Automatic Verification by Abstract Interpretation

(Invited tutorial)

Patrick Cousot

École normale supérieure, Département d'informatique,
45 rue d'Ulm, 75230 Paris cedex 05, France
Patrick.Cousot@ens.fr www.di.ens.fr/~cousot/

Abstract. We discuss the use of abstract interpretation in the context of automatic program verification requiring precise abstractions. We compare entirely manual versus user-guided abstractions ranging from program-specific abstractions including predicate abstraction to the systematic design of abstract domains and iteration strategies.

1 Abstract Interpretation Theory

Abstract interpretation theory [1,2,3,4,5,6] formalizes the notion of abstraction for mathematical constructs involved in the specification of computer systems. Applications range from *static program analysis* [2,3,4,6] (including data-flow analysis [3], set-based analysis [7], etc), *typing* [8], *model-checking* [9], *parsing* [10] to the *design of semantics* [11] and *program transformations* [12]. In this invited tutorial we discuss applications to *automatic program verification*.

2 Requirements

When dealing with undecidable questions on program execution, the automatic verification problem must conciliate *correctness* (which excludes non exhaustive methods such as simulation or test), *automation* (as opposed to the manual production of a program model for model-checking or to the human assistance for provers in deductive methods), *precision* (which excludes rudimentary general-purpose static program analyzers which would produce too many false alarms), *scaling up* (for software of a few hundred thousand lines), and *efficiency* (with minimal space and time requirements allowing for rapid verification during the software production process which excludes a costly iterative refinement process). Note that we consider automatic verification for proving the absence of errors, not their presence that is verification is considered in the sense of proof, not as debugging.

3 Efficiency versus Precision

Among applications of abstract interpretation there have been many where somewhat imprecise answers to undecidable questions are tolerable provided all answers are sound and the imprecision rate remains low (typically 5 to 15%). This is the case for static program analysis when applied to *program optimization* (such as static elimination of run-time array bound checks where imprecision means delaying few array bound checks at run-time [13]), typing (where some programs which cannot go wrong are not typable) [8] or to *program transformation* (such as *partial evaluation* where any static value can always be considered dynamic, the transformed program being simply less efficient) [12], etc. In that case the analysis must be more efficient than precise. So coarse abstractions can be used which allow for the design of time and memory efficient static analyzers scaling up for very large programs.

4 Precision versus Efficiency

In the context of automatic program verification where human interaction must be reduced to a strict minimum, false alarms are undesirable. A 5% rate of false alarms on a program of a few hundred thousand lines would require several person-years effort to manually prove that no error is possible.

Fortunately, the abstract interpretation theory shows that for any program (or finite set of programs), it is possible to achieve full precision and great efficiency [14] by discovering an appropriate abstract domain. In the following we discuss the user-guided design of such abstract domains leading to precise and efficient analyzes.

5 Program-Specific Finite Abstraction

The use of a specific abstraction for a given hardware or software computer system (often called a model [15]) explains the popularity of *abstract model checking* [16]: it is always possible to provide an appropriate model of a given computer system which will model-check for the given property to be verified. The difficulty is how to get this appropriate model from a formal specification of the computer system such as a program. Most fully automatic methods, such as *software model-checking* [17,18], do not proceed directly on the software but on a user-provided finite and small model, which is difficult to design when e.g. sharp data properties must be taken into account. Moreover models for one program are hardly reusable for another program so efforts to design different models for different programs can hardly be cumulated.

6 Foundations of Predicate Abstraction

Predicate abstraction, which consists in specifying a boolean abstraction of software by providing the atomic elements of the abstract domain in logical form

[19], is certainly the most studied alternative [20,21,22]. Using a theorem prover, it is possible to automatically generate the abstract model in boolean form from the user-provided basic predicates and then to reuse existing model checkers. Moreover most implementations incorporate an automatic refinement process by success and failure [20,23] so that the abstraction can be partly automated.

We will first recall that predicate abstraction is an abstract interpretation and show why.

7 Predicate Abstraction in the Large

Then we will discuss a number of difficulties which in light of a recent experience in software verification [24] seem insurmountable to automate this design process in the present state of the art of deductive methods:

Problems of Semantics: for C programs, the prover which is used to automatically design abstract transfer functions has to take the machine-level semantics into account (e.g. floating-point arithmetic with rounding errors as opposed to real numbers). For example ESC is simply unsound with respect to modulo arithmetics [25].

State Explosion Problem: for large programs, the number of needed basic predicates can be huge. One difficulty is that model checking algorithms have worst-case behavior that is exponential in the number of predicates in the model which leads to state explosion. Another difficulty is to anticipate *a priori* which set of predicates introduced in the abstraction will be ultimately useful in the program analysis. The main successes seem to be when the full program can be abstracted very roughly into a small skeleton [15].

Refinement Problem: predicate abstraction *per se* uses a finite domain and is therefore of limited expressive power in comparison with the use of infinite abstract domains [6]. Therefore predicate abstraction is often accompanied by a refinement process to cope with false alarms [20,23]. Under specific conditions, this refinement can be proved equivalent to the use of an infinite abstract domain with widening [26]. This result is of limited scope since these specific conditions (essentially that the widening is by constraint elimination) are not satisfied e.g. by the staged widening with thresholds of [24]. Formally this counterexample-based refinement is a fixpoint computation [14,27] at the concrete semantics level, whence introduces new elements in the abstract domain state by state. In general, this process is very costly so that the needed predicates have to be provided by hand which introduces prohibitive human and computational costs for end-users.

