Software Verification by Abstract Interpretation: Current Trends and Perspectives

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

Rencontre EDF/ENS/X
Paris, 15 November 2004
Talk Outline

- Motivation (1 mn) .................................................. 3
- Abstract interpretation, informally (10 mn) ............... 6
- Applications of abstract interpretation (2 mn) .......... 17
- Application to the verification of embedded, real-time, synchronous, safety super-critical control-command software (10 mn) ........................................... 20
- Examples of abstractions (10 mn) ............................... 35
- Conclusion (2 mn) .................................................. 48
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 $\rightarrow$

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

$x(t)$

Possible trajectories
Safety property

Forbidden zone

Possible trajectories

\[ x(t) \]
Test/Debugging is Unsafe

Forbidden zone

Error !!!

Possible trajectories

Test of a few trajectories
Bounded Model Checking is Unsafe

Forbidden zone

Error !!!

Possible trajectories

Bounded model-checking of trajectory prefixes
Soundness: Erroneous Abstraction — I

Forbidden zone

Possible trajectories

Erroneous trajectory abstraction
Soundness: Erroneous Abstraction — II

Forbidden zone

$x(t)$

Erroneous trajectory abstraction

Error $$$

Possible trajectories

$t$
Imprecision $\Rightarrow$ False Alarms

\[ x(t) \]

Forbidden zone

False alarm

Imprecise trajectory abstraction

Possible trajectories

$t$
Interval Abstraction $\Rightarrow$ False Alarms

$x(t)$

Forbidden zone

False alarms

Imprecise trajectory abstraction by intervals

Possible trajectories
Refinement by Partitionning

\[ x(t) \]

Forbidden zone

Refinement of intervals

Possible trajectories
Applications of Abstract Interpretation
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)
• **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

- Trace-based refinement of the **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

```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** [3].

---

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)

**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 (Cont’d)

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

**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 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, November 2003:
  - 0 alarms,
  - 40mn on 2.8 GHz PC,
  - 300 Megabytes
  
  → A world première!
350,000 lines

0 alarms (mid-October 2004!),
7h \(^1\) on 2.8 GHz PC,
1 Gigabyte

\[\rightarrow A\ world\ grand\ première!\]

\(^1\) 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.
Examples of Abstractions
Intervals:
\[
\begin{align*}
1 & \leq x \leq 9 \\
1 & \leq y \leq 20
\end{align*}
\]

Octagons \[\textbf{[4]}\]:
\[
\begin{align*}
1 & \leq x \leq 9 \\
x + y & \leq 77 \\
1 & \leq y \leq 20 \\
x - y & \leq 04
\end{align*}
\]

Difficulties: many global variables, arrays (smashed or not), IEEE 754 floating-point arithmetic (in program and analyzer) \[\textbf{[5]}\]

Reference


Floating-Point Computations

• Code Sample:

```c
/* 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

(x + a) - (x - a) ≠ 2a

/* 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
```

```
Symbolic abstract domain

- **Interval analysis**: if \(x \in [a, b], y \in [c, d] \& a, c \geq 0 \) 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);

```c
% 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)); }
```

```c
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]>
```

```c
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:**
  ```c
  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 \textit{(impossible using only intervals)}.

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

\textit{Rencontre EDF/ENS/X, 15 November 2004}
Boolean Relations for Boolean Control

- Code Sample:

```c
/* 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 leaves.
Control Partitionning for Case Analysis

- Code Sample:

```c
/* 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 $-100 \leq x \leq 100$ ...

    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

- 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
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 ---

Arithmetic-Geometric Progressions

% 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).
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.
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 $^2$;
- **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).

$^2$ 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
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)
THE END, THANK YOU

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


