The Verification Grand Challenge and Abstract Interpretation

Patrick Cousot

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

Verified Software: Theories, Tools, Experiments
Zürich, ETH, Oct. 10th–14th, 2005

Abstract interpretation

Abstract interpretation

- Abstract interpretation is a mathematical theory of sound approximation of properties of formal systems (including program specifications, semantics, . . .)
- Abstraction is central to the comprehension of complex systems (such as software)
- Discovering new, useful, reusable abstractions can be a full time job

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 successful example: The ASTRÉE static analyzer

The ASTRÉE static analyzer

- Verify the absence of runtime errors in C programs:
  - out-of-bound array accesses[1]
  - integer division by zero
  - IEEE 754-1985 floating point operations overflows and invalid operations (producing Inf or NaN[2])
  - integer arithmetics or cast wrap around, …
- No `union`, `malloc`, recursion, library, strings, …

  … as usual in many (automatically generated) synchronous, time-triggered, real-time, safety critical, embedded software as found in automotive, energy and aerospace applications

Industrial applications

- Nov. 2003: absence of any RTE in the primary flight control software of the fly-by-wire system of a family of existing commercial planes (generated from a proprietary specification language), 132.000 lines
- Mar. 2005: absence of any RTE in the primary flight control software of the fly-by-wire system of commercial plane under certification (generated from a proprietary specification language/SCADE), 500.000 lines, No false alarm (a world première)
- Oct. 2005: 1.000.000 lines

Objectives: verification of binary code (+3 months), automatic analysis of the origin of errors (+6 months), asynchronous communication (+1 year), asynchronous processes (+2 years), …

---

[1] It is completely wrong that "we don't need a proof but a proper compiler": discovering the error at runtime is too late, no compiler checks these verification conditions

[2] Well-written programs check for Inf/NaN inputs which must be shown statically not to propagate

Abstractions

Abstraction of sets of traces[3] with

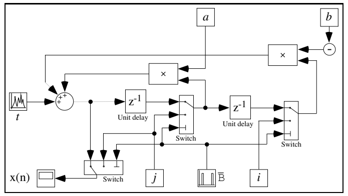
- Intervals abstract domain (basic domain necessary to check the absence of RTE)
- Octagons abstract domain
- Digital filters abstract domain
- Decision trees abstract domain
- Control/data partitioning to handle disjunctions
- . . .

Preprocessing to handle C macros. Abstract domains are parameterized to tailor cost/precision, they talk/communicate symbolically through mutual queries to implement the reduced product

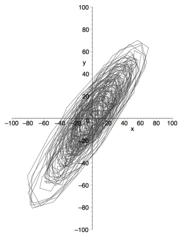
---

Ellipsoid Abstract Domain for Filters

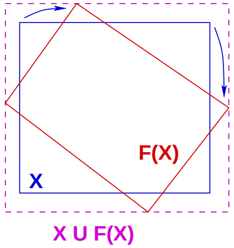
2^{d} Order Digital Filter:



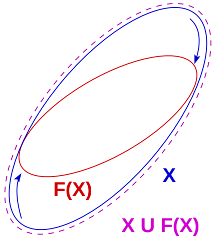
- Computes $X_n = \begin{cases} \alpha X_{n-1} + \beta X_{n-2} + Y_n \\ I_n \end{cases}$
- The concrete computation is bounded, which must be proved in the abstract.
- There is no stable linear invariant
- The simplest stable surface is an ellipsoid



execution trace



unstable interval



stable ellipsoid

---

[3] i.e. more refined that invariants

Filter Example

```
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

see http://www.astree.ens.fr/

Arithmetic-geometric progressions

- Abstract domain: $(\mathbb{R}^+)^5$ [4]
- Concretization (any function bounded by the arithmetic-geometric progression):

$$\gamma \in (\mathbb{R}^+)^5 \longmapsto \wp(\mathbb{N} \mapsto \mathbb{R})$$

$$\gamma(M, a, b, a', b') =$$

$$\{f \mid \forall k \in \mathbb{N} : |f(k)| \leq \left(\lambda x \cdot ax + b \circ (\lambda x \cdot a'x + b')^k\right)(M)\}$$

Reference

see http://www.astree.ens.fr/

---

[4] here in $\mathbb{R}$

Arithmetic-Geometric Progressions (Example 1)

```
% 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; }          ← potential overflow!
    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 -config-sem count.config count.c|grep '|R|'
|R| <= 0. + clock *1. <= 3600001.
```

Arithmetic-geometric progressions (Example 2)

