Automatic Verification of Avionic Synchronous Safety Critical Embedded Software

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

Wednesday Seminar Series, The Computer Laboratory
University of Cambridge, UK, Oct. 20th, 2004

Talk Outline

Motivation

All Computer Scientists Have Experienced Bugs

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

Static Analysis by Abstract Interpretation

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

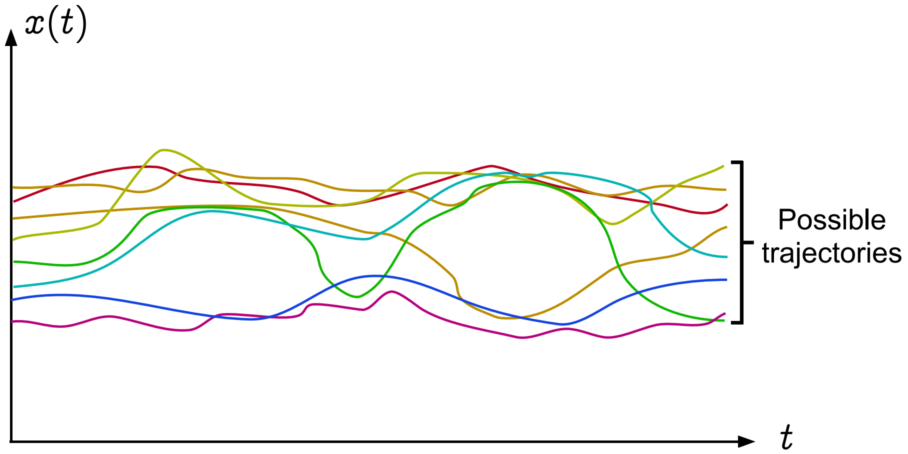
Undecidability $\longrightarrow$

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

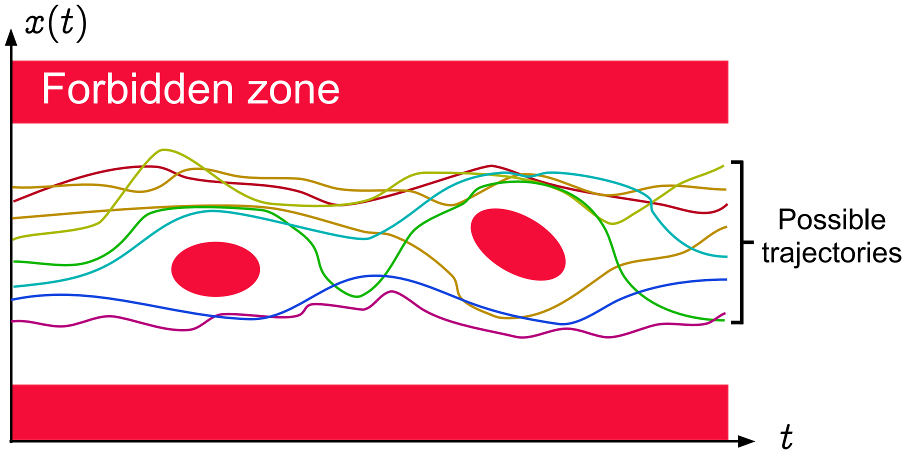
- which is precise enough to imply the desired property, and
- coarse enough to be efficiently computable.

