Abstract

Abstract interpretation is a theory of abstraction and constructive approximation of the mathematical structures used in the formal description of programming languages and the inference or verification of undecidable program properties.

Developed in the late seventies with Radhia Cousot, it has since then been considerably applied to many aspects of programming, from syntax, to semantics, and proof methods where abstractions are sound and complete but incomputable to fully automatic, sound but incomplete approximate abstractions to solve undecidable problems such as static analysis of infinite state software systems, contract inference, type inference, termination inference, model-checking, abstraction refinement, program transformation (including watermarking), combination of decision procedures, security, malware detection, etc.

This last decade, abstract interpretation has been very successful in program verification for mission- and safety-critical systems. An example is Astrée (www.astree.ens.fr) which is a static analyzer to verify the absence of runtime errors in structured, very large C programs with complex memory usages, and involving complex boolean as well as floating-point computations (which are handled precisely and safely by taking all possible rounding errors into account), but without recursion or dynamic memory allocation. Astrée targets embedded applications as found in earth transportation, nuclear energy, medical instrumentation, aeronautics and space flight, in particular synchronous control/command such as electric flight control or more recently asynchronous systems as found in the automotive industry.

Astrée is industrialized by AbsInt (www.absint.com/astree).
All computer scientists have experienced bugs

Ariane 5.01 failure (overflow)  Patriot failure (float rounding)  Mars orbiter loss (unit error)

• Checking the presence of bugs is great
• Proving their absence is even better!

Abstract interpretation

• Started in the 70’s and widely applied since then
• Based on the idea that undecidability and complexity of automated program analysis can be fought by sound approximations or complete abstractions
• Wide-spectrum theory so applications range from static analysis to verification to biology
• Does scale up!

Fighting undecidability and complexity in program verification

• Any automatic program verification method will definitely fail on infinitely many programs (Gödel)

Solutions:
• Ask for human help (theorem-prover/proof assistant based deductive methods)
• Consider (small enough) finite systems (model-checking)
• Do sound approximations or complete abstractions (abstract interpretation)
An informal introduction to abstract interpretation

I) Define the programming language semantics

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

\( t = 0 \)
\( t = 1 \)

Trajectory in state space \( \Sigma \)

\( (x, y) \in \Sigma \)

Space/time trajectory

II) Define the program properties of interest

Formalize what you are interested to \textit{know} about program behaviors

III) Define which specification must be checked

Formalize what you are interested to \textit{prove} about program behaviors

Forbiden zone
IV) Choose the appropriate abstraction

Abstract away all information on program behaviors irrelevant to the proof.

V) Mechanically verify in the abstract

The proof is fully automatic.

Soundness of the abstract verification

Never forget any possible case so the abstract proof is correct in the concrete.

Unsound validation: testing

Try a few cases.
Unsound validation: bounded model-checking
Simulate the beginning of all executions

Forbidden zone

Bounded model-checking

Error !!!

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

Forbidden zone

Erroneous trajectory abstraction

Error !!!

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!

True error
The abstract alarm may correspond to a concrete error

Forbidden zone

Alarm !!!

Error

Error
False alarm
The abstract alarm may correspond to no concrete error (false negative)

Forbidden zone
Alarm !!!

What to do about false alarms?

- **Automatic refinement**: inefficient and may not terminate (Gödel)
- **Domain-specific abstraction**:
  - Adapt the abstraction to the *programming paradigms* typically used in given *domain-specific applications*
  - e.g. *synchronous control/command*: no recursion, no dynamic memory allocation, maximum execution time, etc.

A Touch of Abstract Interpretation Theory

Fixpoint

- **Set** $\mathcal{P}$
- **Transformer** $F \in \mathcal{P} \rightarrow \mathcal{P}$
- **Fixpoint**

\[ x \in \mathcal{P} \text{ is a fixpoint of } F \iff F(x) = x \]

- **Poset** $\langle \mathcal{P}, \leq \rangle$
- **Least fixpoint**

\[ x \in \mathcal{P} \text{ is the least fixpoint of } F \text{ (written } x = \text{lfp}^{\leq} F) \iff F(x) = x \land \forall y \in \mathcal{P} : (F(y) = y) \Rightarrow (x \leq y) \]
Program properties as fixpoints

- Program semantics and program properties can be formalized as least/greatest fixpoints of increasing transformers on complete lattices (1)

