The Verification Grand Challenge and Abstract Interpretation

Patrick Cousot
École normale supérieure, 45 rue d'Ulm
75230 Paris cedex 05, France
Patrick.Cousot@ens.fr
Visiting the Aeronautics and Astronautics Department
MIT, 77 Massachusetts Avenue Cambridge, MA 02139
cousot@mit.edu

1 Introduction

Abstract Interpretation, is a theory of approximation of mathematical structures, in particular those involved in the semantic models of computer systems [6, 3, 7]. Abstract interpretation can be applied to the systematic construction of methods and effective algorithms to approximate undecidable or very complex problems in computer science.

In particular, abstract interpretation-based static analysis, which automatically infers dynamic properties of computer systems, has been very successful these last years to automatically verify complex properties of real-time, safety critical, embedded systems.

For example, ASTRÉE [1, 2, 8] can analyze mechanically and verify formally the absence of runtime errors in industrial safety-critical embedded control/command codes of several hundred thousand lines of C.

We summarize the main reasons for the technical success of ASTRÉE, which provides directions for application of abstract interpretation to the Verification Grand Challenge [10, 11].

2 The Static Analyzer ASTRÉE

ASTRÉE [1, 2, 8] is a static program analyzer aiming at proving the absence of Run Time Errors (RTE) in programs written in the C programming language. ASTRÉE analyzes structured C programs, without dynamic memory allocation and recursion. This encompasses many synchronous, time-triggered, real-time, safety critical, embedded software programs as found in aerospace, automotive, customer electronics, defense, energy, industrial automation, medical device, rail transportation and telecommunications applications.

ASTRÉE aims at proving that the C programming language is correctly used and that there can be no Run-Time Errors (RTE) during any execution in any environment. This covers:

– Any use of C defined by the international norm governing the C programming language (ISO/IEC 9899:1999) as having an undefined behavior (such as division by zero or out of bounds array indexing);

– Any use of C violating the implementation-specific behavior of the aspects defined by ISO/IEC 9899:1999 as being specific to an implementation of the program on a given machine (such as the size of integers and arithmetic overflow);

– Any potentially harmful or incorrect use of C violating optional user-defined programming guidelines (such as no modular arithmetic for integers, even though this might be the hardware choice);

– Any violation of optional, user-provided assertions (similar to assert diagnostics for example), to prove user-defined run-time properties.

ASTRÉE is sound, automatic, efficient, domain-aware, parametric, modular and precise. More precisely:

– ASTRÉE is *sound* in that it always exhaustively considers all possible run-time errors in all possible program executions and never omit to signal a potential run-time error, a minimal requirement for safety critical software;

– ASTRÉE is fully *automatic*, that is never needs to rely on the user's help such as the decoration of programs with inductive invariants;

– ASTRÉE always terminates and has shown to be *efficient* and to scale up to real size programs as found in the industrial practice;

– Like *general-purpose static analyzers*, ASTRÉE relies on programming language-related properties to point at potential run-time errors. Like *specialized static analyzers*, ASTRÉE puts additional restriction on considered program (e.g. no recursion, no side-effect) and so can take specific program structures into account. Moreover, ASTRÉE is *domain-aware* and so knows facts about application domains that are indispensable to make sophisticated proofs. For example, ASTRÉE takes the logic and functional properties of control/command theory into account as implemented in embedded programs [2, 9];

– ASTRÉE is *parametric* in that the degree of precision of the analysis can be adjusted either manually or mechanically. This means that the performance rate (cost of the analysis/precision of the analysis) can be fully adapted to the needs of its end-users;

– ASTRÉE is *modular*. It is made of pieces (so called *abstract domains*) that can be assembled and parameterized to build application specific analyzers, fully adapted to a domain of application or to end-user needs. Written in OCaml, the modularization of ASTRÉE is made easy thanks to OCaml's modules and functors;

– A consequence of undecidability in fully automatic static analysis is false alarms. Even a high selectivity rate of 1 false alarm over 100 operations with potential run-time errors leaves a number of doubtful cases which may be unacceptable for very large safety-critical or mission-critical software (for example, a selectivity rate of 1% yields 1000 false alarms on a program with 100 000 operations);

  In contrast ASTRÉE, being modular, parametric and domain-aware can be made very *precise* and has shown to be able to produce *no false alarm*, that is fully automated correctness proofs.