Abstract Interpretation, Informally

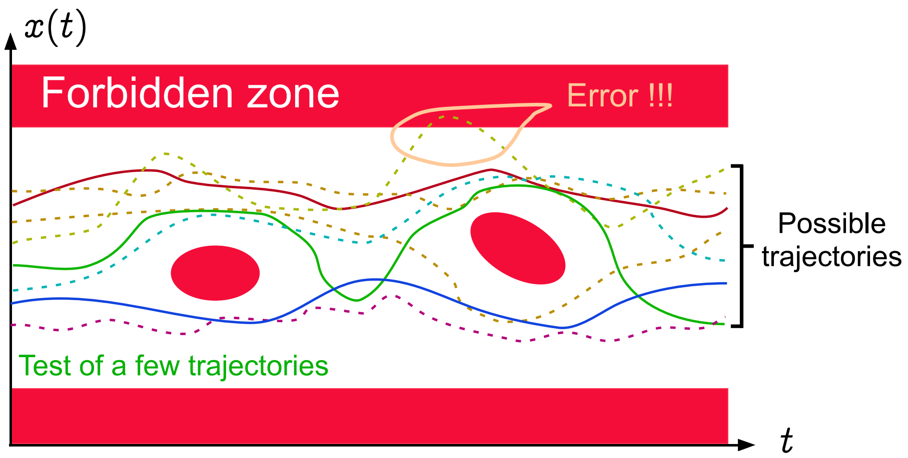
Operational Semantics



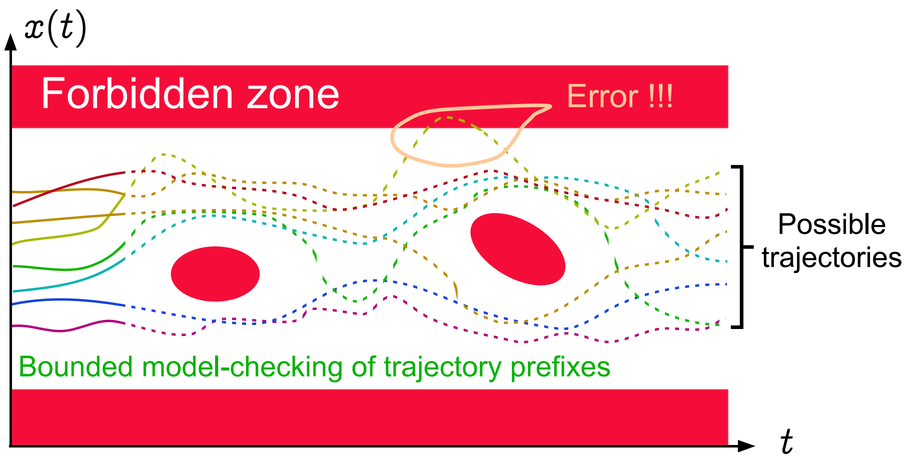
Safety property



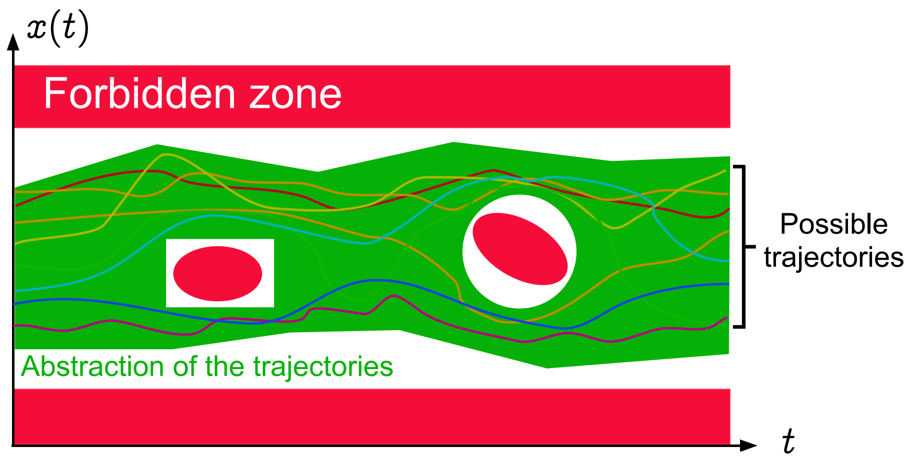
Test/Debugging is Unsafe



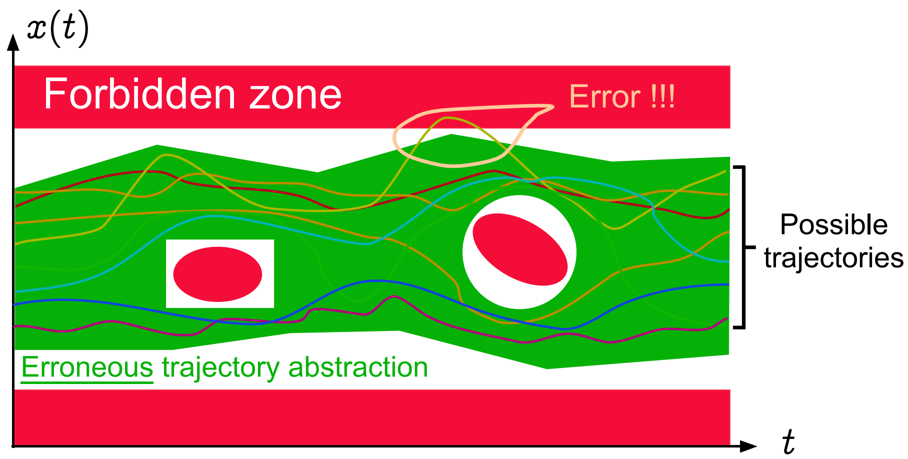
Bounded Model Checking is Unsafe



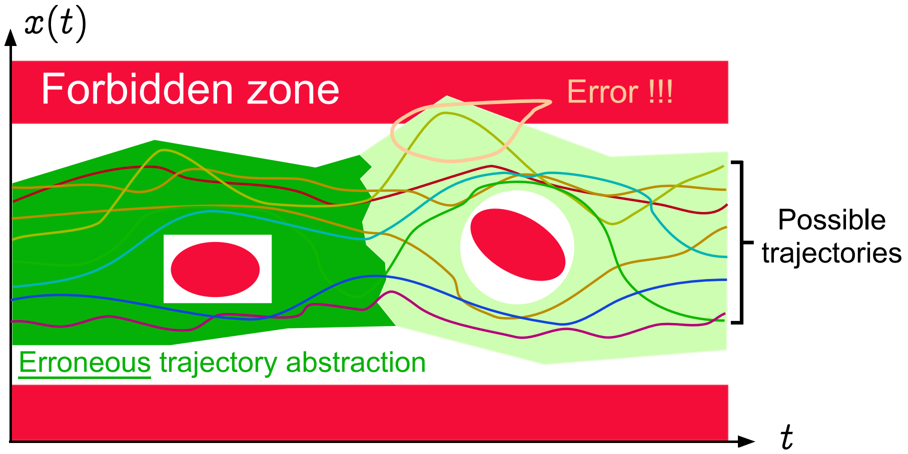
Abstract Interpretation



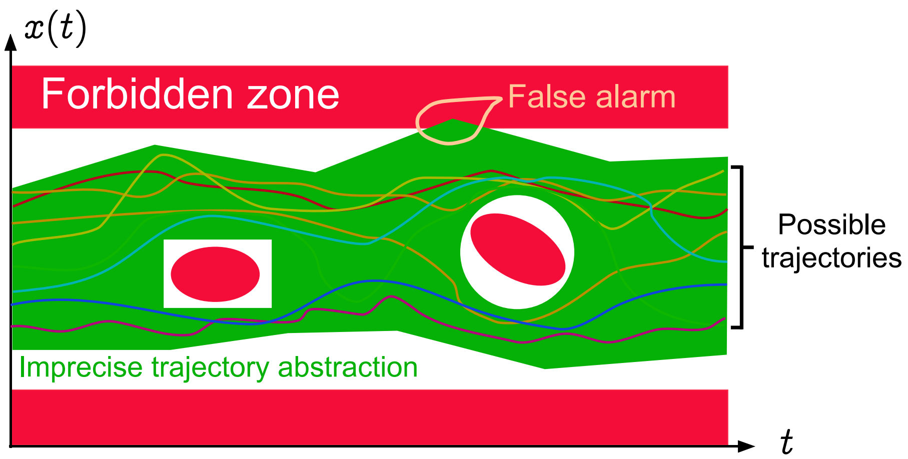
Soundness: Erroneous Abstraction — I



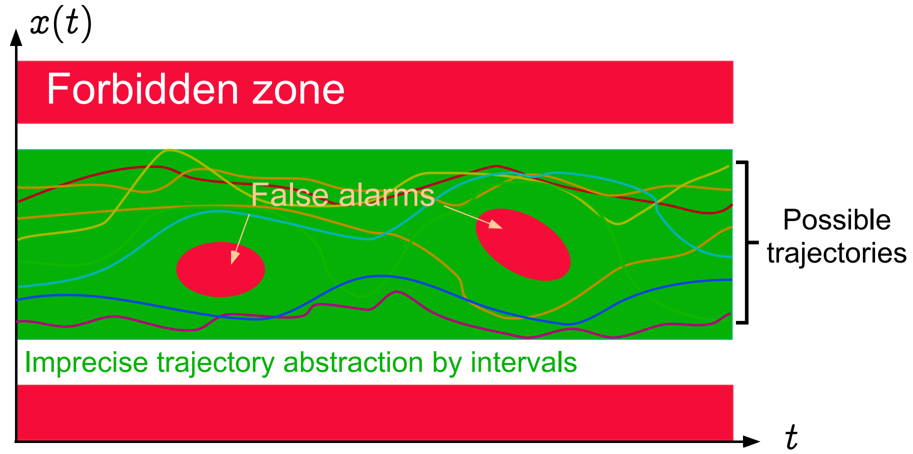
Soundness: Erroneous Abstraction — II



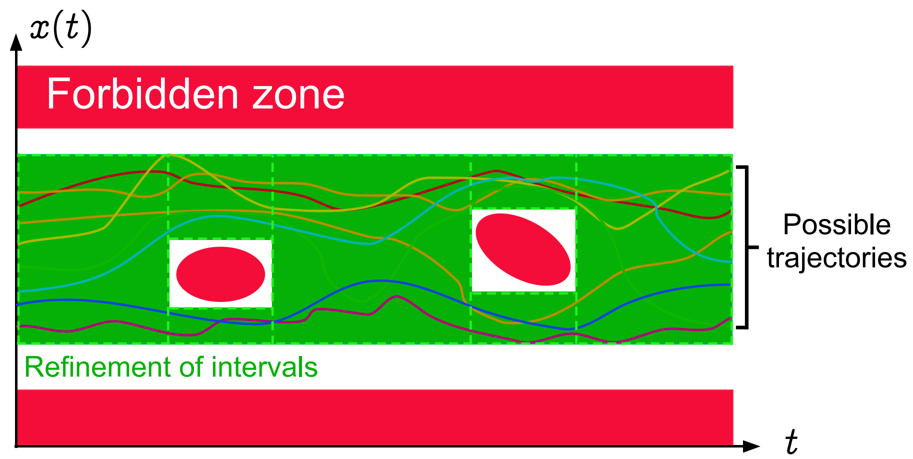
Imprecision $\Rightarrow$ False Alarms



Interval Abstraction $\Rightarrow$ False Alarms



Refinement by Partitionning



Abstract Interpretation, formal sketch

Reference

[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*.

Syntax of Programs

X	variables $X \in \mathbb{X}$
T	types $T \in \mathbb{T}$
E	arithmetic expressions $E \in \mathbb{E}$
B	boolean expressions $B \in \mathbb{B}$
