Automatic Verification of Avionic Synchronous Safety Critical Embedded Software

Patrick Cousot
Jerome C. Hunsaker Visiting Professor
Department of Aeronautics and Astronautics, MIT
cousot@mit.edu  www.mit.edu/~cousot

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

Programming Languages Seminar, College of Computer and Information Science, Northeastern University, Boston, June 1st, 2005

Motivation

All Computer Scientists Have Experienced Bugs

Ariane 5.01 failure  Patriot failure  Mars orbiter loss
(overflow)  (float rounding)  (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:** analyse 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**

\[
\begin{align*}
X & \quad \text{variables } X \in X \\
T & \quad \text{types } T \in T \\
E & \quad \text{arithmetic expressions } E \in E \\
B & \quad \text{boolean expressions } B \in B \\
D & ::= T X; \\
& \quad | \quad T X ; D' \\
C & ::= X = E; \\
& \quad | \quad \text{while } B C' \\
& \quad | \quad \text{if } B C' \text{ else } C'' \\
& \quad | \quad \{ \text{C}_1 \ldots \text{C}_n \}, (n \geq 0) \\
P & ::= D C \\
\end{align*}
\]

**Postcondition semantics**

\[
x(t)
\]

Possible trajectories

---

**References**


States

Values of given type:
\[ \mathcal{V}[T] : \text{values of type } T \in T \]
\[ \mathcal{V}_{\text{int}} \equiv \{ z \in \mathbb{Z} \mid \text{min}_\text{int} \leq z \leq \text{max}_\text{int}\} \]

Program states \( \Sigma[P] \)¹:
\[ \Sigma[D \; C] \equiv \Sigma[D] \]
\[ \Sigma[T \; X; \; ] \equiv \{ X \} \mapsto \mathcal{V}[T] \]
\[ \Sigma[T \; X; \; D] \equiv (\{ X \} \mapsto \mathcal{V}[T]) \cup \Sigma[D] \]

¹ States \( \rho \in \Sigma[P] \) of a program \( P \) map program variables \( X \) to their values \( \rho(X) \)

Concrete Reachability Semantics of Programs

\[
S[X = E] R \equiv \{ \rho[X \leftarrow \mathcal{E}[E]\rho] \mid \rho \in R \cap \text{dom}(E) \}
\]
\[ \rho[X \leftarrow v](X) \equiv v, \quad \rho[X \leftarrow v](Y) \equiv \rho(Y) \]
\[ S[\text{if } B \; C' \; \text{else } C''] R \equiv S[C'](B[B]R) \cup S[\neg B] R \]
\[ B[B] R \equiv \{ \rho \in R \cap \text{dom}(B) \mid B \text{ holds in } \rho \} \]
\[ S[\text{while } B \; C'] R \equiv \text{let } W = \text{lfp}_{\rho} \lambda \mathcal{X} : R \cup S[C']((B[B] \mathcal{X}) \in (\neg B)[W]) \]
\[ B[B] R \equiv \{ \rho \in R \cap \text{dom}(B) \mid B \text{ holds in } \rho \} \]
\[ S[\{} R \equiv R \]
\[ S[\{C_1 \ldots C_n\}] R \equiv S[C_n] \circ \ldots \circ S[C_1] \quad n > 0 \]
\[ S[D \; C] R \equiv S[C](\Sigma[D]) \quad \text{(uninitialized variables)} \]

Not computable (undecidability).

Abstract Semantic Domain of Programs

\[ \langle D^\#[P], \subseteq, \bot, \cup \rangle \]

such that:
\[ \langle D, \subseteq \rangle \xrightarrow{\gamma} \langle D^\#[P], \subseteq \rangle \]

hence \( \langle D^\#[P], \subseteq, \bot, \cup \rangle \) is a complete lattice such that \( \bot = \alpha(0) \) and \( \cup X = \alpha(\cup \gamma(X)) \)
Reduced Product of Abstract Domains

To combine abstractions
\[ <D_1, \subseteq> \xrightarrow{\gamma_1, \alpha_1} <D_1^\parallel, \subseteq_1> \text{ and } <D_2, \subseteq> \xrightarrow{\gamma_2, \alpha_2} <D_2^\parallel, \subseteq_2> \]
the reduced product is
\[ \alpha(X) \overset{\text{def}}{=} \bigcap \{ (x, y) \mid X \subseteq \gamma_1(X) \land X \subseteq \gamma_2(X) \} \]
such that \( \subseteq_1 \times \subseteq_2 \) and
\[ \langle D, \subseteq > \xrightarrow{\gamma_1 \times \gamma_2, \alpha} \langle \alpha(D), \subseteq > \]