  **Complete lattice / cpo of properties**
  \[ \langle \mathcal{P}, \leq, 0, 1, \lor, \land \rangle \]

- Properties of program \( P \)
  \[ S[P] = \text{lfp}\leq F[P] \]

- Transformer of program \( P \)
  \[ F[P] \in \mathcal{P} \rightarrow \mathcal{P}, \text{increasing (or continuous)} \]

(1) Patrick Cousot, Radhia Cousot: Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. POPL, 1977: 267-280

Example: reachable states

- **Transition system** (set of states \( \Sigma \), initial states \( I \subseteq \Sigma \), transition relation \( \tau \))
  \[ \langle \Sigma, I, \tau \rangle \]

- **Right-image** of a set of states by transitions
  \[ \text{post}[\tau]X \triangleq \{ s' \mid \exists s \in X : \tau(s, s') \} \]

- **Reachable states** from initial states \( I \)
  \[ \text{post}[\tau^*]I = \text{lfp}\subseteq \lambda X \cdot I \cup \text{post}[\tau]X \]

(1) Patrick Cousot, Radhia Cousot: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. POPL, 1977: 267-280

Example: Turing/Floyd Invariance Proof

- Bad states
  \[ B \subseteq \Sigma \]

- Prove that no bad state is reachable
  \[ \text{post}[\tau^*]I \subseteq \neg B \]

- Turing/Floyd proof method
  \[ \exists I \in \varphi(\Sigma) : I \subseteq I \land \text{post}[\tau]I \subseteq I \land I \subseteq \neg B \]
Abstraction

- Abstract the concrete properties into \textit{abstract properties}  
  \[ \langle \mathcal{A}, \sqsubseteq, \bot, T, U, \sqcap \rangle \]

- If any concrete property \( P \in \mathcal{P} \) has a best abstraction \( \alpha(P) \in \mathcal{A} \), then the correspondence is given by a \textit{Galois connection}  
  \[ \langle \mathcal{P}, \sqsubseteq \rangle \xrightarrow{\gamma} \langle \mathcal{A}, \sqsubseteq \rangle \]

\[ \forall P \in \mathcal{P} : \forall Q \in \mathcal{A} : \alpha(P) \sqsubseteq Q \Leftrightarrow P \leq \gamma(Q) \]

Example: elementwise abstraction

- \textit{Morphism}  
  \[ h \in \mathcal{P} \mapsto \mathcal{A} \]

- \textit{Abstraction}  
  \[ \alpha(X) \triangleq \{ h(x) \mid x \in X \} \]

- \textit{Galois connection}  
  \[ \langle \varphi(\mathcal{P}), \sqsubseteq \rangle \xrightarrow{\gamma} \langle \varphi(\mathcal{A}), \sqsubseteq \rangle \]

- \textit{Example: rule of signs}  
  \[ h : \mathbb{Z} \to \{-1, 0, 1\} \]
  \[ h(z) \triangleq z/|z| \]

Abstract transformer

- An abstract transformer \( \overline{F} \in \mathcal{A} \rightarrow \mathcal{A} \) is
  - \textit{Sound} iff  
  \[ \forall P \in \mathcal{P} : \alpha \circ F(P) \sqsubseteq \overline{F} \circ \alpha(P) \]

- \textit{Complete} iff  
  \[ \forall P \in \mathcal{P} : \alpha \circ F(P) = \overline{F} \circ \alpha(P) \]

- \textit{Example (rule of sign)}  
  - Addition: sound, incomplete  
  - Multiplication: sound, complete

Example: rule of signs

\[ \{-1, -2, -7\} \odot \{0, -2, -5\} = \{0, 2, 4, 14, 5, 10, 35\} \]

\[ \alpha \]

\[ \gamma \]

\[ \odot \]

Negative \quad Negative or zero \quad \text{Positive or zero}
Fixpoint abstraction

- For an increasing and sound abstract transformer, we have a \textit{fixpoint approximation}

\[ \alpha(\text{lfp} \subseteq F) \subseteq \text{lfp} \subseteq F \]

- For an increasing, sound, and complete abstract transformer, we have an \textit{exact fixpoint abstraction}

\[ \alpha(\text{lfp} \subseteq F) = \text{lfp} \subseteq F \]

