**Application of Abstract Interpretation to the Static Verification of Safety Critical Code**

Patrick Cousot  
École normale supérieure, Paris  
cousot@ens.fr  www.di.ens.fr/~cousot

IBM Research Seminar, Thomas J. Watson Research Center,  
Hawthorne, NY, January 20, 2006

---

**Talk Outline**

- Motivation (1 mn) ............................................. 3  
- Abstract interpretation, reminder (10 mn) ............... 6  
- Applications of abstract interpretation (2 mn) ........ 21  
- A practical application to the ASTRÉE static analyzer (15 mn) 24  
- Examples of abstractions in ASTRÉE (15 mn) ............ 40  
- Conclusion (2 mn) ............................................... 56

---

**Motivation**

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

---

**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.
Static Analysis by Abstract Interpretation

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

Undecidability →

**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.

---

Syntax of programs

\( X \)

variables \( X \in X \)

\( T \)

types \( T \in T \)

\( E \)

arithmetic expressions \( E \in E \)

\( B \)

boolean expressions \( B \in B \)

\( D ::= T \ X \)

\( C ::= X = E \)  \( \mid \) commands \( C \in C \)

\( \mid \) while \( B \ C' \)  \( \mid \) if \( B \ C' \) else \( C'' \)

\( \mid \) \{ \( C_1 \ldots C_n \), \( (n \geq 0) \) \}

\( P ::= D \ C \)  \( \mid \) program \( P \in P \)

---

Postcondition semantics

![Postcondition semantics diagram](image)
States

Values of given type:

\( \forall[T] : \) values of type \( T \in T \)

\( \forall[\text{int}] \equiv \{ z \in \mathbb{Z} | \min_{\text{int}} \leq z \leq \max_{\text{int}} \} \)

Program states \( \Sigma[P] \):

\[ \begin{align*}
\Sigma[D \ C] & \equiv \Sigma[D] \\
\Sigma[T \ X ;] & \equiv \{ X \} \mapsto \forall[T] \\
\Sigma[T \ X ; D] & \equiv (\{ X \} \mapsto \forall[T]) \cup \Sigma[D]
\end{align*} \]

Concrete Semantic Domain of Programs

Concrete semantic domain for reachability properties:

\( \mathcal{D}[P] \equiv \rho(\Sigma[P]) \)

sets of states

i.e. program properties where \( \subseteq \) is implication, \( \emptyset \) is false, \( \cup \) is disjunction.

Abstract Semantic Domain of Programs

\( \langle \mathcal{D}^\#[P], \subseteq, \bot, \cup \rangle \)

such that:

\[ \langle \mathcal{D}[P], \subseteq \rangle \xrightarrow{\gamma \alpha} \langle \mathcal{D}^\#[P], \subseteq \rangle \]

i.e.

\[ \forall X \in \mathcal{D}[P], Y \in \mathcal{D}^\#[P] : \alpha(X) \subseteq Y \iff X \subseteq \gamma(Y) \]

hence \( \langle \mathcal{D}^\#[P], \subseteq, \bot, \cup \rangle \) is a complete lattice such that

\( \bot = \alpha(0) \) and \( \cup X = \alpha(\cup \gamma(X)) \)
Example 1 of Abstraction

**Traces:** set of finite or infinite maximal sequences of states for the operational transition semantics

![Diagram]

**Strongest liberal postcondition:** final states \( s \) reachable from a given precondition \( P \)

\[ \alpha(X) = \lambda P \cdot \{ s \mid \exists \sigma_0 \sigma_1 \ldots \sigma_n \in X : \sigma_0 \in P \land s = \sigma_n \} \]

We have \((\Sigma, \subseteq)\) pointwise:

\[ \langle \rho(\Sigma^\infty), \subseteq \rangle \leftrightarrow_{\alpha} \langle \rho(\Sigma) \cup \rho(\Sigma), \subseteq \rangle \]

Example 2 of Abstraction

**Traces:** set of finite or infinite maximal sequences of states for the operational transition semantics

![Diagram]

**Set of reachable states:** set of states appearing at least once along one of these traces (global invariant)

\[ \alpha_1(X) = \{ \sigma_i \mid \sigma \in X \land 0 \leq i < |\sigma| \} \]

**Partitioned set of reachable states:** project along each control point (local invariant)

\[ \alpha_2(\{ \langle c_i, \rho_i \rangle \mid i \in \Delta \}) = \lambda c. \{ \rho_i \mid i \in \Delta \land c = c_i \} \]

