« Verification of Large Complex Software by Abstract Interpretation »

Patrick Cousot
Computer Science Department
École normale supérieure
45 rue d'Ulm, 75230 Paris cedex 05, France

Patrick.Cousot@ens.fr
www.di.ens.fr/~cousot

11th Annual Asian Computing Science Conference — National Center of Sciences, Tokyo, Japan — 6–8 Dec. 2006

Content

Abstract

Since almost any large complex software has bugs which are not found by test methods, researchers have developed program correctness proof methods which have been successful in the small. This consists in defining a semantics formally describing the executions of a program and then in proving a theorem stating that these executions have a given property (for example that an expected result is provided in a finite time). Fundamental mathematical undecidability results show that these proofs cannot be done automatically by computers.

Confronted with this fundamental difficulty, abstract interpretation proceeds by correct approximation of the semantics. If the approximation is sound, no potential error can ever be overlooked, a basic requirement of formal verification methods. If the approximation is coarse enough, it is computable. If it is precise enough, it yields a correctness proof. The goal is therefore to find cheap approximations (so as to scale up in the large) which are precise enough (to avoid false alarms where a property does hold but this cannot be proved because of an approximation which is too imprecise).

We will introduce a few elements of abstract interpretation and explain how to formalize the abstraction of semantic properties so as to obtain computable approximations leading to effective algorithms for the static analysis of the possible behaviors of programs.

Finally, we will describe an example of application of the theory to the proof of absence of runtime errors on synchronous control/command and underly the difficulties (such as floating point computations). This approach was applied with success to the verification of the electric flight control of the commercial planes.

The importance of software

Software is hidden everywhere

Software is massively present in all mission-critical and safety-critical industrial infrastructures

Accident analysis (metro)

– Paris métro line 12 accident [1]: the driver was going too fast

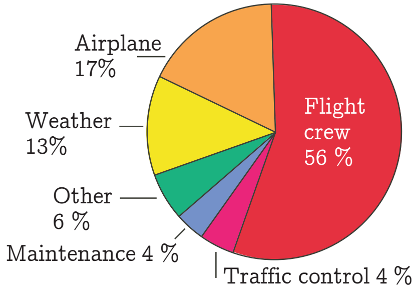
– Roma metro line A accident [2]: the driver went on at a red signal

[1] On August 30th, 2000, at the Notre-Dame-de-Lorette métro station in Paris, a car flipped over on its side and slid to a stop just a few feet from a train stopped on the opposite platform (24 injured).

[2] On October 17th, 2006, a speeding subway train rammed into another train halted at the Vittorio Emanuele station in central Rome (1 dead, 60 injured). The driver might have misunderstood the control centre authorizing the train to proceed to the "next station" (Manzoni, closed to the public) while the driver would have understood it to mean the "next working station" (Vittorio Emanuele, after Manzoni), *La Repubblica*, Oct. 20th, 2006.

Accident analysis (avionics)

Primary cause of major commercial jet accidents world-wide as determined by the investigation authorities between 1995 and 2004 [3] [1]



Reference

[1] D. Michaels & A. Pasztor. *Incidents Prompt New Scrutiny Of Airplane Software Giltches* citing a Boeing source. Wall Street Journal, Vol. CCXLVII, No 125, 30 mai 2006.

[3] Includes only accidents with known causes.

Software replaces human operators

– Computer control is the cheapest and safest solution to avoid such accidents

– New high-speed métro line 14 (Météor): fully automated, no operators

– Modern commercial airplanes: massive automation of control/ commands, piloting, communications, collision avoidance, etc

Why is software erroneous?

(1) Software is huge

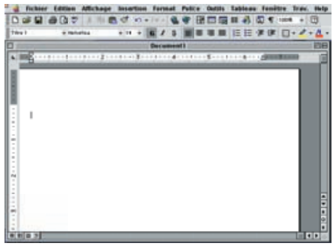
As computer hardware capacity grows...

ENIAC
5,000 flops [4]

NEC Earth Simulator
35×10^{12} flops [5]

4 Floating point operations per second
5 10^{12} = Thousand Billion

Software size grows...

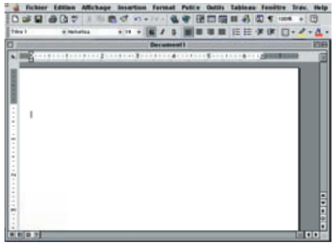


Text editor
1,700,000 lines of C [6]

Operating system
35,000,000 lines of C [7]

6 3 months for full-time reading of the code
7 5 years for full-time reading of the code

. . . and so does the number of bugs



Text editor
1,700,000 lines of C[6]
1,700 bugs (estimation)

Operating system
35,000,000 lines of C[7]
30,000 known bugs

[6] 3 months for full-time reading of the code
[7] 5 years for full-time reading of the code

Computers are finite

- Scientists reason on continuous, infinite mathematical structures (e.g. $\mathbb{R}$)
- Computers can only handle discrete, finite structures

(2) Computers are finite

Overflows

- Numbers are encoded onto a limited number of bits (*binary digits*)
- Some operations may overflow (e.g. integers: 32 bits $\times$ 32 bits $=$ 64 bits)
- Using different number sizes (32, 64, . . . bits) can also be the source of overflows

The Ariane 5.01 maiden flight

– June 4^{th}, 1996 was the maiden flight of Ariane 5

(3) Computers go round

The Ariane 5.01 maiden flight failure

– June 4^{th}, 1996 was the maiden flight of Ariane 5

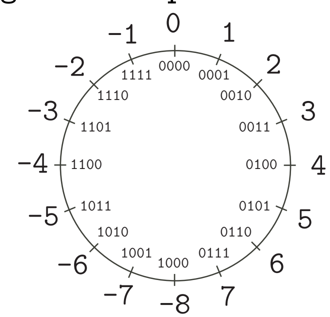
– The launcher was destroyed after 40 seconds of flight because of a software overflow[8]

[8] A 16 bit piece of code of Ariane 4 had been reused within the new 32 bit code for Ariane 5. This caused an uncaught overflow, making the launcher uncontrolable.

Modular arithmetic. . .

– Todays, computers avoid integer overflows thanks to modular arithmetic

– Example: integer 2's complement encoding on 8 bits



. . . can be contrary to common sense

```
# 1073741823 + 1;;
- : int = -1073741824
# -1073741824 - 1;;
- : int = 1073741823
# -1073741824 ÷ -1;;
- : int = -1073741824
```

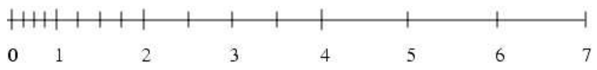

(4) Computers do round

Floats: mapping many to few

– Reals are mapped to floats (floating-point arithmetic)

$$\pm d_0.d_1 d_2 \ldots d_{p-1} \beta^e$$ [9]

– For example on 6 bits (with $p = 3$, $\beta = 2$, $e_{\min} = -1$, $e_{\max} = 2$), there are 32 normalized floating-point numbers. The 16 positive numbers are



[9] where - $d_0 \neq 0$,
- p is the number of significative digits,
- β is the basis (2), and
- e is the exponant ($e_{\min} \leq e \leq e_{\max}$)

Rounding

– Computations returning reals that are not floats, must be rounded

– Most mathematical identities on $\mathbb{R}$ are no longer valid with floats

– Rounding errors may either compensate or accumulate in long computations

– Computations converging in the reals may diverge with floats (and ultimately overflow)