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

Abstract Reachability Semantics of Programs

\[ S[[X = E]]R \overset{\text{def}}{=} \alpha\{\{\rho \mid X \leftarrow \mathcal{E}[E]\} \mid \rho \in \gamma(R) \land \text{dom}(E)\} \]
\[ S[[\text{if } B \text{ then } C \text{ else } C' ]]R \overset{\text{def}}{=} S[[C']]B \cup S[[\neg B]]R \]
\[ B[[\neg B]]R \overset{\text{def}}{=} \alpha\{\{\rho \mid \rho \in \gamma(R) \land \text{dom}(B) \land B \text{ holds in } \rho\} \}
\[ S[[\text{while } B \text{ do } C ]]R \overset{\text{def}}{=} \text{let } W = \text{lfp}_{\subseteq} \lambda X . R \cup S[[C']](B[[\neg B]]X) \text{ in } (B[[\neg B]]W) \]
\[ S[[\{C_1 \ldots C_n\}]R \overset{\text{def}}{=} S[[C_n]] \circ \ldots \circ S[[C_1]] \quad n > 0 \]
\[ S[[D \text{ do } C ]]R \overset{\text{def}}{=} S[[C]](\top) \quad \text{(uninitialized variables)} \]
Abstract Semantics with Convergence Acceleration

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

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

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

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

\[
S^\bullet[\text{while } B \text{ } C'\text{ } R] \triangleq \text{let } F^\bullet = \lambda X . \text{let } Y = R \cup S^\bullet[C'](B^\bullet[B]X)\text{ in if } Y \subseteq X \text{ then } X \triangledown Y \text{ and } W = \text{lfp}_{\geq} F^\bullet \text{ in } (B^\bullet[\neg B]W)
\]

\[
S^\bullet[\text{while } B \text{ } C'\text{ } R] \triangleq \text{let } Y = R \cup S^\bullet[C'](B^\bullet[B]X)\text{ in if } Y \subseteq X \text{ then } X \triangledown Y \text{ and } W = \text{lfp}_{\geq} F^\bullet \text{ in } (B^\bullet[\neg B]W)
\]

\[
S^\bullet[\text{while } B \text{ } C'\text{ } R] \triangleq \text{let } Y = R \cup S^\bullet[C'](B^\bullet[B]X)\text{ in if } Y \subseteq X \text{ then } X \triangledown Y \text{ and } W = \text{lfp}_{\geq} F^\bullet \text{ in } (B^\bullet[\neg B]W)
\]

\[
S^\bullet[\text{while } B \text{ } C'\text{ } R] \triangleq \text{let } Y = R \cup S^\bullet[C'](B^\bullet[B]X)\text{ in if } Y \subseteq X \text{ then } X \triangledown Y \text{ and } W = \text{lfp}_{\geq} F^\bullet \text{ in } (B^\bullet[\neg B]W)
\]

Note: $F^\bullet$ 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


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)

Concrete Operational Semantics

- 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.

- 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 trace operational semantics
- **Volatile environment** is specified by a trusted configuration file.

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

---

Example application

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

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

---

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).

The Class of Considered Periodic Synchronous Programs

```plaintext
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** [EMSOFT ’01].
Challenging aspects

- Size: > 100 kLOC, > 10 000 variables
- Floating point computations
  including filtering, 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 (Cont’d)

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

Infinitary: 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

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)

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)

**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 → it is a VERIFIER!

---

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!

---

Example of Analysis Session

(Airbus A380 Primary Flight Control Software)

- 450,000 lines
- 0 alarms (Nov. 2004),
- 7h on 2.8 GHz a PC,
- 1 Gigabyte

→ A world grand première!

---

\[^{3}\text{It would be possible to favour computation costs rather than precision, and this should go down. For example, the A340 analysis went up to 5 h, before being reduced by requiring less precision while still getting no false alarm.}\]
Examples of Abstractions

General-Purpose Abstract Domains: Intervals and Octagons

Intervals:
\[
\begin{align*}
1 & \leq x < 9 \\
1 & \leq y < 20
\end{align*}
\]

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

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

Floating-Point Computations

/* 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 = 1.000000019e+38;
    y = x + 1.0e21;
    z = x - 1.0e21;
    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

(1) Floats
Real

Rounds

(2) Doubles
Real

Rounds

Symbolic abstract domain

- 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);

```
X cat = new x.x;
1  void main () { int X, Y;
2     _ASTREE_known_fact(((0 <= X) && (X <= 100)));
3     Y = (X - X);
4     _ASTREE_log_vars(Y);
5   }
```

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 \( \text{clock} \) in the concrete user semantics to track elapsed clock ticks.
  - For each variable \( X \), we abstract three intervals: \( X, X+\text{clock}, \) and \( X-\text{clock} \).
  - If \( X+\text{clock} \) or \( X-\text{clock} \) is bounded, so is \( X \).

```
/* boolean.c */
typedef enum {F=0,T=1} BOOL;
BOOL B;
void main () {
unsig 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 leaves.

Boolean Relations for Boolean Control

- Code Sample:

```
```
Control Partitioning for Case Analysis

---

Control point partitioning:

Fork

Join

Trace partitioning:

- 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

- 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

```c
void main() {
    FIRST = TRUE;
    while (TRUE) {
        dev();
        __ASTREE_wait_for_clock();
    }
}
```
(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 4;
- Inexpressivity i.e. indispensable local inductive invar- ant are inexpressible in the abstract → add a new abstract domain to the reduced product (e.g. filters).

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 \( x + \text{clk} \in [a; b] \land x - \text{clk} \in [a; b] \)
- 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.

Conclusion
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


The Future & Grand Challenges

Forthcoming (1 year):
- More general memory model (union)

Future (5 years):
- Asynchronous concurrency (for less critical software)
- Functional properties (reactivity)
- Industrialization

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

References

[4, 5, 6, 7, 8, 9, 10, 11]