Example 3: Reduced Product of Abstract Domains

To combine abstractions

\[ \langle D, \subseteq \rangle \leftarrow_{\alpha_1} \langle D_1, \subseteq_1 \rangle \text{ and } \langle D, \subseteq \rangle \leftarrow_{\alpha_2} \langle D_2, \subseteq_2 \rangle \]

the reduced product is

\[ \alpha(X) \overset{\text{def}}{=} \cap \{ \langle x, y \rangle \mid X \subseteq \gamma_1(x) \land X \subseteq \gamma_2(y) \} \]

such that \( \subseteq \overset{\text{def}}{=} \subseteq_1 \times \subseteq_2 \) and

\[ \langle D_1, \subseteq_1 \rangle \leftarrow_{\gamma_1 \times \gamma_2} \langle \alpha(D), \subseteq \rangle \]

Example: \( x \in [1, 9] \land x \mod 2 = 0 \) reduces to \( x \in [2, 8] \land x \mod 2 = 0 \)

---

\( ^2 \) assuming these values to be totally ordered.

---

\( \alpha_3 \) Partitionned cartesian set of reachable states: project along each program variable (relationships between variables are now lost)

\[ \alpha_3(\lambda c. \{ \rho_i \mid i \in \Delta_c \}) = \lambda c. \lambda x. \{ \rho_i(x) \mid i \in \Delta_c \} \]

\( \alpha_4 \) Partitionned cartesian interval of reachable states: take min and max of the values of the variables

\[ \alpha_4(\lambda c. \lambda x. \{ v_i \mid i \in \Delta_c \}) = \lambda c. \lambda x. \langle \min \{ v_i \mid i \in \Delta_c \}, \max \{ v_i \mid i \in \Delta_c \} \rangle \]

\( \alpha_1, \alpha_2, \alpha_3 \text{ and } \alpha_4 \), whence \( \alpha_4 \circ \alpha_3 \circ \alpha_2 \circ \alpha_1 \) are lower-adjoints of Galois connections
**Abstract Reachability Semantics of Programs**

\[ S^\perp[X = E;] R \triangleq \alpha(\{\rho|X \leftarrow E[B] \rho | \rho \in \gamma(R) \cap \text{dom}(E)\}) \]

\[ S^\perp[\text{if } B \ C'] R \triangleq S^\perp[C'](B^\perp[B] R) \cup B^\perp[-B] R \]

\[ B^\perp[B] R \triangleq \alpha(\{\rho | \rho \in \gamma(R) \cap \text{dom}(B) \mid B \text{ holds in } \rho\}) \]

\[ S^\perp[\text{if } B \ C' \text{ else } C''] R \triangleq S^\perp[C'](B^\perp[B] R) \cup S^\perp[C''](B^\perp[-B] R) \]

\[ S^\perp[\text{while } B \ C'] R \triangleq \text{let } W = \text{lfp}_{\gamma} \lambda \mathcal{X}.R \cup S^\perp[C'](B^\perp[B] \mathcal{X}) \]

\[ \text{in } (B^\perp[-B] W) \]

\[ S^\perp[\{\}] R \triangleq R \]

\[ S^\perp[\{C_1 \ldots C_n\}] R \triangleq S^\perp[C_n] \circ \ldots \circ S^\perp[C_1] \quad n > 0 \]

\[ S^\perp[D \ C'] R \triangleq S^\perp[C](\top) \quad (\text{uninitialized variables}) \]

---

**Abstract Semantics with Convergence Acceleration**

\[ S^\perp[X = E;] R \triangleq \alpha(\{\rho|X \leftarrow E[B] \rho | \rho \in \gamma(R) \cap \text{dom}(E)\}) \]

\[ S^\perp[\text{if } B \ C'] R \triangleq S^\perp[C'](B^\perp[B] R) \cup B^\perp[-B] R \]

\[ B^\perp[B] R \triangleq \alpha(\{\rho | \rho \in \gamma(R) \cap \text{dom}(B) \mid B \text{ holds in } \rho\}) \]

\[ S^\perp[\text{if } B \ C' \text{ else } C''] R \triangleq S^\perp[C'](B^\perp[B] R) \cup S^\perp[C''](B^\perp[-B] R) \]