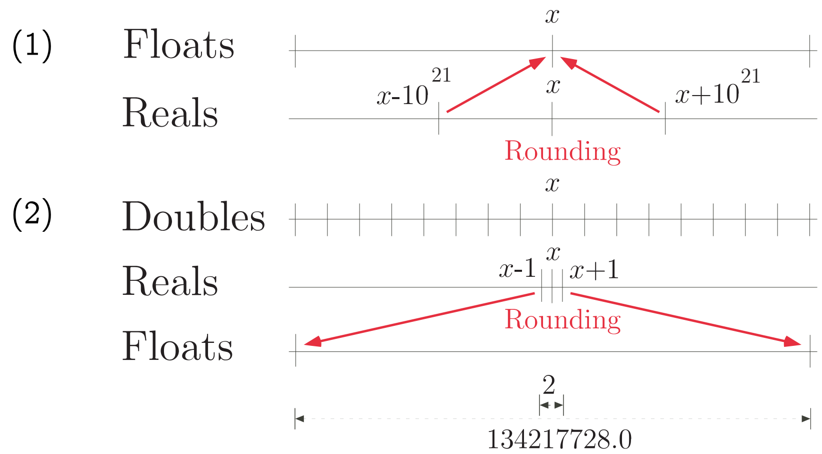
Example of rounding error (1)

```
/* 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
```

```
/* 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
```

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

Explanation of the huge rounding error



Example of rounding error (2)

```
/* 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
```

```
/* double-error.c */
int main () {
double x; float y, z, r;
/* x = ldexp(1.,50)+ldexp(1.,26); */
x = 1125899973951487.0;
y = x + 1;
z = x - 1;
r = y - z;
printf("%f\n", r);
}
% gcc double-error.c
% ./a.out
0.000000
```

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

Example of accumulation of small rounding errors

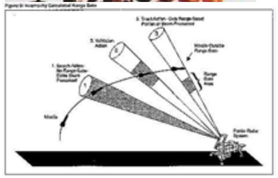
```
% ocaml
        Objective Caml version 3.08.1
# let x = ref 0.0;;
val x : float ref = {contents = 0.}
# for i = 1 to 1000000000 do
     x := !x +. 1.0/.10.0
  done; x;;
- : float ref = {contents = 99999998.7454178184}
```

since $(0.1)_{10} = (0.0001100110011001100\ldots)_2$

The Patriot missile failure

– "On February 25th, 1991, a Patriot missile ... failed to track and intercept an incoming Scud[10]."

– The software failure was due to a cumulated rounding error[11]



10 This Scud subsequently hit an Army barracks, killing 28 Americans.

11
- "Time is kept continuously by the system's internal clock in tenths of seconds"
- "The system had been in operation for over 100 consecutive hours"
- "Because the system had been on so long, the resulting inaccuracy in the time calculation caused the range gate to shift so much that the system could not track the incoming Scud"

Warranty

Excerpt from an GPL open software licence:

> NO WARRANTY. ... *BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.*

You get nothing for free!

What can be done about bugs?

Absence of Warranty

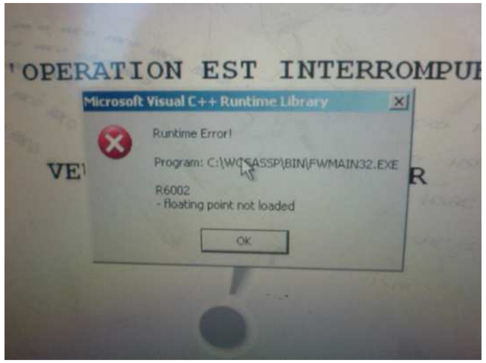
Excerpt from Microsoft software licence:

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

You get nothing for your money either!

Example of Runtime Error

HSBC ATM, 19 Boulevard Sébastopol, Paris, 11/21/2006,8:30AM:



Traditional software validation methods

- The law cannot enforce more than "best practice"
- Manual software validation methods (code reviews, simulations, tests, etc.) do not scale up
- The capacity of programmers/computer scientists remains essentially the same
- The size of software teams cannot grow significantly without severe efficiency losses

Mathematics and computers can help

- Software behavior can be mathematically formalized → semantics
- Computers can perform semantics-based program analyses to realize verification → static analysis
  - but computers are finite so there are intrinsic limitations → undecidability, complexity
  - which can only be handled by semantics approximations → abstract interpretation

Abstract interpretation

There are two fundamental concepts in computer science (and in science in general) :

- **Abstraction**: to reason on complex systems
- **Approximation**: to make effective undecidable computations

These concepts are formalized by Abstract interpretation.

References

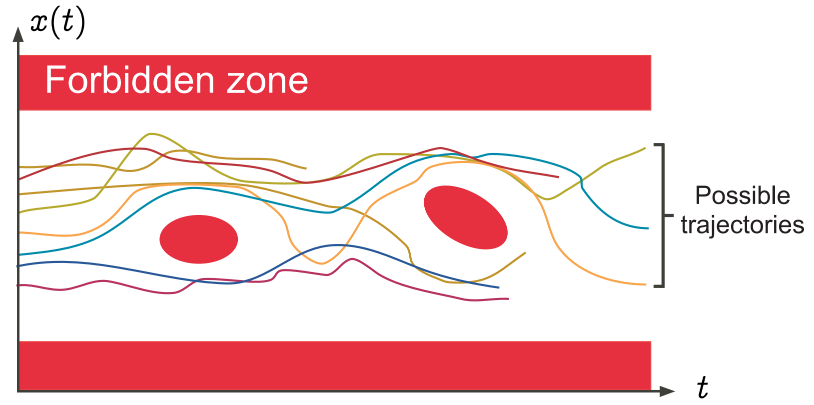
[POPL '77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In 4^{th} *ACM POPL*.

[Thesis '78] P. Cousot. Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique de programmes. Thèse ès sci. math. Grenoble, march 1978.

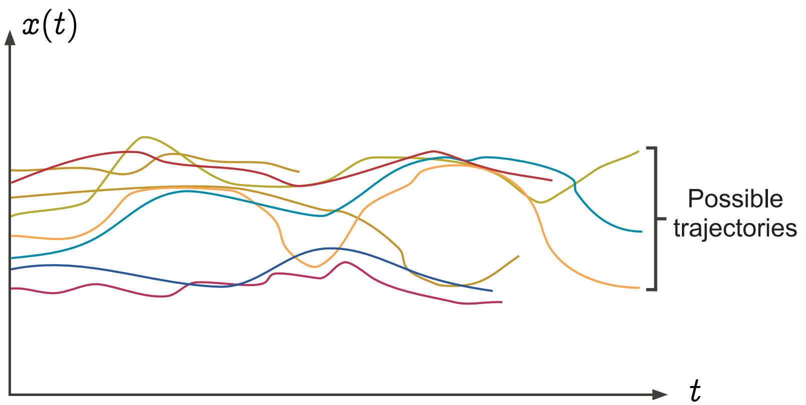
[POPL '79] P. Cousot & R. Cousot. Systematic design of program analysis frameworks. In 6^{th} *ACM POPL*.