8 Generic Abstractions

Finally we discuss a more synthetic general point of view based on the use of adequate parameterized abstract domains and iteration strategies with efficient implementations. This can be used to generate abstractions for specific classes

of programs and properties to get efficient generic analyzers producing few or none false alarms [24].

References

1. Cousot, P.: 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, FR (1978)
2. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: 4^{th} POPL, Los Angeles, CA, ACM Press (1977) 238–252
3. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: 6^{th} POPL, San Antonio, TX, ACM Press (1979) 269–282
4. Cousot, P.: Semantic foundations of program analysis. In Muchnick, S., Jones, N., eds.: Program Flow Analysis: Theory and Applications. Prentice-Hall (1981) 303–342
5. Cousot, P., Cousot, R.: Abstract interpretation frameworks. J. Logic and Comp. **2** (1992) 511–547
6. Cousot, P., Cousot, R.: Comparing the Galois connection and widening/narrowing approaches to abstract interpretation, invited paper. In Bruynooghe, M., Wirsing, M., eds.: Proc. 4^{th} Int. Symp. PLILP '92. Leuven, BE, 26–28 Aug. 1992, LNCS 631, Springer-Verlag (1992) 269–295
7. Cousot, P., Cousot, R.: Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In: Proc. 7^{th} FPCA, La Jolla, CA, ACM Press (1995) 170–181
8. Cousot, P.: Types as abstract interpretations, invited paper. In: 24^{th} POPL, Paris, FR, ACM Press (1997) 316–331
9. Cousot, P., Cousot, R.: Temporal abstract interpretation. In: 27^{th} POPL, Boston, MA, ACM Press (2000) 12–25
10. Cousot, P., Cousot, R.: Parsing as abstract interpretation of grammar semantics. Theoret. Comput. Sci. **290** (2002) 531–544
11. Cousot, P.: Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. Theoret. Comput. Sci. **277** (2002) 47–103
12. Cousot, P., Cousot, R.: Systematic design of program transformation frameworks. In: 29^{th} POPL, Portland, OR, ACM Press (2002) 178–190
13. Cousot, P., Cousot, R.: Static determination of dynamic properties of programs. In: Proc. 2^{nd} Int. Symp. on Programming, Dunod (1976) 106–130
14. Cousot, P.: Partial completeness of abstract fixpoint checking, invited paper. In Choueiry, B., Walsh, T., eds.: Proc. 4^{th} Int. Symp. SARA '2000. Horseshoe Bay, TX, US, LNAI 1864. Springer-Verlag (2000) 1–25
15. Clarke, E., Emerson, E.: Synthesis of synchronization skeletons for branching time temporal logic. In: IBM Workshop on Logics of Programs. Yorktown Heights, NY, US, LNCS 131, Springer-Verlag (1981)
16. Clarke, E., Grumberg, O., Long, D.: Model checking and abstraction. TOPLAS **16** (1994) 1512–1542
17. Holzmann, G.: The model checker Spin. IEEE Trans. Software Engrg. **23** (1997) 279–295

18. Holzmann, G.: Software analysis and model checking. In Brinksma, E., Larsen, K., eds.: Proc. 14^{th} Int. Conf. CAV '2002. Copenhagen, DK, LNCS 2404, Springer-Verlag (2002) 1–16
19. Graf, S., Saïdi, H.: Construction of abstract state graphs with PVS. In Grumberg, O., ed.: Proc. 9^{th} Int. Conf. CAV '97. Haifa, IL, LNCS 1254, Springer-Verlag (1997) 72–83
20. Ball, T., Majumdar, R., Millstein, T., Rajamani, S.: Automatic predicate abstraction of C programs. In: Proc. ACM SIGPLAN 2001 Conf. PLDI. ACM SIGPLAN Not. 36(5), ACM Press (2001) 203–213
21. Das, S., Dill, D., Park, S.: Experience with predicate abstraction. In Halbwachs, N., Peled, D., eds.: Proc. 11^{th} Int. Conf. CAV '99. Trento, IT, LNCS 1633, Springer-Verlag (1999) 160–171
22. Henzinger, T., Jhala, R., Majumdar, R., G.Sutre: Lazy abstraction. In: 29^{th} POPL, Portland, OR, US, ACM Press (2002) 58–70
23. Das, S., Dill, D.: Counter-example based predicate discovery in predicate abstraction. In Aagaard, M., O'Leary, J., eds.: Proc. 4^{th} Int. Conf. on Formal Methods in Computer-Aided Design, FMCAD 2002. Portland,, OR, US, LNCS 1633, SPRINGER (2002) 19–32
24. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software, invited chapter. In Mogensen, T., Schmidt, D., Sudborough, I., eds.: The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones. LNCS 2566. Springer-Verlag (2002) 85–108
25. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J., Stata, R.: Extended static checking for Java. In: Proc. ACM SIGPLAN 2002 Conf. PLDI. ACM SIGPLAN Not. 37(5), ACM Press (2002) 234–245
26. Katoen, J.P., Stevens, P., eds.: Relative Completeness of Abstraction Refinement for Software Model Checking. In Katoen, J.P., Stevens, P., eds.: Proc. 8^{th} Int. Conf. TACAS '2002. Grenoble, FR, LNCS 2280, Springer-Verlag (2002)
27. Giacobazzi, R., Quintarelli, E.: Incompleteness, counterexamples and refinements in abstract model-checking. In Cousot, P., ed.: Proc. 8^{th} Int. Symp. SAS '01. Paris, FR, LNCS 2126, Springer-Verlag (2001) 356–373