```
% 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
```

Directions for application of abstract interpretation to the verification grand challenge

Program verification

Following E.W.D. Dijkstra:

- Program testing: presence of bugs
  - *dynamic* (e.g. program monitoring, . . .)
  - *static* (error pattern recognition, prefix (model)-checking, . . .)
- Program verification: absence of bugs
  - *static*

The Verification Grand Challenge is on verification (???).

Error tracing

- Bugs or false alarms are found during the verification process
- Abstract slicing can extract the part of the program (control + data) which may be responsible for the error
- Parametric abstraction can be used to provide counter-examples
- This can be hard (e.g. accumulation of rounding errors in floating point computations for hours)

Program semantics

- A program is checked with respect to its semantics (internal specification)
- Precise formal semantics (usable for program verification, including at the implementation level) are missing for the most common languages (e.g. C[5])
- No semantics is universal
- Abstract interpretation unifies semantics according to their level of abstraction and can be used to prove their consistency

---

[5] The semantics of C is λP : Program texts · λM : Machine · λS : System · λL : Linker · λC : Compiler · $\mathbb{S}[C, L, S, M][\![P]\!]$ . . . described informally

Specifications

- Specifications translate external requirements in terms of the program semantics
- Specifications are erroneous
- Specifications must be checked with respect to specifications of the specification
- Static analysis by abstract interpretation could be useful for specification verification

On specification satisfaction

- Specification satisfaction can be verified in part
- Such parts are abstractions of the specification (e.g. absence of RTE)
- This shows the need for abstractions of specifications
- Abstract interpretation
  - unifies specifications at various levels of abstraction
  - can be used to prove their consistency
  - can be used to specify by parts (through complex combinations of abstractions)

Complex systems

- Engineers abstract complex physical systems (e.g. using mathematical models)
- Computer scientists abstract complex program computations (e.g. using abstract interpretation)
- A unification of abstraction in computer science and engineering sciences is necessary for the full verification of complex systems, including
  - Abstract models of a program (e.g. using abstract semantics)
  - Abstract models of its environment (e.g. using physical models)

Proofs, abstractions and false alarms

- A program proof involves a program-specific inductive argument
- A static analysis involves a program specific abstraction
- Discovering an appropriate abstraction (e.g. by refinement fixpoint iteration) is equivalent to discovering an inductive proof
- There is no false alarm only if the proof weakest inductive argument is expressible in the abstract

Verification of program families

- How to invent inductive arguments/abstractions avoiding false alarms?
- We can consider program families for which inductive arguments/abstractions are similar
- Examples:
  - Absence of runtime error in synchronous control command programs (ASTRÉE)
  - Sorting, list processing,... (TVLA)
  - Scientific and signal processing applications (PIPS)
  - Numerical programs (Fluctuat)

Application-aware verifiers

- General-purpose verifiers are difficult to built
- Domain-specific verifiers can be made powerful and efficient by incorporating knowledge about programs and specifications
- Example (for digital filters):
  - Polynomial assertions[6], versus
  - Ellipsoidal assertions[7], versus
  - Polyhedral assertions[8], ...

---

[6] Too expensive

[7] OK, if implemented very efficiently and used locally in the program analysis

[8] Not stable

Domains of abstract assertions

- Universal representations (e.g. terms in theorem provers or BDDs in model-checkers) are not always efficient
- Dedicated representations are always algorithmically more efficient
- We can develop reusable libraries of dedicated abstractions [9]

Combination of abstractions

- The modular combination of abstract domains (e.g. reduced product) allow universal uses of dedicated representations
- A domain-specific static analyzer can be built by combining appropriate abstract domains
- This is a generalization from:
  - the design of an inductive argument (e.g. invariant) for a specific program (invariant generator), to
  - the design of an appropriate abstract domain combination for a program family ((invariant generator) generator)

---

[9] e.g. APRON project in France: interchangeable numeric abstractions

Abstract solvers

- Abstract solvers can take various forms:
  - Elimination
  - Iterative
  - Convergence acceleration
  - . . .
- Progress needed on reusable, generic, parametric and modular abstract solvers

Modular analyzers

- Static analyzers are extremely complex
- Efficient static analyzers can be designed by modular combination of abstract domains and abstract solvers
- This leads to a wide spectrum of domain-aware verifiers as opposed to a universal one

The verified verifier (heavy version)

- Any verifier must be qualified (e.g. verified)
- Abstract interpretation formalizes the design and correctness of static analyzers
- An abstract interpretation-based static analyzer is fully formally specified and can be fully verified [10]

The verified verifier (light version)

- A static analyzer computes an assertion and checks that it is inductive
- The computation of the abstract inductive assertion (e.g. invariant) need not be verified
- The check that the abstract assertion is inductive must be verified
- This is much simpler than a complete correctness proof!