$D ::= T\ X\,;$	declarations $D \in \mathbb{D}$, $\mathrm{vars}(D) = \{X\}$
$\mid\ T\ X\ ;\ D'$	$X \notin \mathrm{vars}(D')$, $\mathrm{vars}(D) = \{X\} \cup \mathrm{vars}(D')$
$C ::= X$ = E;	commands $C \in \mathbb{C}$ $(E \prec C)$
$\mid$ while $B\ C'$	$(B \prec C,\ C' \prec C)$
$\mid$ if $B\ C'$	$(B \prec C,\ C' \prec C)$
$\mid$ if $B\ C'$ else C''	$(B \prec C,\ C' \prec C,\ C'' \prec C)$
$\mid$ { $C_1 \ldots C_n$ }, $(n \geq 0)$	$(C_1 \prec C, \ldots, C_n \prec C)$
$P ::= D\ C$	program $P \in \mathbb{P}$ $(C \prec P)$

Concrete Semantic Domain of Programs

Reachability properties:

$$\Sigma[\![D\ C]\!] \stackrel{\text{def}}{=} \Sigma[\![D]\!]$$
$$\Sigma[\![T\ X\,;]\!] \stackrel{\text{def}}{=} \{X\} \mapsto T$$
$$\Sigma[\![T\ X\,;\ D]\!] \stackrel{\text{def}}{=} (\{X\} \mapsto T) \cup \Sigma[\![D]\!]$$

states ρ ($\rho(X)$ is the value of X)

$$\mathcal{D}[\![P]\!] \stackrel{\text{def}}{=} \wp(\Sigma[\![P]\!])$$

sets of states/program properties where $\subseteq$ is implication, $\emptyset$ is false, $\cup$ is disjunction.

