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

1 Introduction

Abstract Interpretation is a theory of approximation of mathematical structures, in particular those involved in the semantic models of computer systems [4,10,11]. Abstract interpretation can be applied to the systematic construction of methods and effective algorithms to approximate undecidable or very complex problems in computer science. The scope of application is rather large e.g. from type inference [5], model-checking [13], program transformation [14], watermarking [15] to context-free grammar parser generation [16].

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 [1,2,3,17,18,25] can analyze mechanically and verify formally the absence of runtime errors in industrial safety-critical embedded synchronous control/command codes of several hundred thousand to one million of lines 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 [22,23].

2 The Static Analyzer ASTRÉE

2.1 Programs Analyzed by ASTRÉE

Astrée [1,2,3,17,18,25] is a static program analyzer aiming at proving the absence of Run Time Errors (RTE) in programs written in the C programming language[1].

Astrée analyzes structured C programs, without side effects in expressions, dynamic memory allocation and recursion. All other features of C are handled including arrays, structures, union types, pointers, pointer arithmetics, etc [31].

[1] C programs are analyzed after macro-expansion.

B. Meyer and J. Woodcock (Eds.): Verified Software, LNCS 4171, pp. 189–201, 2008.

These restrictions encompass 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.

2.2 Specifications Checked by ASTRÉE

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 signed 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.

2.3 Characteristics of ASTRÉE

Astrée is sound, exhaustive, automatic, infinitary, efficient, trace-based, relational, specialized, domain-aware, parametric, modular and precise. More precisely:

- Astrée is *sound* in that it always considers an over-approximation of all possible executions (for example with respect to the rounding of floating-point computations [33]). Since execution is undefined after some runtime errors and may have an unpredictable, implementation dependent effect (e.g. an array bound overflow might destroy code), Astrée may have to assume that execution stops in case of definite runtime error (although in practice it may go on with an "undefined" behavior). If Astrée can prove the absence of any runtime error, the program semantics is well-defined and the analysis is perfectly sound. Otherwise, the results produced by Astrée describe correctly all executions before the first runtime error, if any;
- Astrée is *exhaustive* and considers all possible run-time errors in all possible program executions. Hence Astrée *never* omits 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 program decoration with inductive invariants. It may only happen that a few hypotheses on the range of variation of some inputs or the clock rate and maximal execution time may have to be specified in a separate configuration file to exclude e.g. impossible behaviors of the execution environment;

- Astrée is *infinitary*, that is uses infinite abstractions which are provably more powerful than finite abstract models [12]. This implies that convergence acceleration techniques such as widening/narrowing [10] must be used to enforce termination of fixpoint iterations. Simultaneous widenings in several separate, independently designed, abstract domains are ensured to enforce convergence thanks to an appropriate cooperation between abstract domains [18];
- Astrée always terminates and has shown to be *efficient* and to scale up to real size programs as found in the advanced industrial practice. Observed typical figures are about 1 to 2 hours of computation per hundred thousands lines (although the program size only is not an appropriate measure of the program analysis complexity);
- Astrée is *trace-based*, that is abstracts sets of execution traces as opposed to invariance involving only sets of states. This refined abstraction considerably enhances the precision of the analysis, in particular for functions and procedures [26];
- Astrée is *relational* that is keeps track of relations between the values of program data (variables, fields of structures, array elements, etc). Examples are the octagon abstract domain [28,29,30,32,27] or binary decision trees [3,17]. Contrary to attribute-independent abstract domains (such as intervals [9]), relational abstract domains are expensive (with a polynomial behavior with high degrees, if not exponential [19], in the number of abstract variables). To scale up, Astrée uses program analysis directives (which insertion in the program can be automated by preliminary phases of the analysis) to determine which candidate packs of variables should be separately considered in relational abstractions [3,17].
- 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 restrictions on considered program (e.g. no recursion, no side-effect in expressions, no forward `go to`) and so can take specific program structures into account. For example function and pointer analysis involves no approximation at all, which would not be possible with dynamic memory allocation and recursion;
- 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 [3,20,21];
- Astrée is *parametric* in that the degree of precision of the analysis can be adjusted either manually (through parameters or directives in the program text) or mechanically (by automatic insertion of the directives by preliminary analysis phases). 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 separate parts (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* (and even *no alarm* after minor modifications for some critical programs), that is fully automated correctness proofs.

