Proc. 30th ESOP '2004, Barcelona, ES*, volume 2986 of *LNCS*, pages 33–48. Springer, Mar. 27 – Apr. 4, 2004.

[Fer05a] J. Feret. The arithmetic-geometric progression abstract domain. In R. Cousot, editor, *Proc. 6th Int. Conf. VMCAI 2005*, pages 42–58, Paris, FR, 17–19 Jan. 2005. LNCS 3385, Springer.

[Fer05b] J. Feret. Numerical abstract domains for digital filters. In *1st Int. Work. on Numerical & Symbolic Abstract Domains, NSAD "05*, Maison Des Polytechniciens, Paris, FR, 21 Jan. 2005.

[GM04] R. Giacobazzi and I. Mastroeni. Abstract non-interference: Parameterizing non-interference by abstract interpretation. In *31st POPL*, pages 186–197, Venice, IT, 2004. ACM Press.

[Gra89] P. Granger. Static analysis of arithmetical congruences. *Int. J. Comput. Math.*, 30:165–190, 1989.

[GRS00] R. Giacobazzi, F. Ranzato, and F. Scozzari. Making abstract interpretations complete. *J. ACM*, 47(2):361–416, 2000.

[JP06] Ph. Jorrand and S. Perdrix. Towards a quantum calculus. In *Proc. 4th Int. Work. on Quantum Programming Languages, ENTCS*, 2006.

[Min04a] A. Miné. Relational abstract domains for the detection of floating-point run-time errors. In D. Schmidt, editor, *Proc. 30th ESOP '2004, Barcelona, ES*, volume 2986 of *LNCS*, pages 3–17. Springer, Mar. 27 – Apr. 4, 2004.

[Min04b] A. Miné. *Weakly Relational Numerical Abstract Domains*. Thèse de doctorat en informatique, École polytechnique, Palaiseau, FR, 6 Dec. 2004.

[Min06] A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19:31–100, 2006.

[Mon08] D. Monniaux. The pitfalls of verifying floating-point computations. *TOPLAS*, 30(3):Article No. 12, may 2008.

[MR05] L. Mauborgne and X. Rival. Trace partitioning in abstract interpretation based static analyzer. In M. Sagiv, editor, *Proc. 14th ESOP '2005, Edinburg, UK*, volume 3444 of *LNCS*, pages 5–20. Springer, Apr. 2Ñ̃-10, 2005.

[PCJD07] M. Dalla Preda, M. Christodorescu, S. Jha, and S. Debray. Semantics-based approach to malware detection. In *34th POPL*, pages 238–252, Nice, France, 17–19 Jan. 2007. ACM Press.

[Per06] S. Perdrix. *Modèles formels du calcul quantique : ressources, machines abstraites et calcul par mesure*. PhD thesis, Institut National Polytechnique de Grenoble, Laboratoire Leibniz, 2006.

[RM07] X. Rival and L. Mauborgne. The trace partitioning abstract domain. *TOPLAS*, 29(5), Aug. 2007.

[RT04] F. Ranzato and F. Tapparo. Strong preservation as completeness in abstract interpretation. In D. Schmidt, editor, *Proc. 30th ESOP '04*, volume 2986 of *LNCS*, pages 18–32, Barcelona, ES, Mar. 29 – Apr. 2 2004. Springer.

[RT06] F. Ranzato and F. Tapparo. Strong preservation of temporal fixpoint-based operators by abstract interpretation. In A.E. Emerson and K.S. Namjoshi, editors, *Proc. 7th Int. Conf. VMCAI 2006*, pages 332–347, Charleston, SC, US, 8–10 Jan. 2006. LNCS 3855 , Springer.