Concrete Reachability Semantics of Programs

$$\mathcal{S}[\![X = E;]\!]R \stackrel{\text{def}}{=} \{\rho[X \leftarrow \mathcal{E}[\![E]\!]\rho] \mid \rho \in R \cap \mathrm{dom}(E)\}$$
$$\rho[X \leftarrow v](X) \stackrel{\text{def}}{=} v, \qquad \rho[X \leftarrow v](Y) \stackrel{\text{def}}{=} \rho(Y)$$

$$\mathcal{S}[\![\texttt{if } B\ C']\!]R \stackrel{\text{def}}{=} \mathcal{S}[\![C']\!](\mathcal{B}[\![B]\!]R) \cup \mathcal{B}[\![\neg B]\!]R$$
$$\mathcal{B}[\![B]\!]R \stackrel{\text{def}}{=} \{\rho \in R \cap \mathrm{dom}(B) \mid B \text{ holds in } \rho\}$$
$$\mathcal{S}[\![\texttt{if } B\ C' \texttt{ else } C'']\!]R \stackrel{\text{def}}{=} \mathcal{S}[\![C']\!](\mathcal{B}[\![B]\!]R) \cup \mathcal{S}[\![C'']\!](\mathcal{B}[\![\neg B]\!]R)$$
$$\mathcal{S}[\![\texttt{while } B\ C']\!]R \stackrel{\text{def}}{=} \text{let } \mathcal{W} = \mathsf{lfp}^{\subseteq}_{\emptyset}\ \lambda\mathcal{X}\cdot R \cup \mathcal{S}[\![C']\!](\mathcal{B}[\![B]\!]\mathcal{X}) \text{ in } (\mathcal{B}[\![\neg B]\!]\mathcal{W})$$

$$\mathcal{S}[\![\{\}]\!]R \stackrel{\text{def}}{=} R$$
$$\mathcal{S}[\![\{C_1 \ldots C_n\}]\!]R \stackrel{\text{def}}{=} \mathcal{S}[\![C_n]\!] \circ \ldots \circ \mathcal{S}[\![C_1]\!] \quad n > 0$$

$$\mathcal{S}[\![D\ C]\!]R \stackrel{\text{def}}{=} \mathcal{S}[\![C]\!](\Sigma[\![D]\!]) \quad \text{(uninitialized variables)}$$

Not computable (undecidability).

Example: Abstract Semantic Domain of Programs

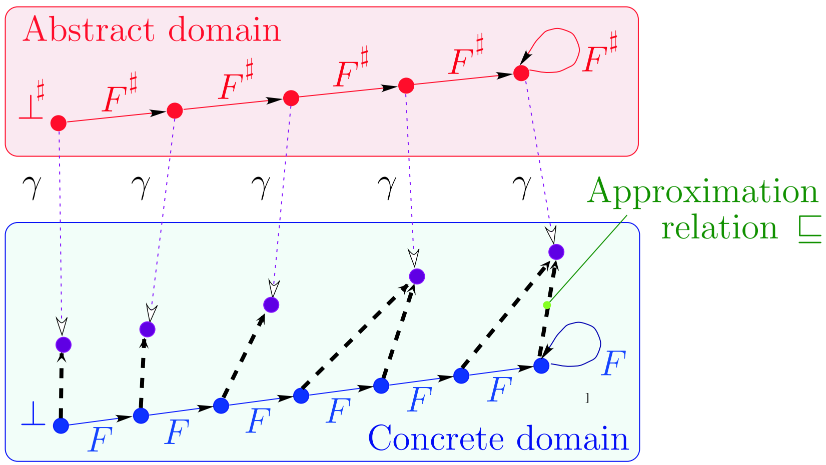
$$\langle \mathcal{D}^\sharp\llbracket P \rrbracket,\ \sqsubseteq,\ \bot,\ \sqcup \rangle$$

such that:

$$\langle \mathcal{D},\ \subseteq \rangle \xrightleftharpoons[\alpha]{\gamma} \langle \mathcal{D}^\sharp\llbracket P \rrbracket,\ \sqsubseteq \rangle$$

hence $\langle \mathcal{D}^\sharp\llbracket P \rrbracket,\ \sqsubseteq,\ \bot,\ \sqcup \rangle$ is a complete lattice such that $\bot = \alpha(\emptyset)$ and $\sqcup X = \alpha(\cup\, \gamma(X))$

Approximate Fixpoint Abstraction

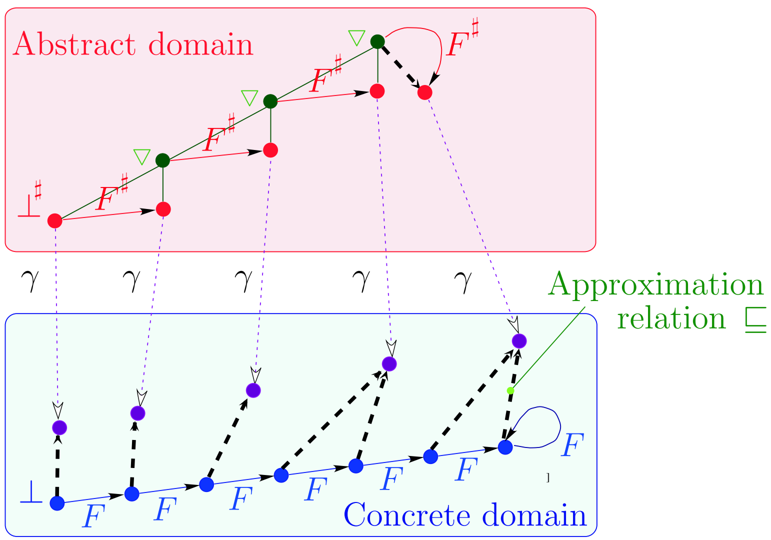


$$F \circ \gamma \sqsubseteq \gamma \circ F^\sharp \ \Rightarrow\ \mathsf{lfp}\, F \sqsubseteq \gamma(\mathsf{lfp}\, F^\sharp)$$