- A verified inductiveness checker can be extracted from the correctness proof (COQ) and run occasionally to validate the abstract assertion (despite its inefficiency)

---

[10] e.g. in COQ as in D. Pichardie thesis, to appear

Acceptance and dissemination of static analysis

- Ultimate success is in effective industrial applications
- Measured only by economic payoff criteria
- Hard to estimate the potential cost of errors discovered by static analysis [11]
- The public demand on software quality might increase
- Regulation might also be necessary (e.g. for safety critical software) to raise the law to the state of the art
- Static analysis (as available at design time) can check a posteriori for fatal errors, which can determine responsibilities in case of software failures

Conclusion

---

[11] The Ariane 5.01 bug is worth billions of $ if discovered by failure after departure but 0 $ if known before!

Conclusion

- Abstraction is indispensable for the Verification Grand Challenge
- The challenge for abstract interpretation is to extend its scope to complex systems, from specification to implementation, including engineering considerations

THE END, THANK YOU

References

[1] `www.astree.ens.fr` [3, 4, 5, 6, 7, 8, 9, 10]

[2] P. Cousot. *Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique de programmes.* Thèse d'État ès sciences mathématiques, Université scientifique et médicale de Grenoble, Grenoble, France, 21 March 1978.

[3] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. *The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, LNCS 2566, pp. 85–108. Springer, 2002.

[4] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. *PLDI'03*, San Diego, pp. 196–207, ACM Press, 2003.

[POPL '77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY, USA.

[PACJM '79] P. Cousot and R. Cousot. Constructive versions of Tarski's fixed point theorems. Pacific Journal of Mathematics 82(1):43–57 (1979).

[POPL '78] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 84–97, Tucson, Arizona, 1978. ACM Press, New York, NY, U.S.A.

[POPL '79] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 269–282, San Antonio, Texas, 1979. ACM Press, New York, NY, U.S.A.

[POPL '92] P. Cousot and R. Cousot. Inductive Definitions, Semantics and Abstract Interpretation. In Conference Record of the 19th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Programming Languages, pages 83–94, Albuquerque, New Mexico, 1992. ACM Press, New York, U.S.A.

[FPCA '95] P. Cousot and R. Cousot. Formal Language, Grammar and Set-Constraint-Based Program Analysis by Abstract Interpretation. In *SIGPLAN/SIGARCH/WG2.8 7th Conference on Functional Programming and Computer Architecture, FPCA'95*. La Jolla, California, U.S.A., pages 170–181. ACM Press, New York, U.S.A., 25-28 June 1995.

[POPL '97] P. Cousot. Types as Abstract Interpretations. In Conference Record of the 24th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Programming Languages, pages 316–331, Paris, France, 1997. ACM Press, New York, U.S.A.

[POPL '00] P. Cousot and R. Cousot. Temporal abstract interpretation. In *Conference Record of the Twentyseventh Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 12–25, Boston, Mass., January 2000. ACM Press, New York, NY.

[POPL '02] P. Cousot and R. Cousot. Systematic Design of Program Transformation Frameworks by Abstract Interpretation. In *Conference Record of the Twentyninth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 178–190, Portland, Oregon, January 2002. ACM Press, New York, NY.

[TCS 277(1–2) 2002] P. Cousot. Constructive Design of a Hierarchy of Semantics of a Transition System by Abstract Interpretation. *Theoretical Computer Science* 277(1–2):47–103, 2002.

[TCS 290(1) 2002] P. Cousot and R. Cousot. Parsing as abstract interpretation of grammar semantics. *Theoret. Comput. Sci.*, 290:531–544, 2003.

[Manna's festschrift '03] P. Cousot. Verification by Abstract Interpretation. *Proc. Int. Symp. on Verification – Theory & Practice – Honoring Zohar Manna's 64th Birthday*, N. Dershowitz (Ed.), Taormina, Italy, June 29 – July 4, 2003. Lecture Notes in Computer Science, vol. 2772, pp. 243–268. © Springer-Verlag, Berlin, Germany, 2003.

[5] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE analyser. *ESOP 2005*, Edinburgh, LNCS 3444, pp. 21–30, Springer, 2005.

[6] J. Feret. Static analysis of digital filters. *ESOP'04*, Barcelona, LNCS 2986, pp. 33—-48, Springer, 2004.

[7] J. Feret. The arithmetic-geometric progression abstract domain. In *VMCAI'05*, Paris, LNCS 3385, pp. 42–58, Springer, 2005.

[8] Laurent Mauborgne & Xavier Rival. Trace Partitioning in Abstract Interpretation Based Static Analyzers. *ESOP'05*, Edinburgh, LNCS 3444, pp. 5–20, Springer, 2005.

[9] A. Miné. A New Numerical Abstract Domain Based on Difference-Bound Matrices. *PADO'2001*, LNCS 2053, Springer, 2001, pp. 155–172.

[10] A. Miné. Relational abstract domains for the detection of floating-point run-time errors. *ESOP'04*, Barcelona, LNCS 2986, pp. 3—17, Springer, 2004.

[POPL '04] P. Cousot and R. Cousot. An Abstract Interpretation-Based Framework for Software Watermarking. In *Conference Record of the Thirtyfirst Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 173–185, Venice, Italy, January 14-16, 2004. ACM Press, New York, NY.

[DPG-ICALP '05] M. Dalla Preda and R. Giacobazzi. Semantic-based Code Obfuscation by Abstract Interpretation. In Proc. 32nd Int. Colloquium on Automata, Languages and Programming (ICALP'05 – Track B). LNCS, 2005 Springer-Verlag. July 11-15, 2005, Lisboa, Portugal. To appear.

[EMSOFT '01] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and precise WCET determination for a real-life processor. *ESOP (2001)*, LNCS 2211, 469–485.

[RT-ESOP '04] F. Ranzato and F. Tapparo. Strong Preservation as Completeness in Abstract Interpretation. ESOP 2004, Barcelona, Spain, March 29 - April 2, 2004, D.A. Schmidt (Ed), LNCS 2986, Springer, 2004, pp. 18–32.