\[ S^\perp[\text{while } B \ C'] R \triangleq \text{let } \mathcal{F}^\perp = \lambda \mathcal{X}.\text{let } \mathcal{Y} = R \cup S^\perp[C'](B^\perp[B] \mathcal{X}) \]

\[ \text{in if } \mathcal{Y} \subseteq \lambda \alpha \text{ then } \mathcal{X} \text{ else } \mathcal{X} \uplus \mathcal{Y} \text{ and } W = \text{lfp}_{\gamma} \mathcal{F}^\perp \text{ in } (B^\perp[-B] W) \]

\[ S^\perp[\{\}] R \triangleq R \]

\[ S^\perp[\{C_1 \ldots C_n\}] R \triangleq S^\perp[C_n] \circ \ldots \circ S^\perp[C_1] \quad n > 0 \]

\[ S^\perp[D \ C'] R \triangleq S^\perp[C](\top) \quad (\text{uninitialized variables}) \]

---

**Note:** \( F^\perp \) not monotonic
Applications of Abstract Interpretation

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 ASTRÉE Static Analyzer

Reference
Programs analysed by ASTRÉE

- **Application Domain:** large safety critical embedded real-time synchronous software for non-linear control of very complex control/command systems.
- **C programs:**
  - with
    - basic numeric datatypes, structures and arrays
    - pointers (including on functions),
    - floating point computations
    - tests, loops and function calls
    - limited branching (forward goto, break, continue)

---

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, execution stops on first runtime error\(^4\))

---

Abstract Semantics

- **without**
  - union
  - dynamic memory allocation
  - recursive function calls
  - backward branching
  - conflicting side effects
- **C libraries, system calls (parallelism)**

---

**Reachable states** for the concrete trace operational semantics

**Volatile environment** is specified by a trusted configuration file.

**Requirements:**

- **Soundness:** absolutely essential
- **Precision:** few or no false alarm\(^5\) (full certification)
- **Efficiency:** rapid analyses and fixes during development

\(^4\) semantics of C unclear after an error, equivalent if no alarm

\(^5\) Potential runtime error signaled by the analyser due to overapproximation but impossible in any actual program run.
Implicit Specification: Absence of Runtime Errors

- No violation of the norm of C (e.g. array index out of bounds, division by zero)
- No implementation-specific undefined behaviors (e.g. maximum short integer is 32767, NaN)
- 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 family/A380 fly-by-wire system

  ![Airbus A340](image)

- 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: × 3

Challenging aspects

- Size: > 100 kLOC, > 10,000 variables
- Floating point computations
  including interconnected networks of filters, non linear control with feedback, interpolations...
- Interdependencies among variables:
  - Stability of computations should be established
  - Complex relations should be inferred among numerical and boolean data
  - Very long data paths from input to outputs
Characteristics of the ASTRÉE Analyzer

**Static**: compile time analysis (≠ run time analysis Rational Purify, Parasoft Insure++)

**Program Analyzer**: analyzes programs not micromodels of programs (≠ PROMELA in SPIN or Alloy in the Alloy Analyzer)

**Automatic**: no end-user intervention needed (≠ ESC Java, ESC Java 2)

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

---

Characteristics of the ASTRÉE Analyzer (Cont’d)

**Specializable**: can easily incorporate new abstractions (and reduction with already existing abstract domains) (≠ 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)

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

**Infinite**: all abstractions use infinite abstract domains with widening/narrowing (≠ model checking based analyzers such as VeriSoft, Bandera, Java PathFinder)

**Efficient**: always terminate (≠ counterexample-driven automatic abstraction refinement BLAST, SLAM)

---

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 OCAML modules from a collection each implementing an abstract domain

**Precise**: very few or no false alarm when adapted to an application domain → 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 (Nov. 2004),
  7h 6 on 2.8 GHz PC,
  1 Gigabyte
  ➔ A world grand première!