Abstract Reachability Semantics of Programs

$$\begin{aligned}
\mathcal{S}^\sharp\llbracket X = E; \rrbracket R &\stackrel{\text{def}}{=} \alpha(\{\rho[X \leftarrow \mathcal{E}\llbracket E \rrbracket\rho] \mid \rho \in \gamma(R) \cap \mathrm{dom}(E)\})\\
\mathcal{S}^\sharp\llbracket \texttt{if } B\ C' \rrbracket R &\stackrel{\text{def}}{=} \mathcal{S}^\sharp\llbracket C' \rrbracket(\mathcal{B}^\sharp\llbracket B \rrbracket R) \sqcup \mathcal{B}^\sharp\llbracket \neg B \rrbracket R\\
\mathcal{B}^\sharp\llbracket B \rrbracket R &\stackrel{\text{def}}{=} \alpha(\{\rho \in \gamma(R) \cap \mathrm{dom}(B) \mid B \text{ holds in } \rho\})\\
\mathcal{S}^\sharp\llbracket \texttt{if } B\ C' \texttt{ else } C'' \rrbracket R &\stackrel{\text{def}}{=} \mathcal{S}^\sharp\llbracket C' \rrbracket(\mathcal{B}^\sharp\llbracket B \rrbracket R) \sqcup \mathcal{S}^\sharp\llbracket C'' \rrbracket(\mathcal{B}^\sharp\llbracket \neg B \rrbracket R)\\
\mathcal{S}^\sharp\llbracket \texttt{while } B\ C' \rrbracket R &\stackrel{\text{def}}{=} \text{let } \mathcal{W} = \mathsf{lfp}^{\sqsubseteq}_{\bot}\, \lambda \mathcal{X} \cdot R \sqcup \mathcal{S}^\sharp\llbracket C' \rrbracket(\mathcal{B}^\sharp\llbracket B \rrbracket \mathcal{X})\\
&\qquad \text{in } (\mathcal{B}^\sharp\llbracket \neg B \rrbracket \mathcal{W})\\
\mathcal{S}^\sharp\llbracket \{\} \rrbracket R &\stackrel{\text{def}}{=} R\\
\mathcal{S}^\sharp\llbracket \{C_1 \ldots C_n\} \rrbracket R &\stackrel{\text{def}}{=} \mathcal{S}^\sharp\llbracket C_n \rrbracket \circ \ldots \circ \mathcal{S}^\sharp\llbracket C_1 \rrbracket \quad n > 0\\
\mathcal{S}^\sharp\llbracket D\ C \rrbracket R &\stackrel{\text{def}}{=} \mathcal{S}^\sharp\llbracket C \rrbracket(\top) \quad \text{(uninitialized variables)}
\end{aligned}$$

Convergence Acceleration with Widening



Example: Abstract Semantics with Convergence Acceleration [1]

$$\begin{aligned}
\mathcal{S}^\sharp[\![X = E;]\!]R &\stackrel{\text{def}}{=} \alpha(\{\rho[X \leftarrow \mathcal{E}[\![E]\!]\rho] \mid \rho \in \gamma(R) \cap \mathrm{dom}(E)\})\\
\mathcal{S}^\sharp[\![\texttt{if } B\ C']\!]R &\stackrel{\text{def}}{=} \mathcal{S}^\sharp[\![C']\!](\mathcal{B}^\sharp[\![B]\!]R) \sqcup \mathcal{B}^\sharp[\![\neg B]\!]R\\
\mathcal{B}^\sharp[\![B]\!]R &\stackrel{\text{def}}{=} \alpha(\{\rho \in \gamma(R) \cap \mathrm{dom}(B) \mid B \text{ holds in } \rho\})\\
\mathcal{S}^\sharp[\![\texttt{if } B\ C' \texttt{ else } C'']\!]R &\stackrel{\text{def}}{=} \mathcal{S}^\sharp[\![C']\!](\mathcal{B}^\sharp[\![B]\!]R) \sqcup \mathcal{S}^\sharp[\![C'']\!](\mathcal{B}^\sharp[\![\neg B]\!]R)\\
\mathcal{S}^\sharp[\![\texttt{while } B\ C']\!]R &\stackrel{\text{def}}{=} \text{let } \mathcal{F}^\sharp = \lambda \mathcal{X} \cdot \text{let } \mathcal{Y} = R \sqcup \mathcal{S}^\sharp[\![C']\!](\mathcal{B}^\sharp[\![B]\!]\mathcal{X})\\
&\qquad\qquad \text{in if } \mathcal{Y} \sqsubseteq \mathcal{X} \text{ then } \mathcal{X} \text{ else } \mathcal{X} \mathbin{\nabla} \mathcal{Y}\\
&\quad \text{and } \mathcal{W} = \mathsf{lfp}_{\perp}^{\sqsubseteq} \mathcal{F}^\sharp \text{ in } (\mathcal{B}^\sharp[\![\neg B]\!]\mathcal{W})\\
\mathcal{S}^\sharp[\![\{\}]\!]R &\stackrel{\text{def}}{=} R\\
\mathcal{S}^\sharp[\![\{C_1 \ldots C_n\}]\!]R &\stackrel{\text{def}}{=} \mathcal{S}^\sharp[\![C_n]\!] \circ \ldots \circ \mathcal{S}^\sharp[\![C_1]\!] \quad n > 0\\
\mathcal{S}^\sharp[\![D\ C]\!]R &\stackrel{\text{def}}{=} \mathcal{S}^\sharp[\![C]\!](\top) \quad \text{(uninitialized variables)}
\end{aligned}$$

Applications of Abstract Interpretation

[1] Note: $\mathcal{F}^\sharp$ not monotonic!

