Abstract Interpretation: From Theory to Tools

Patrick Cousot

cims.nyu.edu/~pcousot/
pcousot@cims.nyu.edu

Bugs everywhere!

- Ariane 5.01 failure (overflow error)
- Patriot failure (float rounding error)
- Mars orbiter loss (unit error)
- Russian Proton-M/DM-03 rocket carrying 3 Glonass-M satellites (unknown programming error :)

Heartbleed (buffer overrun)

On the limits of bug finding

- Giant software manufacturers can rely on gentle end-users to find myriads of bugs;
- But what about:
  - Can passengers really help?
  - Is dynamic/static bug finding always enough?
  - Proving the absence of bugs is much better!

• These are great proofs of the presence of bugs!
Formal Methods

Mathematical and engineering principles applied to the specification, design, construction, verification, maintenance, and evolution of very high quality software

Strongly promoted by Harlan D. Mills since the 70’s e.g.
- ...

Main formal methods for verification

Objective: prove automatically that a program does satisfy a specification given either explicitly or implicitly (e.g. absence of runtime errors)

Deductive methods: use a theorem prover-proof assistant to check a user-provided proof argument

Enumerative, symbolic, bounded, solver (e.g. Z3)-based, interpolation, statistical, etc model-checking: check the specification by enumerating finitely many possibilities

Abstract interpretation: use approximation ideas to consider infinitely many possibilities

Fundamental limitations

By Gödel’s undecidability, no perfect solution is and will ever be possible:

Deductive methods: the burden is on the end-user and the proofs are exponential in the size of programs

Model-checking: severe unsolved scalability problem

Abstract interpretation: may produce false alarms (but no false negative)

Unsound methods (Coverity, Klocwork, Purify, etc): no correctness guarantee at all.
The Evolution of Formal Methods

Proliferation

<table>
<thead>
<tr>
<th>WCET</th>
<th>Axiomatic semantics</th>
<th>WCET</th>
<th>Axiomatic semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Confidentiality analysis</td>
<td></td>
<td>Confidentiality analysis</td>
</tr>
<tr>
<td></td>
<td>Program synthesis</td>
<td></td>
<td>Program synthesis</td>
</tr>
<tr>
<td></td>
<td>Grammar analysis</td>
<td></td>
<td>Grammar analysis</td>
</tr>
<tr>
<td></td>
<td>Statistical model-checking</td>
<td></td>
<td>Statistical model-checking</td>
</tr>
<tr>
<td></td>
<td>Invariance proof</td>
<td></td>
<td>Invariance proof</td>
</tr>
<tr>
<td></td>
<td>Probabilistic verification</td>
<td></td>
<td>Probabilistic verification</td>
</tr>
<tr>
<td></td>
<td>Parsing</td>
<td></td>
<td>Parsing</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Security protocol verification</th>
<th>Systems biology analysis</th>
<th>Operational semantics</th>
<th>Abstraction refinement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dataflow analysis</td>
<td>Model checking</td>
<td>Database query</td>
<td>CEGAR</td>
</tr>
<tr>
<td>Obfuscation</td>
<td>Denotational semantics</td>
<td>Dependence analysis</td>
<td>Program transformation</td>
</tr>
<tr>
<td>Effect systems</td>
<td>Theories combination</td>
<td>Code contracts</td>
<td>Interpolants</td>
</tr>
<tr>
<td>Trace semantics</td>
<td>Abstract semantics</td>
<td>Integrity analysis</td>
<td>Abstract model checking</td>
</tr>
<tr>
<td>Symbolic execution</td>
<td>Quantum entanglement detection</td>
<td>Bisimulation</td>
<td>SMT solvers</td>
</tr>
<tr>
<td>Quantum entanglement detection</td>
<td>Steganography</td>
<td>Tautology testers</td>
<td></td>
</tr>
<tr>
<td>Type theory</td>
<td>Steganography</td>
<td>Tautology testers</td>
<td></td>
</tr>
</tbody>
</table>

Change of Scale

- **1993**: IBM Flight Control. A HH60 helicopter avionics component was developed on schedule in three increments comprising 33 KLOC of JOVIAL [6]. A total of 79 corrections were required during statistical certification for an error rate of 2.3 errors per KLOC for verified software with no prior execution or debugging.

- **2013**: Astrée checks automatically the absence of any runtime error in the control/command software of the A380 and A400M by abstract interpretation i.e. > 1000 KLOC of C


The Theory of Abstract Interpretation: Unifies Formal Methods
The need for a unified account of formal methods

**Axiomatic semantics**
- Dataflow analysis
- Abstraction refinement

**Denotational semantics**
- Dependence analysis
- Separation logic

**Operational semantics**
- Operational semantics
- Termination proof

**Dataflow analysis**
- Dependence analysis
- Abstraction refinement

**Invariance proof**
- Type inference

**Model checking**
- Symbolic execution
- Detection

**Symbolic execution**
- Abstract model checking
- Malware detection

**Program synthesis**
- Shape analysis

**Grammar analysis**
- Code contracts

**Statistical model-checking**
- Interpolants

**Invariance proof**
- Abstract interpretation

**Quantum entanglement detection**
- SMT solvers

**SMT solvers**
- Abstraction refinement

**Malware detection**
- Code refactoring

**Steganography**
- Detection

**Concrete universe of discourse**

What is abstraction in AI?
What is abstraction in AI?

Concrete universe of discourse

Elements

Concrete universe of discourse

Abstract universe of properties

Elements

Concrete universe of discourse

Abstract universe of properties

Abstract properties

Properties
What is abstraction in AI?