Iterative fixpoint computation

- Fixpoint of increasing transformers on cpos can be computed iteratively as limits of (transfinite) iterates

\[
\begin{align*}
F^0 &\triangleq \bot \\
F^{\beta+1} &\triangleq F(F^\beta), \quad \beta + 1 \text{ successor ordinal} \\
F^\lambda &\triangleq \bigcup_{\beta < \lambda} F^\beta, \quad \lambda \text{ limit ordinal} \\
\text{Ultimately stationary at rank } \epsilon \\
\text{Converges to } F^\epsilon = \text{lfp} \subseteq F
\end{align*}
\]

- \( \epsilon = \omega \) when \( F \) is continuous

- Finite iterates when \( F \) operates on a cpo satisfying the ascending chain condition

Widening

- Definition (widening \( \triangledown \in \mathcal{A} \times \mathcal{A} \rightarrow \mathcal{A} \))

- \( \langle \mathcal{A}, \subseteq \rangle \) poset

- Over-approximation

\[ \forall x, y \in \mathcal{A} : x \subseteq x \triangledown y \land y \subseteq x \triangledown y \]

- Termination

Given any sequence \( \langle x^n, n \in \mathbb{N} \rangle \), the widened sequence \( \langle y^n, n \in \mathbb{N} \rangle \)

\[
y^0 \triangleq x^0, \ldots, y^{n+1} \triangleq y^n \triangledown x^n, \ldots
\]

converges to a limit \( y^\ell \) (such that \( \forall m \geq \ell : y^m = y^\ell \))

Example: (simple) widening for polyhedra

- Iterates

- Widening

\[ \bar{F}^n \cap \bar{F}(F^n) \]
Iteration with widening

- **Iterates with widening** for transformer $\overline{F} : \mathcal{A} \rightarrow \mathcal{A}$
  
  $\overline{F}^0 \triangleq \bot$
  
  $\overline{F}^{n+1} \triangleq \overline{F}^n$ when $\overline{F}(\overline{F}^n) \subseteq \overline{F}^n$
  
  $\overline{F}^{n+1} \triangleq \overline{F} \uplus \overline{F}(\overline{F}^n)$ otherwise

- The **widening speeds up convergence** (at the cost of imprecision)

**Theorem** (Limit of iterates with widening) The iterates of $\overline{F}$ with widening $\uplus$ from $\bot$ on a poset $\langle \mathcal{A}, \subseteq, \bot \rangle$ converge to a limit $\overline{F}$ such that $\overline{F}(\overline{F}) \subseteq \overline{F}$ (and so lfp$\overline{F} \subseteq \overline{F}$ when $\overline{F}$ is increasing).

- Can be improved by a **narrowing**.

---

**Intuition for iteration with widening**

- Iteration

- Iteration with widening (using the derivative as in Newton-Raphson method)

---

**Reduced product**

- The **reduced product** combines abstractions by performing their conjunction in the abstract

- Example: (positive or zero) $\otimes$ odd = <positive,odd>

---

**Recent advances**

- The same principles apply to **termination**

  - Patrick Cousot, Radhia Cousot: An abstract interpretation framework for termination. POPL 2012: 245-258

- and to **probabilistic programs**

ASTRÉE

Target language and applications

• C programming language
  • Without recursion, longjump, dynamic memory allocation, conflicting side effects, backward jumps, system calls (stubs)
  • With all its horrors (union, pointer arithmetics, etc)
  • Reasonably extending the standard (e.g. size & endianess of integers, IEEE 754-1985 floats, etc)
  • Originally for synchronous control/command
    • e.g. generated from Scade

Implicit specification

• Absence of runtime errors: overflows, division by zero, buffer overflow, null & dangling pointers, alignment errors, …
• Semantics of runtime errors:
  • Terminating execution: stop (e.g. floating-point exceptions when traps are activated)
  • Predictable outcome: go on with worst case (e.g. signed integer overflows result in some integer, some options: e.g. modulo arithmetics)
  • Unpredictable outcome: stop (e.g. memory corruption)
Abstractions

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

Example of general purpose abstraction: octagons

- Invariants of the form $\pm x \pm y \leq c$, with $O(N^2)$ memory and $O(N^3)$ time cost.
- Example:

```c
typedef enum {F=0, T=1} BOOL;

void main () {
    unsigned int X, Y;
    while (1) {
        ...
        B = (X == 0);
        ...
        if (!B) {
            Y = 1 / X;
        }    
        ...
    }
}
```

- At $\star$, the interval domain gives $L \leq \max(\max A, (\max Z)+(\max V))$.
- In fact, we have $L \leq A$.
- To discover this, we must know at $\star$ that $R = A-Z$ and $R > V$.
- Here, $R = A-Z$ cannot be discovered, but we get $L-Z \leq \max R$ which is sufficient.
- We use many octagons on small packs of variables instead of a large one using all variables to cut costs.