Applications of Abstract Interpretation

- **Static Program Analysis** [POPL '77], [POPL '78], [POPL '79] including **Dataflow Analysis** [POPL '79], [POPL '00], **Set-based Analysis** [FPCA '95], **Predicate Abstraction** [Manna's festschrift '03], ...
- **Syntax Analysis** [TCS 290(1) 2002]
- **Hierarchies of Semantics (including Proofs)** [POPL '92], [TCS 277(1–2) 2002]
- **Typing & Type Inference** [POPL '97]

Applications of Abstract Interpretation (Cont'd)

- **(Abstract) Model Checking** [POPL '00]
- **Program Transformation** [POPL '02]
- **Software Watermarking** [POPL '04]
- **Bisimulations** [RT-ESOP '04]

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

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

Reference

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

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

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

www.astree.ens.fr

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

- without
  - `union`
  - dynamic memory allocation
  - recursive function calls
  - backward branching
  - conflict side effects
  - C libraries
- Application Domain: safety critical embedded real-time synchronous software for non-linear control of very complex control/command systems.

Concrete Operational Semantics

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

Abstract Semantics

- Reachable states for the concrete operational semantics
- Volatile environment is specified by a *trusted* configuration file.

Implicit Specification: Absence of Runtime Errors

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

Example application

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


- C program, automatically generated from a proprietary high-level specification (à la Simulink/SCADE)
- A340 family: 132,000 lines, 75,000 LOCs after preprocessing, 10,000 global variables, over 21,000 after expansion of small arrays
- A380: $\times$ 3

The Class of Considered Periodic Synchronous Programs

```
declare volatile input, state and output variables;
initialize state and output variables;
loop forever
    - read volatile input variables,
    - compute output and state variables,
    - write to volatile output variables;
  wait_for_clock ();
end loop
```

- Requirements: the only interrupts are clock ticks;
- Execution time of loop body less than a clock tick [3].

Reference

[3] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and precise WCET determination for a real-life processor. *ESOP (2001)*, LNCS 2211, 469–485.

Characteristics of the ASTRÉE Analyzer

Static: compile time analysis ($\neq$ run time analysis Rational Purify, Parasoft Insure++)

Program Analyzer: analyzes programs not micromodels of programs ($\neq$ PROMELA in SPIN or Alloy in the Alloy Analyzer)

Automatic: no end-user intervention needed ($\neq$ ESC Java, ESC Java 2)

Sound: covers the whole state space ($\neq$ MAGIC, CBMC) so never omit potential errors ($\neq$ UNO, CMC from coverity.com) or sort most probable ones ($\neq$ Splint)

Characteristics of the ASTRÉE Analyzer (Cont'd)

Multiabstraction: uses many numerical/symbolic abstract domains ($\neq$ symbolic constraints in Bane or the canonical abstraction of TVLA)

Infinitary: all abstractions use infinite abstract domains with widening/narrowing ($\neq$ model checking based analyzers such as VeriSoft, Bandera, Java PathFinder)

Efficient: always terminate ($\neq$ counterexample-driven automatic abstraction refinement BLAST, SLAM)

Characteristics of the ASTRÉE Analyzer (Cont'd)

Specializable: can easily incorporate new abstractions (and reduction with already existing abstract domains) ($\neq$ general-purpose analyzers PolySpace Verifier)

Domain-Aware: knows about control/command (e.g. digital filters) (as opposed to specialization to a mere programming style in C Global Surveyor)

Parametric: the precision/cost can be tailored to user needs by options and directives in the code

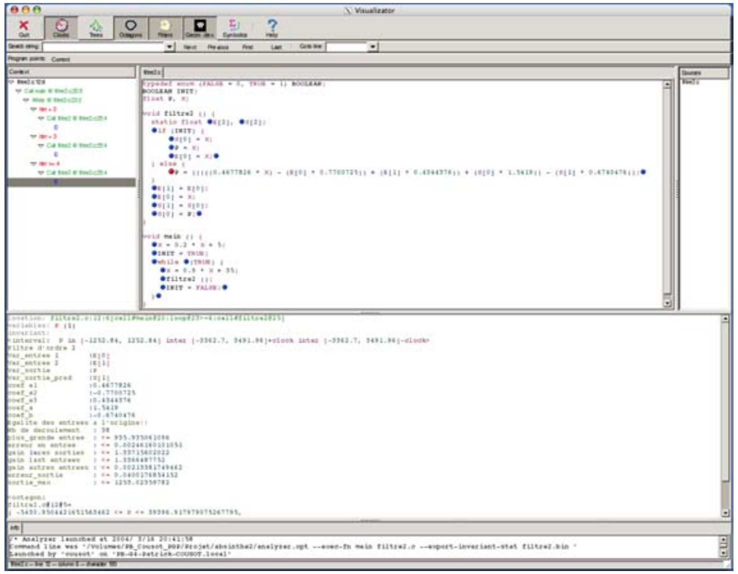
Characteristics of the ASTRÉE Analyzer (Cont'd)

Automatic Parametrization: the generation of parametric directives in the code can be programmed (to be specialized for a specific application domain)

Modular: an analyzer instance is built by selection of O-CAML modules from a collection each implementing an abstract domain

Precise: very few or no false alarm when adapted to an application domain $\longrightarrow$ it is a VERIFIER!