Abstract interpretation (1) Very informal introduction

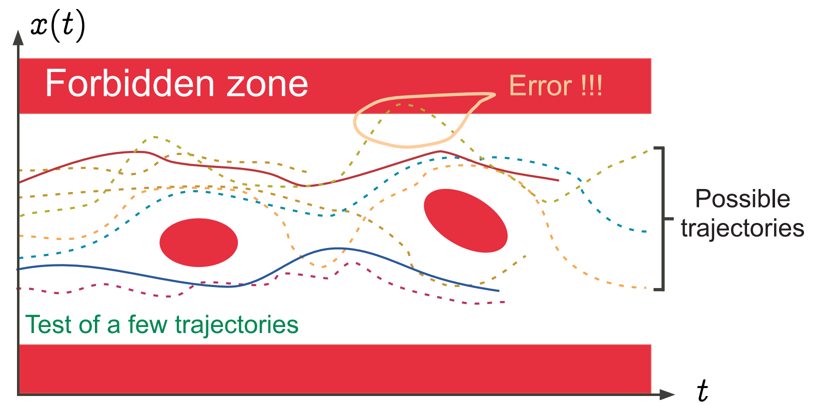
Safety property



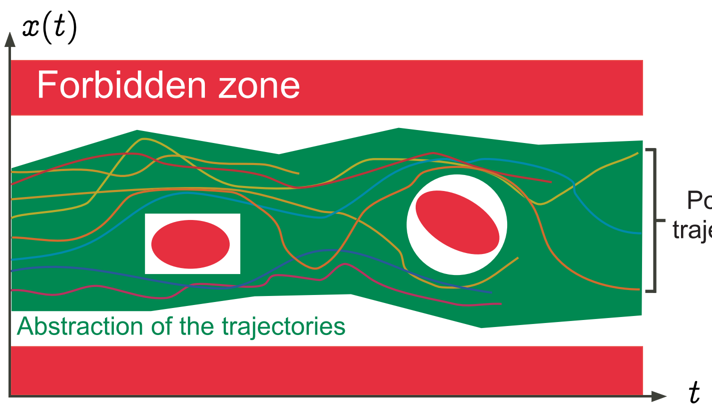
Operational semantics



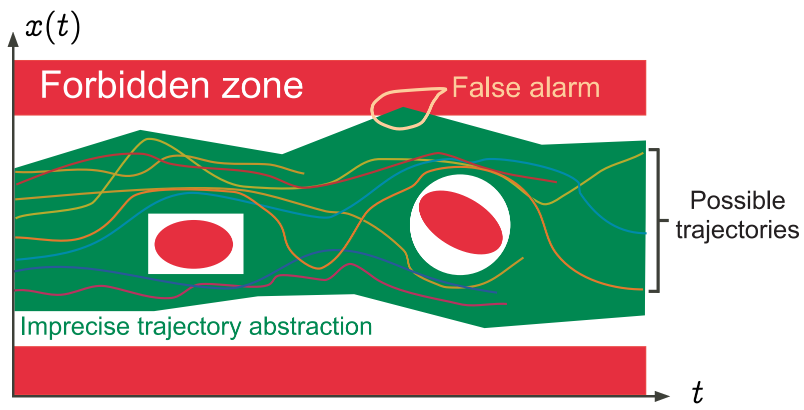
Test/debugging is unsafe



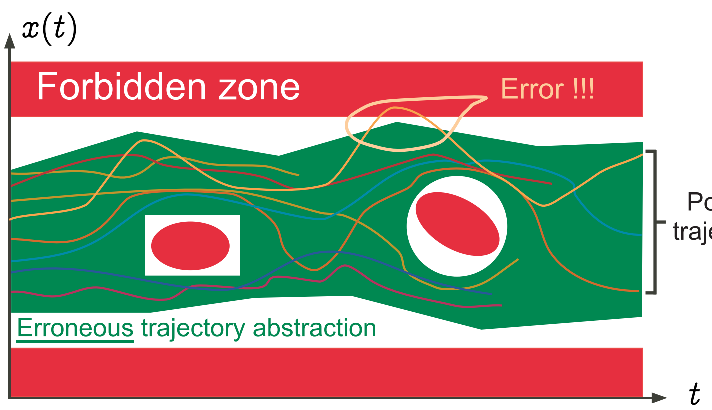
Abstract interpretation is safe



Imprecision ⇒ false alarms

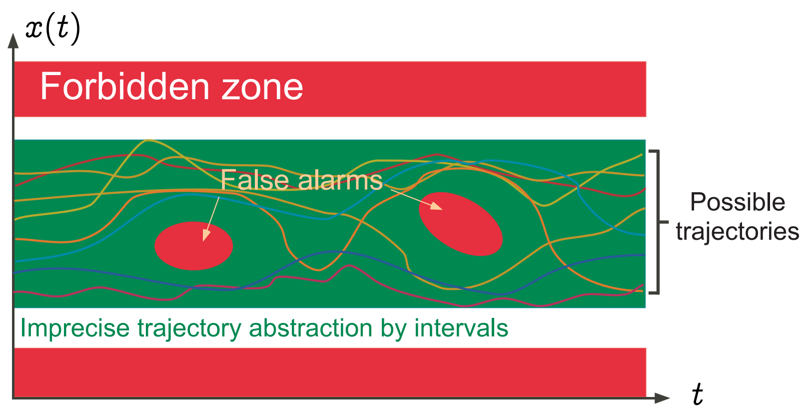


Soundness requirement: erroneous abstraction [12]

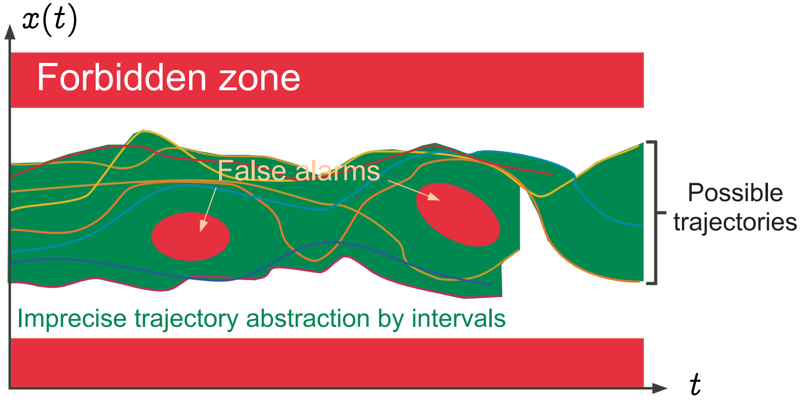


12 This situation is always excluded in static analysis by abstract interpretation.

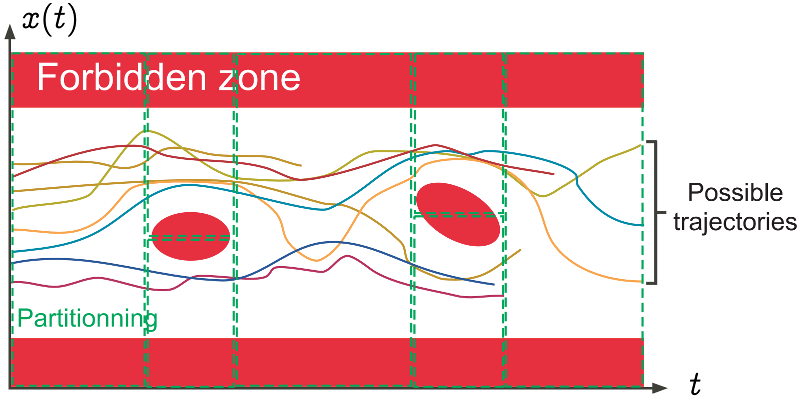
Global interval abstraction → false alarms



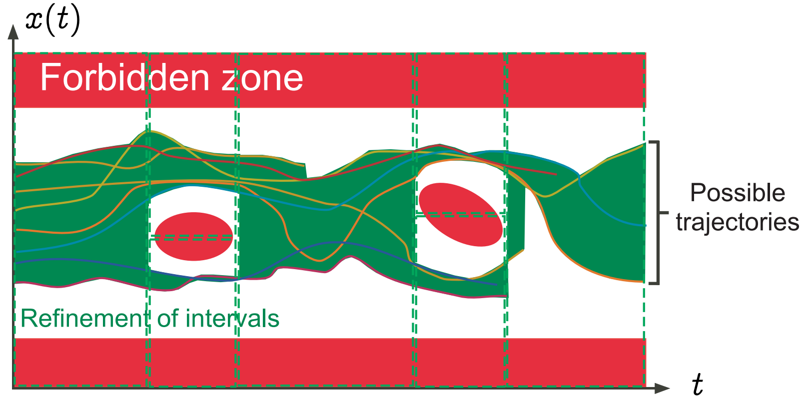
Local interval abstraction → false alarms