Examples of Abstractions

6 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:
\[
\begin{align*}
1 \leq x & \leq 9 \\
1 \leq y & \leq 20
\end{align*}
\]
Octagons [10]:
\[
\begin{align*}
1 \leq x & \leq 9 \\
x + y & \leq 77 \\
1 \leq y & \leq 20 \\
x - y & \leq 04
\end{align*}
\]

Difficulties: many global variables, arrays (smashed or not), IEEE 754 floating-point arithmetic (in program and analyzer) [POPL ’77, 10, 11]

Floating-Point Computations

```c
/* float-error.c */
int main () {
    float x, y, z, r;
    x = 1.0000000199e+38;
    y = x + 1.0e21;
    z = x - 1.0e21;
    r = y - z;
    printf("%f\n", r);
}
```

```c
/* double-error.c */
int main () {
    double x, float y, z, r;
    x = 1.0000000199e+38;
    y = x + 1.0e21;
    z = x - 1.0e21;
    r = y - z;
    printf("%f\n", r);
}
```

\[(x + a) - (x - a) \neq 2a\]

Explanation of the huge rounding error

1. Floats
   - Rounding

2. Doubles
   - Rounding
Floating-point linearization [11, 12]

- Approximate arbitrary expressions in the form
  \[a_0, b_0] + \sum_k ([a_k, b_k] \times V_k)\]

- Example:
  \[Z = X - (0.25 \times X)\] is linearized as
  \[Z = [(0.749 \cdots, 0.750 \cdots] \times X + (2.35 \cdots 10^{-38} \times [-1, 1])\]

- Allows simplification even in the interval domain
  if \(X < [-1, 1]\), we get \(|Z| \leq 0.750\cdots\) instead of \(|Z| \leq 1.25\cdots\)

- Allows using a relational abstract domain (octagons)

- Example of good compromise between cost and precision

Symbolic abstract domain [11, 12]

- 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);

\[
\begin{align*}
\text{void main (0 \{ \text{ int } X, Y; \\
2 \quad \text{ASTREE}_\text{linear_fact}((0 = X) \&\& (X = 100)); \\
3 \quad Y = (X - 0); \\
4 \quad \text{ASTREE}_\text{log_sqrt}(Y)); \\
5 \}}
\end{align*}
\]

Control Partitionning for Case Analysis

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

\[
\begin{align*}
\text{void main () \{ \\
1 \quad \text{float t}[5] = \{4.0, 0.0, 0.0, 10.0, 10.0\}; \\
2 \quad \text{float c}[5] = \{0.0, 1.0, 2.0, 0.0, 0.0\}; \\
3 \quad \text{float d}[5] = \{4.0, 0.0, 0.0, 20.0, 0.0\}; \\
4 \quad \text{float } x, \; r; \\
5 \quad \text{int } i = 0; \\
6 \quad \ldots \text{ found invariant } -100 \leq x \leq 100 \ldots \\
7 \quad \text{while } (i < 3) \&\& (x > t[3] + c[3]) \{ \\
8 \quad \quad i = i + 1; \\
9 \quad \quad x = (x - t[3]) * c[3] + d[3]; \\
10 \quad \}
\end{align*}
\]
Ellipsoid Abstract Domain for Filters

- Computes $X_n = \{ \alpha X_{n-1} + \beta X_{n-2} + Y_n \}$
- 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.

Arithmetic-geometric progressions\(^7\) [8]

- Abstract domain: $(\mathbb{R}^+)^5$
- Concretization:
  $$\gamma \in (\mathbb{R}^+)^5 \mapsto \varphi(N \mapsto \mathbb{R})$$
  $$\gamma(M, a, b, a', b') =$$
  $$\{ f \mid \forall k \in \mathbb{N} : |f(k)| \leq \left( \lambda x.a x + b \circ (\lambda x.a' x + b')^k \right)(M) \}$$

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

\(^7\) here in R
Arithmetic-geometric progressions (Example 2)

```c
void main()
{ FIRST = TRUE;
  while (TRUE) {
    dev();
    FIRST = FALSE;
    _ASTREE_wait_for_clock();
  }
}
```

(Automatic) Parameterization

- All abstract domains of ASTRÉE are parameterized, e.g.
  - variable packing for octagons 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 \( x + clk \in [a;b] \wedge x - clk \in [a;b] \)
- 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) \( \rightarrow \) improve algorithm;
- Automatized parametrization (e.g. variable packing) \( \rightarrow \) improve pattern-matched program schemata;
- Iteration strategy for fixpoints \( \rightarrow \) fix widening \(^8\);
- Inexpressivity i.e. indispensable local inductive invariant are inexpressible in the abstract \( \rightarrow \) add a new abstract domain to the reduced product (e.g. filters).

\(^8\) 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:
References

[2] www.mat.univie.ac.at [4, 5, 6, 7, 8, 9, 10, 11, 12]