Example of general purpose abstraction: decision trees

```c
/* boolean.c */
typedef enum {F=0, T=1} BOOL;

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.

Example of domain-specific abstraction: ellipses

```c
typedef enum {FALSE = 0, TRUE = 1} BOOLEAN;

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; }
}
Example of domain-specific abstraction: exponentials

```c
% cat count.c
typedef enum {FALSE = 0, TRUE = 1} BOOLEAN;
volatile BOOLEAN I; int R; BOOLEAN T;
void main() {
R = 0;
while (TRUE) {
  __ASTREE_log_vars(R);
  if (I) { R = R + 1; } else { R = 0; }
  T = (R >= 100);
  __ASTREE_wait_for_clock();
}
}
% cat count.config
__ASTREE_volatile_input((I [0,1]));
__ASTREE_max_clock((3600000));
% astree -exec-fn main -exec-sq-sem count.config count.c|grep '\|R\|'
|R| <= 0. + clock *1. <= 3600001.
```

Example of domain-specific abstraction: exponentials

```c
% 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) + 4.91048e-33);
    E = 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
```

An erroneous common belief on static analyzers

“The properties that can be proved by static analyzers are often simple” [2]

Like in mathematics:
- May be simple to state (no overflow)
- But harder to discover \(s[0], s[1]\) in \([-1327, 0.2698354, 1327.02698354]\)
- And difficult to prove (since it requires finding a non trivial non-linear invariant for second order filters with complex roots [Fer04], which can hardly be found by exhaustive enumeration)

Reference


Industrial applications
Examples of applications

- Verification of the absence of runtime-errors in
  - Fly-by-wire flight control systems (*)
  - ATV docking system (*)
- Flight warning system (on-going work) (*)

(*) No false alarm at all!

Industrialization

- 8 years of research (CNRS/ENS/INRIA):
  www.astree.ens.fr
- Industrialization by AbsInt (since Jan. 2010):
  www.absint.com/astree/

ASTRÉEA: Verification of embedded real-time parallel C programs
Parallel programs

- Bounded number of processes with shared memory, events, semaphores, message queues, blackboards,…
- Processes created at initialization only
- Real time operating system (ARINC 653) with fixed priorities (highest priority runs first)
- Scheduled on a single processor

Verified properties

- Absence of runtime errors
- Absence of unprotected data races

Semantics

- No memory consistency model for C
- Optimizing compilers consider sequential processes out of their execution context

```
init: flag1 = flag2 = 0

process 1: flag1 = 1;
if (!flag2)
{
    /* critical section */
}

process 2: flag2 = 1;
if (!flag1)
{
    /* critical section */
}
```

write to flag1/2 and read of flag2/1 are independent so can be reordered → error!

- We assume:
  - sequential consistency in absence of data race
  - for data races, values are limited by possible interleavings between synchronization points

Abstractions

- Based on Astrée for the sequential processes
- Takes scheduling into account
- OS entry points (semaphores, logbooks, sampling and queuing ports, buffers, blackboards, …) are all stubbed (using Astrée stubbing directives)
- Interference between processes: flow-insensitive abstraction of the writes to shared memory and inter-process communications

Example of application: FWS

- Degraded mode (5 processes, 100 000 LOCS)
  - 1h40 on 64-bit 2.66 GHz Intel server
  - A few dozens of alarms
- Full mode (15 processes, 1 600 000 LOCS)
  - 24 h
  - a few hundreds of alarms !!! work going on !!!
    (e.g. analysis of complex data structures, logs, etc)
Conclusion

Cost-effective verification

- The rumor has it that:
  - Manuel validation (testing/debugging/bug finding) is costly, unsafe, not a verification!
  - Formal proofs by theorem provers are extremely laborious hence costly to create and maintain for program/specifications changing over time (15/20 years for planes)
  - Model-checkers are unsound or do not scale up for complex software (which is unbounded)

Cost-effective verification

- Why not try abstract interpretation?
  - Domain-specific static analysis scales and can deliver no or few false alarms on large industrial code!
  - Conceptual bugs are discovered through their consequences on runtime errors
  - Very cost effective
  - Compliant with DO178C formal methods!

The End, Thank You