Refinement by partitionning



Intervals with partitionning



Abstract interpretation (2) A few elements of AI

(2.1) Program semantics

Description of a computation step

- Transition system $\langle \Sigma, \tau \rangle$, states $\Sigma = \{\bullet, \ldots, \odot \ldots\}$, transitions $\tau = \{\bullet \longrightarrow \bullet, \ldots, \bullet \longrightarrow \odot \ldots\}$
- Example
  - States : $\langle p, v \rangle$, p is a program point, v assigns values to variables
  - Transitions $\langle p, v \rangle \longrightarrow \langle p', v' \rangle$ for assignment:

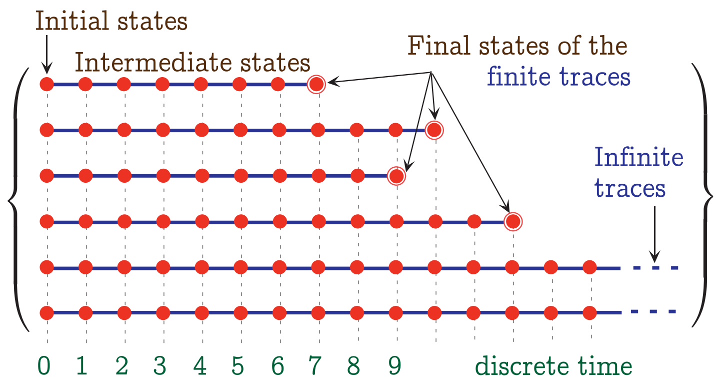
```
p:
   X = X + 1;
p'
```

$v'(\mathtt{X}) = v(\mathtt{X}) + 1$ si $v(\mathtt{X}) < \mathtt{maxint}$

$v'(\mathtt{Y}) = v(\mathtt{Y})$ si $\mathtt{Y} \neq \mathtt{X}$

Blocking state ($\odot$) if $v(\mathtt{X}) \geqslant \mathtt{maxint}$.

Description of a complete computation by a trace



States $\Sigma = \{\bullet, \ldots, \odot \ldots\}$, transitions $\tau = \{\bullet \longrightarrow \bullet, \ldots, \bullet \longrightarrow \odot \ldots\}$

Least Fixpoint Trace Semantics

Traces $= \{\odot \mid \odot$ is a final state$\}$

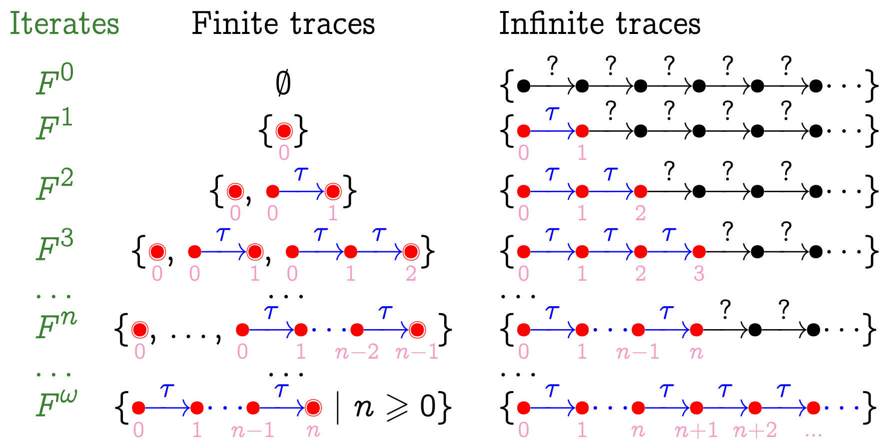
$\cup\ \{\bullet \longrightarrow \bullet \longrightarrow \ldots \longrightarrow \odot \mid \bullet \longrightarrow \bullet$ is a transition step & $\bullet \longrightarrow \ldots \longrightarrow \odot \in \text{Traces}^+\}$

$\cup\ \{\bullet \longrightarrow \bullet \longrightarrow \ldots \longrightarrow \ldots \mid \bullet \longrightarrow \bullet$ is a transition step & $\bullet \longrightarrow \ldots \longrightarrow \ldots \in \text{Traces}^\infty\}$

- In general, the equation has multiple solutions;
- Choose the least one for the computational ordering:

"more finite traces & less infinite traces".

Iterative Fixpoint Calculation of the Trace Semantics



Trace Semantics

Trace semantics of a transition system $\langle\Sigma,\tau\rangle$:

- $\Sigma^+ \stackrel{\text{def}}{=} \bigcup_{n>0} [0,n[\longmapsto \Sigma$ finite traces
- $\Sigma^\omega \stackrel{\text{def}}{=} [0,\omega[\longmapsto \Sigma$ infinite traces
- $\mathcal{S}[\![\langle\Sigma,\tau\rangle]\!] = \mathsf{lfp}^{\sqsubseteq} F \in \mathcal{D} = \wp(\Sigma^+ \cup \Sigma^\omega)$ trace semantics
- $F(X) = \{s \in \Sigma^+ \mid s \in \Sigma \wedge \forall s' \in \Sigma : \langle s, s'\rangle \notin \tau\}$
  $\cup \{ss'\sigma \mid \langle s, s'\rangle \in \tau \wedge s'\sigma \in X\}$ trace transformer
- $X \sqsubseteq Y \stackrel{\text{def}}{=} (X \cap \Sigma^+) \subseteq (Y \cap \Sigma^+) \wedge (X \cap \Sigma^\omega) \supseteq (Y \cap \Sigma^\omega)$
  computational ordering

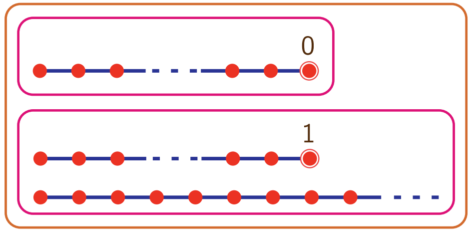
(2.2) Program Properties

Program Properties & Static Analysis

- A program property $\mathcal{P} \in \wp(\mathcal{D})$ is a set of possible semantics for that program (hence a subset of the semantic domain $\mathcal{D}$)
- A property $\mathcal{P} \in \wp(\mathcal{D})$ is stronger (or more precise) than a property $\mathcal{Q} \in \wp(\mathcal{D})$ iff $\mathcal{P} \subseteq \mathcal{Q}$ (i.e. P implies Q, $P \Rightarrow Q$)
- The strongest program property [13] is $\{\mathcal{S}[\![P]\!]\} \in \wp(\mathcal{D})$
- A static analysis effectively approximates the strongest property of programs

[13] also called the *collecting semantics*

Example of program property

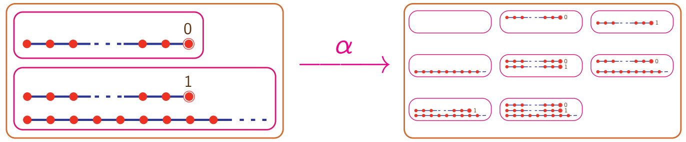


– Correct implementations: `print 0`, `print 1`, `[print 1|loop]`, ...
– Incorrect implementations: `[print 0|print 1]`

(2.3) Abstraction of Program Properties

Abstraction

– Replace actual concrete properties $\mathcal{P} \in \wp(\mathcal{D})$ by an approximate abstract properties $\alpha(\mathcal{P})$
– Example :
- $\mathcal{D} = \wp(\Sigma^+ \cup \Sigma^\omega)$ semantic domain
- $\mathcal{P} \in \wp(\mathcal{D})$ concrete properties
- $\alpha(\mathcal{P}) \stackrel{\text{def}}{=} \wp(\bigcup \mathcal{P})$ abstract properties



Commonly Required Properties of the Abstraction

– [In this talk,] we consider overapproximations:

$$\mathcal{P} \subseteq \alpha(\mathcal{P})$$

- If the abstract properties $\alpha(\mathcal{P})$ is true then the concrete properties $\mathcal{P}$ is also true
- If the abstract properties $\alpha(\mathcal{P})$ is false then the concrete properties $\mathcal{P}$ may be true[14] or false!