Example of Analysis Session



Benchmarks (Airbus A340 Primary Flight Control Software)

- 132,000 lines, 75,000 LOCs after preprocessing
- Comparative results (commercial software):

  4,200 (false?) alarms,

  3.5 days;
- Our results:

  0 alarms,

  40mn on 2.8 GHz PC,

  300 Megabytes

  ⟶ A world première!

(Airbus A380 Primary Flight Control Software)

- 350,000 lines
- 0 alarms (last week!),

  7h [2] on 2.8 GHz PC,

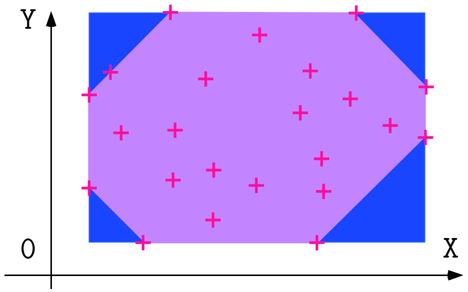
  1 Gigabyte

  ⟶ A world grand première!

Examples of Abstractions

[2] We are still in a phase where we favour precision rather than computation costs, and this should go down. For example, the A340 analysis went up to 5 h, before being reduced by requiring less precision while still getting no false alarm.

General-Purpose Abstract Domains: Intervals and Octagons



Intervals:

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

Octagons [4]:

$$\begin{cases} 1 \leq x \leq 9 \\ x + y \leq 77 \\ 1 \leq y \leq 20 \\ x - y \leq 04 \end{cases}$$

Difficulties: many global variables, arrays (smashed or not), IEEE 754 floating-point arithmetic (in program <u>and</u> analyzer) [5]

Reference

[4] A. Miné. A New Numerical Abstract Domain Based on Difference-Bound Matrices. In *PADO'2001*, LNCS 2053, Springer, 2001, pp. 155–172.

[5] A. Miné. Relational abstract domains for the detection of floating-point run-time errors. In *ESOP'04*, Barcelona, LNCS 2986, pp. 1—17, Springer, 2004.

Floating-Point Computations

- Code Sample:

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

```
/* 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
```

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

Symbolic abstract domain

- Interval analysis: if $x \in [a, b]$ and $y \in [c, d]$ then $x - y \in [a - c, b - d]$ 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)>
```


Clock Abstract Domain for Counters

- Code Sample:

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

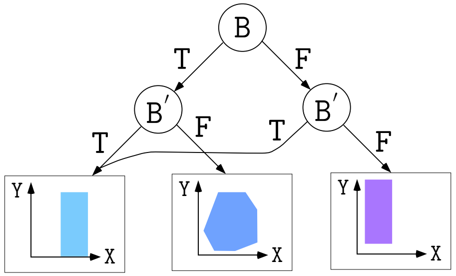
- Output `T` is true iff the volatile input `I` has been true for the last `n` clock ticks.
- The clock ticks every `s` seconds for at most `h` hours, thus `R` is bounded.
- To prove that `R` cannot overflow, we must prove that `R` cannot exceed the elapsed clock ticks *(impossible using only intervals)*.

- Solution:
  - We add a phantom variable `clock` in the concrete user semantics to track elapsed clock ticks.
  - For each variable `X`, we abstract three intervals: `X`, `X+clock`, and `X-clock`.
  - If `X+clock` or `X-clock` is bounded, so is `X`.

Boolean Relations for Boolean Control

- Code Sample:

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



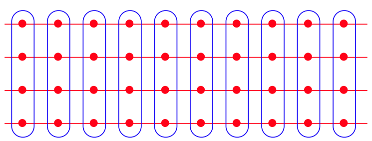
The boolean relation abstract domain is parameterized by the height of the decision tree (an analyzer option) and the abstract domain at the leafs

Control Partitionning for Case Analysis

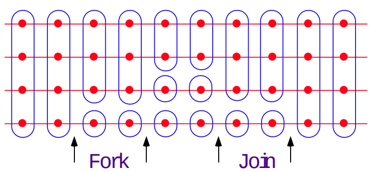
- Code Sample:

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

  ... found invariant -100 ≤ x ≤ 100 ...

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

Control point partitionning:



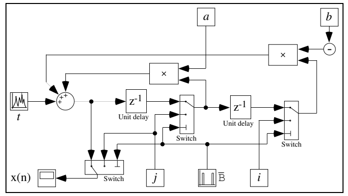
Trace partitionning:



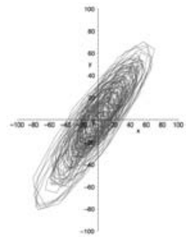
Delaying abstract unions in tests and loops is more precise for non-distributive abstract domains (and much less expensive than disjunctive completion).

Ellipsoid Abstract Domain for Filters

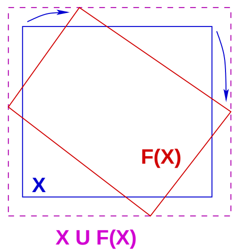
2d Order Digital Filter:



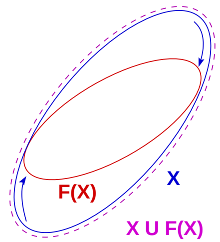
- Computes $X_n = \begin{cases} \alpha X_{n-1} + \beta X_{n-2} + Y_n \\ I_n \end{cases}$
- The concrete computation is bounded, which must be proved in the abstract.
- There is no stable interval or octagon.
- The simplest stable surface is an ellipsoid.



execution trace



unstable interval



stable ellipsoid

Filter Example [6]