The strength of ASTRÉE is that, despite fundamental undecidability limitations, it scales up and can automatically do (or has shown to be easily adaptable by specialists to do) complex proofs of absence of RTE for the considered family of synchronous control/command software. Such proofs are large, complex and subtle, well beyond human capacity, even using provers or proof assistants. This strength comes from a careful, domain-specific design of the abstract interpretation. Any abstraction that would not be able to express and automatically infer, without loss of information, an *inductive* invariant which is necessary to prove absence of RTE for any program in the considered family would inexorably produce false alarms and in practice many, because of cascaded dependencies. Essentially ASTRÉE has demonstrated in practice that for a specific program property (absence of RTE) and a specific family of programs (synchronous control/command C programs) it is possible to find an abstract interpretation of the program which encompasses all necessary inductive proofs.

This strength is also the weakness of ASTRÉE. Since ASTRÉE produces "miracles" on the considered family of properties and programs, end-users would like it to produce very good results on any C program. Obviously this is impossible since the abstractions considered in ASTRÉE will miss the inductive invariants which are out of its precisely defined scope. However, abstractions can be explored outside the current scope and incorporated in the static analyzer.

3 Directions for application of abstract interpretation to the Verification Grand Challenge

In light of the ASTRÉE, we propose a few directions for application of abstract interpretation to verification.

3.1 Program verification

"A program verifier uses automated mathematical and logical reasoning to check the consistency of programs with their internal and external specifications" [11]. Following E.W.D. Dijkstra, there is a clear distinction between the verification or proof of the presence of bugs (that is "testing" or "debugging") from the verification or proof of the absence of bugs (that is "correctness verification" or "verification" for short). Of course the Verification Grand Challenge addresses the *correctness verification* only since the real challenge should be to find the *last* bug.

3.2 Error tracing

Nevertheless, bugs have to be considered in the development process. When an automatic verification system signals an error, it is important to be able to trace the origin of the error, in particular to determine whether it is a bug or a false alarm. Abstract slicing may be useful to trace back the part of the computation which is involved in the bug/false alarm. Finding counter-examples can be extremely difficult, if not impossible, e.g. when tracking the consequences of accumulating rounding errors after hours of floating point computations.

3.3 Program semantics

A program is checked with respect to a semantics that is a formal description of its computations. Numerous semantics have been proposed which differ in the level of abstraction at which they describe computations (e.g. sets of reachable states versus computation histories) and in the method for associating computations to programs (e.g. by induction on an abstract syntax using fixpoints versus using rule-based formal systems). These semantics can be organized in a hierarchy by abstract interpretation [5] so that different analyzers can rely on different semantics which can be formally guaranteed to be coherent, at various levels of abstractions.

In practice, although norms do exist for programming languages like C, they are of little help because too many program behaviors are left unspecified. So one must rely on compilers and machines to know, e.g. the effect of evaluating an arithmetic expressions. Since the Verification Grand Challenge addresses "significant software products", it is clear that methods for defining the semantics of programs are needed, at a level of precision which is compatible with the implementation. An approach could be, like in ASTRÉE, to reject programs for which this compatibility cannot be formally guaranteed. The abstraction methods to do so, might then be part of the programming language semantics.

3.4 Specification

The program semantics restricts the verification to properties that can be expressed in terms of this semantics. The specifications (such as invariance, safety, security, liveness) further restricts the verification process to specific properties. Specifications themselves translate external requirements in terms of program computations.

