Directions for Research in Approximate System Analysis

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

Program analysis is mainly concerned with the design of program analyzers to automatically determine semantic properties of programs written in some programming language. Such analyzers take programs as input and output some useful information about their runtime behavior. This information is useful for optimizing compilers [15], partial evaluators [11], abstract debuggers [1], models-checkers [2], formal verifiers [13], etc. The difficulty of the task comes from the fact that programs are infinite states so that all interesting questions about program executions are undecidable. Hence the automatically produced information, although sound, must be incomplete.

With the appearance of new computing paradigms, the scope of program analysis has been constantly broadening these last two decades. The term "program static analysis" is therefore too restricted since the analysis problem appears as soon as one considers computer systems with states which evolve continuously or discretely over time, from term rewriting to communication protocols, critical embedded real-time systems and image compression.

Three approaches have been considered:

- Formal methods based on general or dedicated theorem proving techniques. This approach is quite general although, because of undecidability, it ultimately relies on human interaction which can be prohibitive if not impossible for very large systems [13];

- The design of specific algorithms for decidable subproblems, such as, e.g., finite state systems. This approach has been very successful although generalizations are quite difficult. Moreover time and memory complexity is sometimes preclusive for these decidable problems [2];

- Approximation methods for simplifying the verification problem, e.g. by restricting the class of considered properties, which produce manifest results ('"yes", "no") but can be imprecise ("I don't know") although never incorrect or unsound.

Abstract interpretation is a formalization of program analysis based on the last idea. Program analysis algorithms approximate the incomputable collection of all possible behaviors of the program as specified by a standard semantics [6, 3]. The central idea is that of discrete approximation, from above when more behaviors are considered than really possible (as in invariant generation for approximate safety analysis) or dually from below when considering less behaviors than existing ones (as in approximate liveness analysis). The approximation may be static (e.g. Galois connection based approximation) that is made before the analysis is started or preferably dynamic (e.g. widening/narrowing based approximation) when approximations are made during the analysis itself (hence can be better adapted to its cost) [7]. Admittedly, the practical question is to find the proper cost/precision balance.

Generic/parameterized abstract interpreters and hierarchies of abstractions have been designed which help solving this balancing problem. Changing the cost/precision ratio of the analysis does not involve a complete redesign of the analyzer but only a change of modules encapsulating the abstract domain and corresponding operations.

From a theoretical standpoint, the numerous results available in computational complexity are hardly applicable because they concern cost only but neither precision (which might be approached from a probabilistic point of view) nor (economical) benefit. In absence of more theoretical work on this subject, the answer is therefore mostly experimental. For example, following the failure of the maiden flight 501 of the Ariane 5 launcher [12], a recent success story was the static analysis of Ariane's embedded programs to prove the absence of runtime errors [9].

In contrast to simple special-purpose abstract domains (such as bit-vectors in dataflow analysis or boolean functions in strictness analysis), the effective design of wide-scope general-purpose abstract domains involves complex use of both sophisticated data structures and efficient algorithms. An example of such successful numerical abstract domain (in order to approximated a set of vectors of numbers) is the linear relation abstract domain (a set of vectors of numbers is upper approximated by its convex hull) which was originally designed for inferring linear relationships invariantly satisfied by the numerical variables of a program and applied to array bound checking [8]. Linear relation analysis was later used to prove (un)reachability properties of linear hybrid systems [10] (a model involving variables evolving continuously over time) thus showing the wide scope of this abstraction. Despite recent algorithmic progress, this problem still raises problems such as handling polyhedra with floating point arithmetic. This is much more efficient than the use of rational numbers but introduces imprecision, a problem shared with computational geometry algorithmics.

If numerical abstractions are rather well-understood, this is not the case for the approximation of sets of discrete structures (sets of words, trees, graphs, hypergraphs, etc.). Contributions from automata and formal language theory are needed to design e.g. abstract domains that would be applicable to closure analysis of functional languages, pointer analysis of imperative programs, communication analysis of parallel and distributed programs, etc. (see e.g. [14]) whereas these problems are presently considered from quite different and specific points of view.