```
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; }
}
```

Reference

[6] J. Feret. Static analysis of digital filters. In *ESOP'04*, Barcelona, LNCS 2986, pp. 33—48, Springer, 2004.

(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).

The main loop invariant for the A340

A textual file over 4.5 Mb with

- 6,900 boolean interval assertions ($x \in [0;1]$)
- 9,600 interval assertions ($x \in [a;b]$)
- 25,400 clock assertions ($x+\mathsf{clk} \in [a;b] \wedge x-\mathsf{clk} \in [a;b]$)
- 19,100 additive octagonal assertions ($a \leq x+y \leq b$)
- 19,200 subtractive octagonal assertions ($a \leq x-y \leq b$)
- 100 decision trees
- 60 ellipse invariants, etc ...

involving over 16,000 floating point constants (only 550 appearing in the program text) $\times$ 75,000 LOCs.

Possible origins of imprecision and how to fix it

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

- Abstract transformers (not best possible) $\longrightarrow$ improve algorithm;
- Automatized parametrization (e.g. variable packing) $\longrightarrow$ improve pattern-matched program schemata;
- Iteration strategy for fixpoints $\longrightarrow$ fix widening [3];
- Inexpressivity i.e. indispensable local inductive invariant are inexpressible in the abstract $\longrightarrow$ add a new abstract domain to the reduced product (e.g. filters).

Conclusion

[3] 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.

Conclusion

- Most applications of abstract interpretation tolerate a small rate (typically 5 to 15%) of false alarms:
  - Program transformation → do not optimize,
  - Typing → reject some correct programs, etc,
  - WCET analysis → overestimate;
- Some applications require no false alarm at all:
  - Program verification.
- Theoretically possible [SARA '00], practically feasible [PLDI '03]

Reference

[SARA '00] P. Cousot. Partial Completeness of Abstract Fixpoint Checking, invited paper. In *4th Int. Symp. SARA '2000*, LNAI 1864, Springer, pp. 1–25, 2000.

[PLDI '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. PLDI'03, San Diego, June 7–14, ACM Press, 2003.

The Future & Grand Challenges

Future (5 years):

- Asynchronous concurrency (for less critical software)
- Functional properties (reactivity)
- Industrialization

Grand challenge:

- Verification from specifications to machine code (verifying compiler)
- Verification of systems (quasi-synchrony, distribution)

THE END, THANK YOU

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

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 *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY, USA.

[PACJM '79] P. Cousot and R. Cousot. Constructive versions of Tarski's fixed point theorems. Pacific Journal of Mathematics 82(1):43–57 (1979).

[POPL '78] 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*, pages 84–97, Tucson, Arizona, 1978. ACM Press, New York, NY, U.S.A.

[POPL '79] 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*, pages 269–282, San Antonio, Texas, 1979. ACM Press, New York, NY, U.S.A.

[POPL '92] P. Cousot and R. Cousot. Inductive Definitions, Semantics and Abstract Interpretation. In Conference Record of the 19th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Programming Languages, pages 83–94, Albuquerque, New Mexico, 1992. ACM Press, New York, U.S.A.

[FPCA '95] P. Cousot and R. Cousot. Formal Language, Grammar and Set-Constraint-Based Program Analysis by Abstract Interpretation. In *SIGPLAN/SIGARCH/WG2.8 7th Conference on Functional Programming and Computer Architecture, FPCA'95*. La Jolla, California, U.S.A., pages 170–181. ACM Press, New York, U.S.A., 25-28 June 1995.

[POPL '97] P. Cousot. Types as Abstract Interpretations. In Conference Record of the 24th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Programming Languages, pages 316–331, Paris, France, 1997. ACM Press, New York, U.S.A.

[POPL '00] P. Cousot and R. Cousot. Temporal abstract interpretation. In *Conference Record of the Twentyseventh Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 12–25, Boston, Mass., January 2000. ACM Press, New York, NY.

[POPL '02] P. Cousot and R. Cousot. Systematic Design of Program Transformation Frameworks by Abstract Interpretation. In *Conference Record of the Twentyninth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 178–190, Portland, Oregon, January 2002. ACM Press, New York, NY.

[TCS 277(1–2) 2002] P. Cousot. Constructive Design of a Hierarchy of Semantics of a Transition System by Abstract Interpretation. *Theoretical Computer Science* 277(1–2):47–103, 2002.

[TCS 290(1) 2002] P. Cousot and R. Cousot. Parsing as abstract interpretation of grammar semantics. *Theoret. Comput. Sci.*, 290:531–544, 2003.

[Manna's festschrift '03] P. Cousot. Verification by Abstract Interpretation. *Proc. Int. Symp. on Verification – Theory & Practice – Honoring Zohar Manna's 64th Birthday*, N. Dershowitz (Ed.), Taormina, Italy, June 29 – July 4, 2003. Lecture Notes in Computer Science, vol. 2772, pp. 243–268. © Springer-Verlag, Berlin, Germany, 2003.

[POPL '04] 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*, pages 173–185, Venice, Italy, January 14-16, 2004. ACM Press, New York, NY.

[RT-ESOP '04] F. Ranzato and F. Tapparo. Strong Preservation as Completeness in Abstract Interpretation. Proc. Programming Languages and Systems, 13th European Symposium on Programming, ESOP 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, D.A. Schmidt (Ed), Lecture Notes in Computer Science 2986, Springer, 2004, pp. 18–32.