– All information is lost at once:

$$\alpha(\alpha(\mathcal{P})) = \alpha(\mathcal{P})$$

– The abstraction of more precise properties is more precise:

$$\text{si } \mathcal{P} \subseteq \mathcal{Q} \text{ then } \alpha(\mathcal{P}) \subseteq \alpha(\mathcal{Q})$$

[14] In this case, this is a *"false alarm"*.

Galois Connection

– We have got a Galois Connection :

$$\langle \wp(\mathcal{D}),\ \subseteq\rangle \underset{\alpha}{\overset{1}{\leftrightarrows}} \langle \wp(\mathcal{D}),\ \subseteq\rangle$$

$\uparrow$ Concrete properties — $\uparrow$ Abstract properties

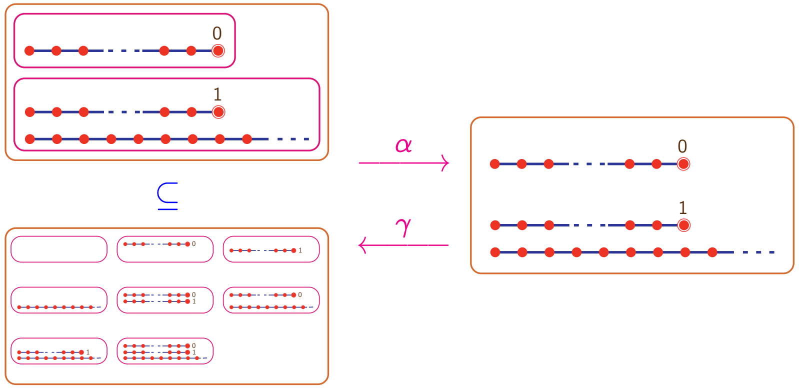
– With an isomorphic mathematical/computer representation:

$$\langle \wp(\mathcal{D}),\ \subseteq\rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \mathcal{D}^\sharp,\ \sqsubseteq\rangle$$

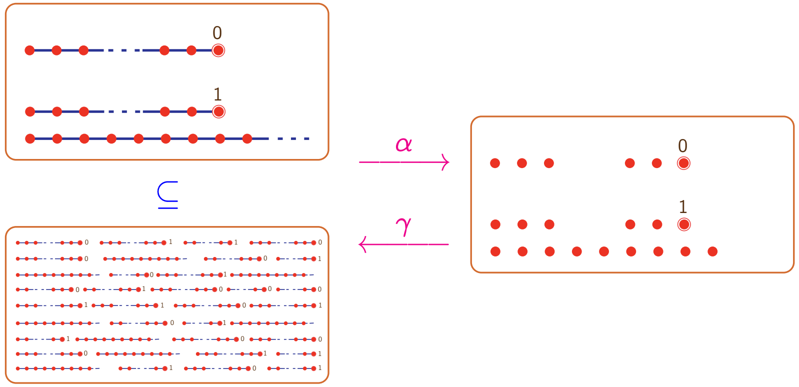
$\uparrow$ Concrete properties — $\uparrow$ Abstract domain

$$\forall \mathcal{P} \in \wp(\mathcal{D}) : \forall \mathcal{Q} \in \mathcal{D}^\sharp : \alpha(\mathcal{P}) \sqsubseteq \mathcal{Q} \iff \mathcal{P} \subseteq \gamma(\mathcal{Q})$$

Example 1 de Galois Connection



Example 2 de Galois Connection



Function Abstraction

– Let $\langle \wp(\mathcal{D}),\ \subseteq\rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \mathcal{D}^\sharp,\ \sqsubseteq\rangle$

– How to abstract an *operator* $F \in \wp(\mathcal{D}) \overset{\leqslant}{\longmapsto} \wp(\mathcal{D})$?

– The most precise sound overapproximation is

$$F^\sharp \in \mathcal{D}^\sharp \overset{\leqslant}{\longmapsto} \mathcal{D}^\sharp$$
$$F^\sharp = \alpha \circ F \circ \gamma$$

– This is a Galois Connection

$$\langle \wp(\mathcal{D}) \overset{\leqslant}{\longmapsto} \wp(\mathcal{D}),\ \subseteq\rangle \underset{\lambda F \cdot \alpha \circ F \circ \gamma}{\overset{\lambda F^\sharp \cdot \gamma \circ F^\sharp \circ \alpha}{\leftrightarrows}} \langle \mathcal{D}^\sharp \overset{\leqslant}{\longmapsto} \mathcal{D}^\sharp,\ \sqsubseteq\rangle$$

Fixpoint Abstraction

- Let $\langle \wp(\mathcal{D}),\ \subseteq\rangle \xleftarrow[\alpha]{\gamma} \langle \mathcal{D}^\sharp,\ \sqsubseteq\rangle$
- How to abstract a *fixpoint property* $\mathsf{lfp}^{\subseteq}\ F$ where $F \in \wp(\mathcal{D}) \overset{\leqslant}{\longmapsto} \wp(\mathcal{D})$?
- Approximate sound abstraction:

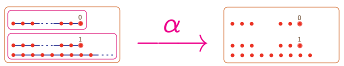
$$\mathsf{lfp}^{\subseteq}\ F \subseteq \gamma(\mathsf{lfp}^{\sqsubseteq}\ \alpha \circ F \circ \gamma)$$

- Complete abstraction: if $\alpha \circ F = F^\sharp \circ \alpha$ then

$$F^\sharp = \alpha \circ F \circ \gamma, \text{ and}$$

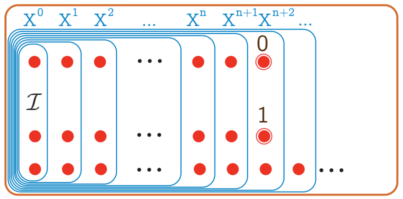
$$\alpha(\mathsf{lfp}^{\subseteq}\ F) = \mathsf{lfp}^{\sqsubseteq}\ F^\sharp$$

Example: Accessible States

- Transition system: $\langle \Sigma,\ \tau\rangle$
- Initial states: $\mathcal{I} \subseteq \Sigma$
- Abstraction: 

- Accessible states: $\mathsf{lfp}^{\subseteq}\ F^\sharp$,

$$F^\sharp(X) = \mathcal{I} \cup \{s' \mid \exists s \in X : \langle s,\ s'\rangle \in \tau\}$$



Convergence acceleration of the iterative fixpoint computation

- The fixpoint $\mathsf{lfp}^{\sqsubseteq}\ F^\sharp$, $F^\sharp \in \mathcal{D}^\sharp \overset{\leqslant}{\longmapsto} \mathcal{D}^\sharp$ is computed iteratively [15]:

$$X^0 = \bot \qquad X^{n+1} = F^\sharp(X^n) \qquad X^\omega = \bigsqcup_{n \geqslant 0} X^n$$

- For systems of equations $\mathcal{D}^\sharp = \prod_{i=1}^{n} \mathcal{D}^\sharp_n$, we use asynchronous iterations
- Convergence acceleration techniques have been developped to overapproximate the limit.

15 $\langle \mathcal{D}^\sharp,\ \sqsubseteq\rangle$ is a poset, $F^\sharp$ is monotonic, $\bot$ is the infimum, the least upper bound $\sqcup$ must exist for the iterates (in general transfinite).