Program analysis relies on the precise specification of the semantics of programming languages. An informal intuitive understanding being inadequate beyond toy languages. Many semantics at different levels of abstraction/refinement are needed for program analysis so that a unified framework for presenting all these semantics seems indispensable (see [4] for a very first step). The situation is particularly unclear for parallel and distributed programs since the theory of concurrency is far from offering a unified semantic model of concurrency which is directly manageable for program analysis. Moreover considering (computable) approximations of semantics brings in new perspectives for applied semanticians (like numerical analysis for applied mathematicians).

Although the formal derivation of a program analyzer from its specification as an approximation of a semantics is entirely feasible by formal computation (see e.g. a type inference algorithm [5]), this is a formidable task for realistic programming languages which is a real challenge for program specification et formal derivation. Assistance from symbolic and algebraic computation systems would be welcomed, but this involves a shift of interest from the present numerical domains inherited from mathematics to discrete domains.

We hope to have shown that approximate program/system analysis is an active research area offering a wide range of practical and theoretical open problems. Their resolution strongly depends on progress or slight shifts of interest in many other research areas. Substantial progress in program analysis is closely related to global progress in other areas of theoretical computer science. Its main contribution could be a theory of discrete approximation which, we think, is presently missing for non-numerical structures.

References

[1] F. Bourdoncle. Abstract debugging of higher-order imperative languages. In *Proc. PLDI*, pages 46–55. ACM Press, 1993. (document)

[2] E.M. Clarke, O. Grumberg, and D.E. Long. Verification tools for finite-state concurrent systems. In *Decade of concurrency–Reflections and Perspectives*, LNCS 803. Springer-Verlag, 1994. (document)

[3] P. Cousot. Abstract interpretation. *Symposium on Models of Programming Languages and Computation, ACM Comput. Surv.*, 28(2):324–328, 1996. (document)

[4] P. Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *ENTCS*, 6, 1997. URL: `http://www.elsevier.nl/locate/entcs/volume6.html`, 25 pages. (document)

[5] P. Cousot. Types as abstract interpretations, invited paper. In *24th POPL*, pages 316–331, Paris, FRA, jan 1997. ACM Press. (document)

[6] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th POPL*, pages 238–252, Los Angeles, Calif., 1977. ACM Press. (document)

[7] P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation, invited paper. In M. Bruynooghe and M. Wirsing, editors, *Proc. Int. Work. PLILP '92,*, Leuven, BEL, 13–17 aug 1992, LNCS 631, pages 269–295. Springer-Verlag, 1992. (document)

[8] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *5th POPL*, pages 84–97, Tucson, Ariz., 1978. ACM Press. (document)

[9] A. Deutsch, G. Gonthier, and M. Turin. La vérification des programmes d'ariane. *Pour la Science*, 243:21–22, jan 1998. (in French). (document)

[10] N. Halbwachs, Y.E. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design*, 11(2):157–185, August 1997. (document)

[11] N.D. Jones, Gomard C.K., Sestoft P., L.O. (Andersen, and T.) Mogensen. *Partial Evaluation and Automatic Program Generation.* International Series in Computer Science. Prentice-Hall, jun 1993. (document)

[12] J.L. Lions. ARIANE 5 flight 501 failure, report by the inquiry board. `http://www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html`. (document)

[13] J. Rushby. Automated deduction and formal methods. In R. Alur and T.A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, number 1102 in LNCS, pages 169–183, New Brunswick, NJ, July/August 1996. Springer-Verlag. (document)

[14] A. Venet. Automatic determination of communication topologies in mobile systems. In G. Levi, editor, *Proc. SAS '98*, Pisa, ITA, 14–16 sep 1998, LNCS 1503, pages 152–167. Springer-Verlag, 1998. (document)

[15] R. Wilhelm and D. Maurer. *Compiler Design.* Addison-Wesley, 1995. (document)