Specifications cannot be simply be considered as correct, since in practice they are not or only one side of interfaces satisfies the given specifications. Abstract interpretation techniques could be used both to analyze specifications and to check programs for resistance to specification unsatisfaction.

3.5 Specification and verification of complex systems

More generally, specifications refer, especially in the case of embedded systems, to an external world which should be taken into account to prove the correctness of a whole system, not only the program component. Progress has to be made on the abstraction of this external, often physical world, to be compatible with the program interfaces. We envision that abstraction can be applied to the full system (program + reactive environment) although the descriptions of the program and physical part of the system are a quite different nature (e.g. continuous versus discrete). A unification of abstraction in computer science and engineering sciences must be considered to achieve the goal of full system verification.

3.6 Verification of program families

The considered programs to be verified may range from one program (with a finite specific abstraction), to a family of programs with specific characteristics, to a programming language or even a family of programming languages. A broad spectrum verifier is likely to have many customers but also to produce too many false alarms, a recurrent complain of end-users of static analyzers. A finite abstraction can always be found for a given program and specification but discovering this abstraction amounts to making the proof [4], i.e. iteratively computing the weakest inductive argument. To get no false alarm, the consideration of families of programs for which generic, precise and efficient abstractions can be found might be a useful alternative, as was the case in ASTRÉE.

3.7 Required precision of verifiers

Automatic program verification requires the discovery of inductive arguments (for loops, recursion, etc). Proceeding by direct reference to the program semantics (as in refinement-based methods) amounts to the computation of the program semantics restricted to the program specification, which is not a finitary process. Abstraction is therefore necessary but leads to false alarms. The condition for absence of false alarm is that the weakest inductive argument suitable for the proof be expressible without loss of precision in the abstract (including for its transformers in the induction step) [4]. There is obviously no hope to find an abstract domain containing all of such inductive arguments, since this will ultimately amount to include all first-order predicates with arithmetic and one is back to undecidability.

3.8 Abstract assertions

The choice of the form of the abstract assertions depends on the considered family of programs, the nature of the considered specifications and the corresponding necessary inductive arguments. Universal representations (as terms or specific encodings of sets of states), to be used in all circumstances, are likely to be very inefficient. The specific abstract assertions are implemented as abstract domains in ASTRÉE using specific encoding and computer representations that lead to efficient manipulation algorithms. The study of efficient implementation of abstract assertions and efficient algorithms in abstract domains can certainly make significant progress, in particular by considering the domains of applications of programs.

3.9 Application-aware verifiers

ASTRÉE is a program verifier with a very precise scope of application that is of synchronous, real-time control command systems. It can therefore incorporate knowledge about such programs, looking e.g. for ellipsoidal assertions when encountering digital filters [9]. In absence of such domain specific knowledge, a verifier might have to look for polynomial invariants, at a much higher cost.

Among the application domains that have been largely neglected by the verification community are the numerical applications involving intensive floating point computations. To be sound ASTRÉE must perform a rigorous analysis of floating point computations [13]. Further abstractions of this complex semantics are needed.

3.10 Abstract solvers

ASTRÉE uses sophisticated iteration techniques to propagate assertions and perform inductive steps by widening in solvers (see e.g. trace partitioning [12]). A lot of progress can be done on abstract solvers, in particular for generic, parametric and modular ones.

3.11 Combination of abstractions

A verification in ASTRÉE is done by parts, each part corresponding to an abstract domain handling specific abstract assertions, with an interaction between the parts, formalized by the reduced product [7]. So a specific version of ASTRÉE is built by incorporating a choice of abstract domains, which can be program specific, and of the corresponding interactions.

3.12 Modular analyzers

The modular design of ASTRÉE might be a useful approach to the necessity to have specific analyzers adapted to domains of applications and the need for general tools for program verification. One can imagine a large collections of abstract domains and solvers that can be combined on demand to adjust the cost/precision ratio, depending upon the proposed application of the verifier.