Static analysis by abstract interpretation

1. Define the programming language semantics $\mathcal{S} \in \mathcal{L} \mapsto \mathcal{D}$ and the concrete properties $\wp(\mathcal{D})$;
2. Let $\mathcal{Q} \in \wp(\mathcal{D})$ be the property to be proved about the program P : $\mathcal{S}[\![P]\!] \in \mathcal{Q}$
3. Choose the abstraction $\langle \wp(\mathcal{D}),\ \subseteq\rangle \xleftarrow[\alpha]{\gamma} \langle \mathcal{D}^\sharp,\ \sqsubseteq\rangle$
4. The abstract interpretation theory formally define an abstract semantics $\mathcal{S}^\sharp[\![P]\!] \sqsupseteq \alpha(\{\mathcal{S}[\![P]\!]\})$
5. The static analysis algorithm is the computation/overapproximation of the abstract semantics (whence correct by construction)

6. The result of the computation is either
   - $\mathcal{S}[\![P]\!] \in \gamma(\mathcal{S}^\sharp[\![P]\!]) \subseteq \mathcal{Q}$ (correctness proof), or
   - $\gamma(\mathcal{S}^\sharp[\![P]\!]) \not\subseteq \mathcal{Q}$ (property not satisfied (error) or approximation too coarse (false alarm))
7. The abstraction must be chosen in terms of the property $\mathcal{Q}$ to be proved, to be
   - coarse enough to be automatically computatble,
   - precise enough to obtain a correctness proof: $\gamma(\mathcal{S}^\sharp[\![P]\!]) \subseteq \mathcal{Q}$;

Abstract interpretation (3) A few applications

Applications of Abstract Interpretation

Any reasoning on complex computer systems must involve a correct approximation of their behaviors, as formalized by Abstract Interpretation [5, 20, 21, 34]

- Syntax of programming languages [30]
- Semantics of programming languages [13, 27]
- Proofs of programs [11, 12]
- Typing and type inference [18]
- Model-checking [23, 28, 31]
- Bisimulations [42]

- Static analysis of programming languages [3, 7, 15, 16, 22, 26]
  - imperative [2, 4, 6, 9, 19]
  - parallel [10, 8]
  - logic/constraint [14]
  - fonctionnal [17]
- Transformation of programs [29]
- Steganography [33]
- Obfuscation [36]
- Malware detection [37]
- ...

Abstract interpretation
(4) Application to critical software

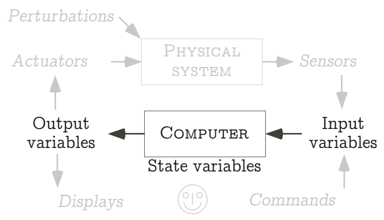
(4.1) The ASTRÉE static analyzer

www.astree.ens.fr [25, 32, 35]

ASTRÉE is a specialized static analyzer

– Embedded real-time synchronous control/command C programs:

```
Declare and initialize state
variables;
loop forever
  read volatile input variables,
  compute output and
  state variables,
  write state variables;
  wait for next clock tick
end loop
```

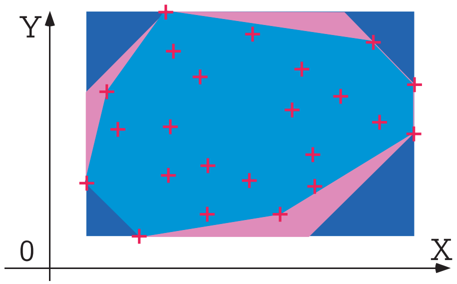


Objective of ASTRÉE

– Prove automatically the absence of runtime errors:
  - No division by 0, NaN, out of range array access, nil/dangling pointer
  - No signed integer/float overflows
  - Verification of user-defined properties (for example machine dependent properties)
– Requirements:
  - efficiency (must operate on a workstation)
  - precision (few false alarms)
– No alarm → full certification

(4.2) Examples of abstractions

General purpose numerical abstract domains



Approximation of a set of points

Intervals: [2]

$$\bigwedge_{i=1}^{n} a_i \leqslant x_i \leqslant b_i$$

Octagons: [40]

$$\bigwedge_{i=1}^{n} \bigwedge_{j=1}^{n} \pm x_i \pm y_j \leqslant a_{ij}$$

Polyhedra: [6]

$$\bigwedge_{j=1}^{m} (\sum_{i=1}^{n} a_{ji} x_i) \leqslant b_j$$

Floating-point linearization [40, 41]

- 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 $\mathtt{X} \in$ [-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

Symbolic abstract domain [40, 41]

- Interval analysis: if $x \in [a, b]$ and $y \in [c, d]$ then $x - y \in [a - d, b - c]$ so if $x \in [0, 100]$ then $x - x \in [-100, 100]$!!!
- The symbolic abstract domain propagates the symbolic values of variables and performs simplifications;
- Must maintain the maximal possible rounding error for float computations (overestimated with intervals);

```
% cat -n x-x.c
     1  void main () { int X, Y;
     2          __ASTREE_known_fact(((0 <= X) && (X <= 100)));
     3          Y = (X - X);
     4          __ASTREE_log_vars((Y));
     5  }
```

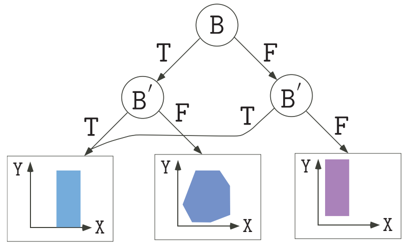
```
astree -exec-fn main -no-relational x-x.c
Call main@x-x.c:1:5-x-x.c:1:9:
<interval: Y in [-100, 100]>
```

```
astree -exec-fn main x-x.c
Call main@x-x.c:1:5-x-x.c:1:9:
<interval: Y in {0}> <symbolic: Y = (X -i 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;
    }
    ...
  }
}
```



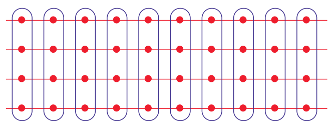
The boolean relation abstract domain is parameterized by the height of the decision tree (an analyzer option) and the abstract domain at the leafs

Control Partitionning for Case Analysis

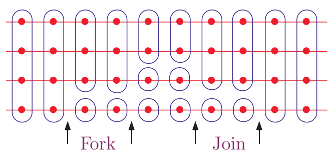
– Code Sample:

```
/* 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:



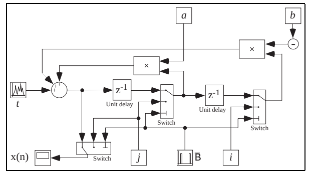
Trace partitionning:



Delaying abstract unions in tests and loops is more precise for non-distributive abstract domains (and much less expensive than disjunctive completion).

Ellipsoid Abstract Domain for Filters

2nd Order Digital Filter:

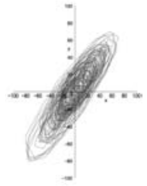


– Computes $X_n = \begin{cases} \alpha X_{n-1} + \beta X_{n-2} + Y_n \\ I_n \end{cases}$

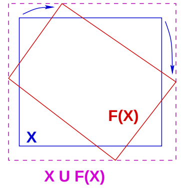
– The concrete computation is bounded, which must be proved in the abstract.

– There is no stable interval or octagon.

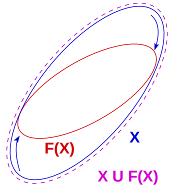
– The simplest stable surface is an ellipsoid.



execution trace



unstable interval



stable ellipsoid

Filter Example [38]

```
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; }
}
```

Slow divergences by rounding accumulation