Concrete universe of discourse

Elements

Properties

Abstract universe of discourse

Abstract universe of properties

Properties

Elements

Properties

Concrete universe of discourse

Elements

Properties

Abstract properties

Abstract properties

γ

α

γ

α

Inclusion

⊆

⊑

Abstract implication

Provable abstract properties are true in the concrete
Abstract interpretation: example

Theory:

Applications:

Practice:

I) Define the programming language semantics

Formalize the concrete executions of programs (e.g. transition system)

II) Define the program properties of interest

Formalize what you are interested to know about program behaviors

A very informal introduction to abstract interpretation

Patrick Cousot, Radhia Cousot: Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. POPL 1977: 238-252

Patrick Cousot, Radhia Cousot: Systematic Design of Program Analysis Frameworks. POPL 1979: 269-282
III) Define which specification must be checked
Formalize what you are interested to prove about program behaviors

No trajectory should hit the forbidden zone

IV) Choose the appropriate abstraction
Abstract away all information on program behaviors irrelevant to the proof

Abstraction by geometric forms (rectangles, polyhedra, ellipsoids, abstraction by parts, etc)

V) Mechanically verify in the abstract
The proof is fully automatic

Provable abstract properties are true in the concrete

Soundness of the abstract verification
Never forget any possible case so the abstract proof is correct in the concrete

Abstractation of the trajectories
Unsound validation: testing

Try a few cases

Forbidden zone
Error !!!

Test of a few trajectories

Unsound validation: bounded model-checking

Simulate the beginning of all executions

Forbidden zone
Error !!!

Bounded model-checking

Possible trajectories

Unsound validation: static analysis

Many static analysis tools are **unsound** (e.g. Coverity, etc.) so inconclusive

Forbidden zone
Error !!!

Erroneous trajectory abstraction

Incompleteness

When abstract proofs may fail while concrete proofs would succeed

Forbidden zone
Alarm !!!

Error or false alarm ?

By soundness an alarm must be raised for this overapproximation!
Abstract domains

For example, to avoid a useless widening, this is not possible with analyses carried out independently. Combining the results would enable the interval analysis to continue with the interval and parity analyses find respectively that not very precise as it does not permit analyses to improve each other during the computation. Consider for analyses is sound as in these abstract domains overapproximation of this combination of abstract domains is performed, ideally, by a.

In these abstract domains, because abstract domains do not use a uniform machine representation of the information they manipulate, the combination is performed using an easily computable but not optimal operation or for performance reasons, the conjunction is performed using an easily computable but not optimal operation.

The following classic abstract domains, however, are not used in Astrée.

- **Intervals**: $x \in [a, b]$
- **Simple congruences**: $x \equiv a[b]$
- **Octagons**: $\pm x \pm y \leq a$
- **Ellipses**: $x^2 + by^2 - axy \leq d$
- **Exponentials**: $-a^{bt} \leq y(t) \leq a^{bt}$

Collecting semantics: partial traces

Examples of abstract interpretation-based program verification tools
Example 1: Astrée

Example of domain-specific abstraction: ellipses

```c
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 [?????, ?????] */
}
void main () { X = 0.2 * X + 5; INIT = TRUE;
    while (1) {
        X = 0.9 * X + 35; /* simulated filter input */
    filter (); INIT = FALSE; }
```
Example of domain-specific abstraction: ellipses

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 [?????, ?????] */
} 

void main () { X = 0.2 * X + 5; INIT = TRUE;
    while (1) {
        X = 0.9 * X + 35; /* simulated filter input */
        filter (); INIT = FALSE; }
}

Example II: cccheck

Example of domain-specific abstraction: ellipses

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 [-1418.3753, 1418.3753] */
} 

void main () { X = 0.2 * X + 5; INIT = TRUE;
    while (1) {
        X = 0.9 * X + 35; /* simulated filter input */
        filter (); INIT = FALSE; }
}

Code Contract Static Checker (cccheck)

- Available within MS Visual Studio
Comments on screenshot (courtesy Francesco Logozzo)

- A screenshot from Clousot/cccheck on the classic binary search.
- The screenshot shows from left to right and top to bottom
  1. C# code + CodeContracts with a buggy BinarySearch
  2. cccheck integration in VS (right pane with all the options integrated in the VS project system)
  3. cccheck messages in the VS error list
- The features of cccheck that it shows are:
  1. basic abstract interpretation:
     a. the loop invariant to prove the array access correct and that the arithmetic operation may
        overflow is inferred fully automatically
     b. different from deductive methods as e.g. ESC/Java or Boogie or Dafny where the loop
        invariant must be provided by the end-user
  2. inference of necessary preconditions:
     a. Clousot finds that array may be null (message 3)
     b. Clousot suggests and propagates a necessary precondition invariant (message 1)
  3. array analysis (* disjunctive reasoning):
     a. to prove the postcondition one must infer properties of the content of the array
     b. please note that the postcondition is true even if there is no precondition requiring the
        array to be sorted.
  4. verified code repairs:
     a. from the inferred loop invariant does not follow that index computation does not
        overflow
     b. suggest a code fix for it (message 2)

To explore abstract interpretation...

- A good starting point:

Conclusion

- 40 years after Harlan D. Mills pioneer ideas, abstract interpretation-based formal methods have made considerable progress both in theory and practice
- May become indispensable as
  - safety and security become central to computer science
  - programmers are held responsible for their errors
  - machines hence programming becomes more and more complicated (if not intractable, e.g. parallelism, cloud, etc)
The End, Thank You