3.13 The verified verifier

A recurrent question about ASTRÉE is whether it has been verified and this question is likely to appear for any verifier. A verification has three phases, the computation of an inductive assertion implied by the semantics and the specification which involves resolution of fixpoint inequations, the verification that the assertion is indeed inductive and finally the proof that the inductive assertion implies the specification. All phases are formally specified by abstract interpretation theory. The first phase is indeed the more complex but, from a strict soundness point of view, it does not need to be formally verified. Only the second and third phases of the verifier must be verified, which is simpler. Preliminary work on ASTRÉE shows that this is indeed possible. The verified verifier is indeed part of the Verification Grand Challenge.

3.14 Acceptance and dissemination of static analysis

The dissemination and widespread adoption of formal methods is confronted with economic payoff criteria. Not doing any correctness proof is, at first sight, easier and less expensive. Regulation might be necessary to enforce the adoption of formal methods to produce safer software (e.g. in industrial norms). Static analysis, which has shown to scale up in an industrial context, is a very good candidate. End-users might also be willing to enforce their right for verified software products. The ability to perform automatically static analyzes showing that products are not state of the art might even be a decisive argument to change present-day wide-open laws regarding software reliability.

4 Conclusion

Abstraction, as formalized by Abstract Interpretation is certainly central in the Verification Grand Challenge, as shown by its recent applications, that do scale up for real-life safety critical industrial applications. A Grand Challenge for abstract interpretation is to extend its scope to complex systems, from specification to implementation, not only to the program part, as is presently the case.

References

[1] 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, invited chapter. In T. Mogensen, D.A. Schmidt, and I.H. Sudborough, editors, *The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, LNCS 2566, pages 85–108. Springer, 2002.

[2] 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. In *Proc. ACM SIGPLAN '2003 Conf. PLDI*, pages 196–207, San Diego, CA, US, 7–14 June 2003. ACM Press.

[3] 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, FR, 21 Mar. 1978.

[4] P. Cousot. Partial completeness of abstract fixpoint checking, invited paper. In B.Y. Choueiry and T. Walsh, editors, *Proc. 4^{th} Int. Symp. SARA '2000*, Horseshoe Bay, TX, US, LNAI 1864, pages 1–25. Springer, 26–29 Jul. 2000.

[5] P. Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Theoret. Comput. Sci.*, 277(1—2):47–103, 2002.

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

[7] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *6^{th} POPL*, pages 269–282, San Antonio, TX, 1979. ACM Press.

[8] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The astrée analyser. In M. Sagiv, editor, *Proc. 14^{th} ESOP '2005, Edinburg, UK*, volume 3444 of *LNCS*, pages 21–30. Springer, Apr. 2Ñ-10, 2005.

[9] J. Feret. Static analysis of digital filters. In D. Schmidt, editor, *Proc. 30th ESOP '2004, Barcelona, ES*, volume 2986 of *LNCS*, pages 33–48. Springer, Mar. 27 – Apr. 4, 2004.

[10] C.A.R. Hoare. The verifying compiler, a grand challenge for computing research. *J. ACM*, 50(1):63–69, 2003.

[11] C.A.R. Hoare. The verifying compiler, a grand challenge for computing research. In R. Cousot, editor, *Proc. 6th Int. Conf. VMCAI 2005*, pages 78–78, Paris, FR, 7–19 Jan. 2005. LNCS 3385, Springer.

[12] L. Mauborgne and X. Rival. Trace partitioning in abstract interpretation based static analyzer. In M. Sagiv, editor, *Proc. 14th ESOP '2005, Edinburg, UK*, volume 3444 of *LNCS*, pages 5–20. Springer, Apr. 2Ñ-10, 2005.

[13] A. Miné. Relational abstract domains for the detection of floating-point run-time errors. In D. Schmidt, editor, *Proc. 30th ESOP '2004, Barcelona, ES*, volume 2986 of *LNCS*, pages 3–17. Springer, Mar. 27 – Apr. 4, 2004.