```
X = 1.0;
while (TRUE) { ①
  X = X / 3.0;
  X = X * 3.0;
}
```

- With reals $\mathbb{R}$: $x = 1.0$ at ①
- With floats: rounding errors
- Accumulation of rounding errors: possible cause of divergence

Solution [35] : bound the cumulated rounding error as a function of the number of iterations by arithmetico-geometric progressions:

- Relation $|\mathtt{X}| \leq a \cdot b^n + c$, where a, b, c are constants determined by the analysis, n is the iterate number
- Number of iterates bounded by N : $|\mathtt{X}| \leq a \cdot b^N + c$

Arithmetic-geometric progressions [16] [39]

- Abstract domain: $(\mathbb{R}^+)^5$
- Concretization:

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

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

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

[16] here in $\mathbb{R}$

Arithmetic-Geometric Progressions (Example 1)

```
% 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.
```


Arithmetic-geometric progressions (Example 2)

```
% 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
```


(Automatic) Parameterization

– All abstract domains of ASTRÉE are parameterized, e.g.
  - variable packing for octagones and decision trees,
  - partition/merge program points,
  - loop unrollings,
  - thresholds in widenings, ...;
– End-users can either parameterize by hand (analyzer options, directives in the code), or
– choose the automatic parameterization (default options, directives for pattern-matched predefined program schemata).

Possible origins of imprecision and how to fix it

In case of false alarm, the imprecision can come from:

– Abstract transformers (not best possible) ⟶ improve algorithm;
– Automatized parametrization (e.g. variable packing) ⟶ improve pattern-matched program schemata;
– Iteration strategy for fixpoints ⟶ fix widening [17];
– Inexpressivity i.e. indispensable local inductive invariant are inexpressible in the abstract ⟶ add a new abstract domain to the reduced product (e.g. filters).

[17] This can be very hard since at the limit only a precise infinite iteration might be able to compute the proper abstract invariant. In that case, it might be better to design a more refined abstract domain.

(4.3) Results

Application to the A 340/A 380

– Primary flight control software of the electric flight control system of the Airbus A340 family and the A380


– C program, automatically generated from of high-level specification (à la Simulink/SCADE)
– A340 : 100.000 to 250.000 LOCs
– A380 : 400.000 to 1.000.000 LOCs

A world première

– In Nov. 2005, analysis of 400.000 lines of C code[18]:

time	memory	false alarms
13h 52mn	2,2 Gb	**0**

– In Nov. 2006, analysis of 750.000 lines of C code:

time	memory	false alarms
34h 30mn	4,8 Go	**0**

[18] on an AMD Opteron 248, 64 bits, a single processor

Current Projects

Static Analysis of Synchronous Programs

– MSU[19] of the FAS[20] for the ATV[21]—ISS[22] rendez-vous (mission critical)

– C version of an ADA program generated from Simulink + Scade + manual code

– 190 000 LOCs

[19] MSU: Monitoring and Safing Unit
[20] FAS: Flight Application Software
[21] ATV: Automated Trasfer Vehicle
[22] ISS: International Space Station

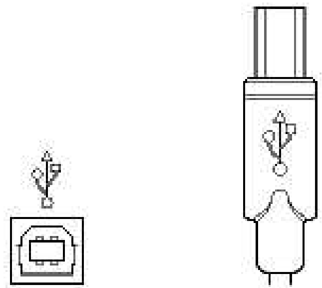
Static Analysis of Asynchronous Controlers

– USB communications

– Driver + Model of the (hardware) controler

– Asynchonous interferences between driver/ controler[23]

– Low-level data structures



[23] SLAM for example is unsound since it is purely sequential on the driver and completely ignores the controler interfering asynchronously in parallel.

Static Analysis of Asynchronous Programs

– Parallel processes
– Shared variables/ semaphors & message communications
– Scheduling with static priorities/delays on waits
– Example application: flight warning system of commercial planes (about 3 500 000 Locs)

Results of the ASTRÉE project

– ASTRÉE is a practical proof that software static analysis by abstract interpretation does scale up [24]
– *With a lot of efforts*, theoretical & purely speculative research on abstract interpretation can find its way into industrial practice,
– Effective industrial use, if the methodology changes and cost are marginal [25]
– Forthcoming commercialization (1/3 years)

[24] The main difficulty for formal methods.
[25] Developing and maintaining a formal proof or a finite model of a large program, would not be considered as marginal methodology changes and cost!

Conclusion

THE END, THANK YOU

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

Bibliographic References

[2] P. Cousot and R. Cousot. – Static determination of dynamic properties of programs. *In : Proceedings of the Second International Symposium on Programming*, Paris, 1976. pp. 106–130. – Dunod, Paris.

[3] P. Cousot and R. 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*, Los Angeles, Californie, 1977. pp. 238–252. – ACM Press, New York, New York, USA.

[4] P. Cousot and R. Cousot. – Static determination of dynamic properties of recursive procedures. *In : IFIP Conference on Formal Description of Programming Concepts, St-Andrews, N.B., Canada*, edited by E. Neuhold. pp. 237–277. – North-Holland Pub. Co., Amsterdam, Pays-Bas, 1977.

[5] 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, Grenoble, 21 mars 1978.

[6] P. Cousot and N. Halbwachs. – Automatic discovery of linear restraints among variables of a program. *In : Conference Record of the Fifth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Tucson, Arizona, 1978. pp. 84–97. – ACM Press, New York, New York, USA.

[7] P. Cousot and R. Cousot. – Systematic design of program analysis frameworks. *In : Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Antonio, Texas, 1979. pp. 269–282. – ACM Press, New York, New York, USA.

[8] P. Cousot and R. Cousot. – Semantic analysis of communicating sequential processes. *In : Seventh International Colloquium on Automata, Languages and Programming*, edited by J. de Bakker and J. van Leeuwen. *Lecture Notes in Computer Science 85*, pp. 119–133. – Springer, Berlin, Allemagne, juillet 1980.

[9] P. Cousot. – Semantic Foundations of Program Analysis, chapitre invité. *In : Program Flow Analysis: Theory and Applications*, edited by S. Muchnick and N. Jones, Chapter 10, pp. 303–342. – Prentice-Hall, Inc., Englewood Cliffs, New Jersey, USA, 1981.

[10] P. Cousot and R. Cousot. – Invariance Proof Methods and Analysis Techniques For Parallel Programs, chapitre invité. *In : Automatic Program Construction Techniques*, edited by A. Biermann, G. Guiho and Y. Kodratoff, Chapter 12, pp. 243–271. – Macmillan, New York, New York, USA, 1984.

[11] P. Cousot and R. Cousot. – 'À la Floyd' induction principles for proving inevitability properties of programs, chapitre invité. *In : Algebraic Methods in Semantics*, edited by M. Nivat and J. Reynolds, Chapter 8, pp. 277–312. – Cambridge University Press, Cambridge, Royaume Uni, 1985.

[12] P. Cousot. – Methods and Logics for Proving Programs, chapitre invité. *In : Formal Models and Semantics*, edited by J. van Leeuwen, Chapter 15, pp. 843–993. – Elsevier Science Publishers B.V., Amsterdam, Pays-Bas, 1990, *Handbook of Theoretical Computer Science*, Vol. B.

[13] P. Cousot and R. Cousot. – Inductive Definitions, Semantics and Abstract Interpretation. *In : Conference Record of the Ninthteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Albuquerque, Nouveau Mexique, USA, 1992. pp. 83–94. – ACM Press, New York, New York, USA.

[14] P. Cousot and R. Cousot. – Abstract Interpretation and Application to Logic Programs. *Journal of Logic Programming*, Vol. 13, nº 2–3, 1992, pp. 103–179. – (The editor of Journal of Logic Programming has mistakenly published the unreadable galley proof. For a correct version of this paper, see http://www.di.ens.fr/~cousot.).

[15] P. Cousot and R. Cousot. – Abstract Interpretation Frameworks. *Journal of Logic and Computation*, Vol. 2, nº 4, août 1992, pp. 511–547.

[16] P. Cousot and R. Cousot. – Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation, papier invité. *In : Proceedings of the Fourth International Symposium Programming Language Implementation and Logic Programming, PLILP '92*, edited by M. Bruynooghe and M. Wirsing. *Louvain, Belgique, 26–28 août 1992, Lecture Notes in Computer Science 631*, pp. 269–295. – Springer, Berlin, Allemagne, 1992.

[17] P. Cousot and R. Cousot. – Higher-Order Abstract Interpretation (and Application to Comportment Analysis Generalizing Strictness, Termination, Projection and PER Analysis of Functional Languages), papier invité. *In : Proceedings of the 1994 International Conference on Computer Languages*, Toulouse, 16–19 mai 1994. pp. 95–112. – IEEE Computer Society Press, Los Alamitos, Californie, USA.

[18] P. Cousot. – Types as Abstract Interpretations, papier invité. *In : Conference Record of the Twentyfourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Paris, janvier 1997. pp. 316–331. – ACM Press, New York, New York, USA.

[19] P. Cousot. – The Calculational Design of a Generic Abstract Interpreter, chapitre invité. *In : Calculational System Design*, edited by M. Broy and R. Steinbrüggen, pp. 421–505. – NATO Science Series, Series F: Computer and Systems Sciences. IOS Press, Amsterdam, Pays-Bas, 1999, Volume 173.

[20] P. Cousot. – Interprétation abstraite. *Technique et science informatique*, Vol. 19, nº 1-2-3, janvier 2000, pp. 155–164.

[21] P. Cousot. – Abstract Interpretation Based Formal Methods and Future Challenges, chapitre invité. *In : « Informatics — 10 Years Back, 10 Years Ahead »*, edited by R. Wilhelm, pp. 138–156. – Springer, Berlin, Allemagne, 2001, *Lecture Notes in Computer Science* , Vol. 2000.

[22] P. Cousot. – Partial Completeness of Abstract Fixpoint Checking, papier invité. *In : Proceedings of the Fourth International Symposium on Abstraction, Reformulation and Approximation, SARA '2000*, edited by B. Choueiry and T. Walsh, pp. 1–25. – Springer, Berlin, Allemagne, 26–29 juillet 2000, *Horseshoe Bay, Texas, USA, Lecture Notes in Artificial Intelligence 1864.*

[23] P. Cousot and R. Cousot. – Temporal Abstract Interpretation. *In : Conference Record of the Twentyseventh Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Boston, Massachusetts, USA, janvier 2000. pp. 12–25. – ACM Press, New York, New York, USA.

[24] P. Cousot and R. Cousot. – Static Analysis of Embedded Software: Problems and Projects, papier invité. *In : Proceedings of the First International Workshop on Embedded Software, EMSOFT '2001*, edited by T. Henzinger and C. Kirsch. *Lecture Notes in Computer Science* , Vol. 2211, pp. 97–113. – Springer, Berlin, Allemagne, 2001.

[25] 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, chapitre invité. *In : The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, edited by T. Mogensen, D. Schmidt and I. Sudborough, pp. 85–108. – Springer, Berlin, Allemagne, 2002, *Lecture Notes in Computer Science 2566.*

[26] P. Cousot and R. Cousot. – Modular Static Program Analysis, papier invité. *In : Proceedings of the Eleventh International Conference on Compiler Construction, CC '2002*, edited by R. Horspool, Grenoble, 6–14 avril 2002. pp. 159–178. – Lecture Notes in Computer Science 2304, Springer, Berlin, Allemagne.

[27] P. Cousot. – Constructive Design of a Hierarchy of Semantics of a Transition System by Abstract Interpretation. *Theoretical Computer Science*, Vol. 277, n^{o} 1—2, 2002, pp. 47–103.

[28] P. Cousot and R. Cousot. – On Abstraction in Software Verification, papier invité. *In : Proceedings of the Fourteenth International Conference on Computer Aided Verification, CAV '2002*, edited by E. Brinksma and K. Larsen. *Copenhague, Danemark, Lecture Notes in Computer Science 2404*, pp. 37–56. – Springer, Berlin, Allemagne, 27–31 juillet 2002.

[29] P. Cousot and R. Cousot. – Systematic Design of Program Transformation Frameworks by Abstract Interrpetation. *In : Conference Record of the Twentyninth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Portland, Oregon, USA, janvier 2002. pp. 178–190. – ACM Press, New York, New York, USA.

[30] P. Cousot and R. Cousot. – Parsing as Abstract Interpretation of Grammar Semantics. *Theoretical Computer Science*, Vol. 290, n^{o} 1, janvier 2003, pp. 531–544.

[31] P. Cousot. – Verification by Abstract Interpretation, chapitre invité. *In : Proceedings of the International Symposium on Verification – Theory & Practice – Honoring Zohar Manna's 64th Birthday*, edited by N. Dershowitz, pp. 243–268. – Taormina, Italie, Lecture Notes in Computer Science 2772, Springer, Berlin, Allemagne, 29 juin – 4 juillet 2003.

[32] 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 : Proceedings of the ACM SIGPLAN '2003 Conference on Programming Language Design and Implementation (PLDI)*, San Diego, Californie, USA, 7–14 juin 2003. pp. 196–207. – ACM Press, New York, New York, USA.

[33] P. Cousot and R. Cousot. – An Abstract Interpretation-Based Framework for Software Watermarking. *In : Conference Record of the Thirtyfirst Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Venise, Italie, 14–16 janvier 2004. pp. 173–185. – ACM Press, New York, New York, USA.

[34] P. Cousot and R. Cousot. – Basic Concepts of Abstract Interpretation% invitedchapter. *In : Building the Information Society*, edited by P. Jacquart, Chapter 4, pp. 359–366. – Kluwer Academic Publishers, Dordrecht, Pays-Bas, 2004.

[35] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux and X. Rival. – The ASTRÉE analyser. *In : Proceedings of the Fourteenth European Symposium on Programming Languages and Systems, ESOP '2005, Édimbourg, Écosse*, edited by M. Sagiv, pp. 21–30. – Springer, Berlin, Allemagne, 2–10 avril 2005, *Lecture Notes in Computer Science* , Vol. 3444.

[36] M. Dalla Preda and R. Giacobazzi. Semantic-based Code Obfuscation by Abstract Interpretation. In Proc. 32nd Int. Colloquium on Automata, Languages and Programming (ICALP'05 – Track B). LNCS, 2005 Springer-Verlag. July 11-15, 2005, Lisboa, Portugal. To appear.

[37] M. D. Preda, M. Christodorescu, S. Jha and S. Debray. – A Semantics-Based Approach to Malware Detection. *In : Conference Record of the Thirtyfourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Nice, France, 2007. – ACM Press, New York, New York, United States. To appear.

[38] J. Feret. – Static analysis of digital filters. – *ESOP'04*, Barcelone, Espagne, *Lecture Notes in Computer Science* , Vol. 2986, pp. 33—-48, Springer, Berlin, Allemagne, 2004.

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

[40] A. Miné. – Relational abstract domains for the detection of floating-point run-time errors. *ESOP'04*, Barcelone, Espagne, *Lecture Notes in Computer Science* , Vol. 2986, pp. 3—-17, Springer, Berlin, Allemagne, 2004.

[41] A. Miné. Weakly Relational Numerical Abstract Domains. *PhD Thesis*, École Polytechnique, 6 december 2004.

[42] F. Ranzato and F. Tapparo. Strong Preservation as Completeness in Abstract Interpretation. ESOP 2004, Barcelona, Spain, March 29 - April 2, 2004, D.A. Schmidt (Ed), LNCS 2986, Springer, 2004, pp. 18–32.