2.4 Program Verification with ASTRÉE

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, even more than the program itself, whence well beyond human capacity, even using provers or proof assistants.

This strength comes from a careful, domain-specific design of the abstract interpretation. Any imprecise 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. On the other hand, an abstraction that would be too precise for the objective of proving absence of RTE in the considered family of programs would lead to excessive computational and memory costs. 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 at reasonable costs.

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, new abstractions can be explored outside the current scope of Astrée and easily 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" [23]. 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 [38,39]. Then constraint solving techniques can help finding an actual counter-example. However, 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 [7] 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, linkers, loaders and machines to know, e.g. the exact 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 for absence of side effects in expressions, 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, the restricted language being a subset of a larger existing language where undefined behaviors have been excluded. Such programming norms limiting the use of obscure, error-prone and/or non-portable features of programming languages tend to be more widely accepted in practice.

3.4 Compilation

That the semantics of a programming language precisely reflects program executions depends upon the correctness of the translation into machine code. So that verification of the compiler [24], of the object code [35,36] or of the translation [37] must be part of the complete verification process.

3.5 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. Thus it is necessary to define adequate specification languages, their semantics with respect to that of the programming language. This ranges from implicit internal specifications (like absence of runtime errors as defined by the semantics of the programming language) to arbitrary complex specification languages.

Specifications cannot be simply be considered as correct, since in practice they are not or e.g. 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.

Finally, the analysis of the specification, should be checked to remain valid in the implementation, e.g. by reanalysis of the program or by translation validation.

3.6 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 [8].

3.7 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 [6], 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.

3.8 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) [6]. There is obviously no hope to find an abstract domain containing all of such inductive arguments, since this will ultimately amount to include e.g. all first-order predicates with arithmetic and one is back to undecidability.

3.9 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.

3.10 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 [20]. In absence of such domain specific knowledge, a verifier might have to look for costly nonlinear invariants.

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 [30]. Further abstractions of this complex semantics are needed.

3.11 Abstract Solvers

Astrée uses sophisticated iteration techniques to propagate assertions and perform inductive steps by widening in solvers (see e.g. trace partitioning [26]). A

lot of progress can still be done on abstract solvers, in particular for generic, parametric, modular and parallel [34] ones.

3.12 Combination of Abstractions

A verification in Astrée is done by parts, each part corresponding to a separate abstract domain handling specific abstract assertions, with an interaction between the parts, formalized by the reduced product [11]. 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 [18].

3.13 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.

3.14 User Interface

Static analyzers like Astrée yield extremely complex informations on program executions that the end-user may want to understand. For that purpose, a user-interface is needed to present internal information in understandable form, at different levels of abstractions.

3.15 Verifier Infrastructure

Once a programming, a specification and a user-interface language have been chosen, and their semantics defined, a static analyzer has to provide computer representations of these languages for the purpose of the program analysis, the specification checking and the report of the results to the end-user. Despite the difficulty to design such a general-purpose infrastructure, it should be sharable between different verifiers to accelerate experimentations and developments.

3.16 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,

1. the computation of an inductive assertion implied by the semantics and the specification which involves resolution of fixpoint inequations,
2. the verification that the assertion is indeed inductive, and
3. 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.

3.17 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 permissive 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 design to implementation.

References

1. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The Astrée Static Analyzer, `http://www.astree.ens.fr/`
2. 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. In: Mogensen, T.Æ., Schmidt, D.A., Sudborough, I.H. (eds.) The Essence of Computation. LNCS, vol. 2566, pp. 85–108. Springer, Heidelberg (2002)
3. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A static analyzer for large safety-critical software. In: Proceedings of the ACM SIGPLAN '2003 Conference on Programming Language Design and Implementation (PLDI), San Diego, California, United States, June 7-14, 2003, pp. 196–207. ACM Press, New York (2003)
4. Cousot, P.: Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique de programmes (in French). In: Thèse d'État ès sciences mathématiques, Université scientifique et médicale de Grenoble, Grenoble, France, March 21, (1978)
5. Cousot, P.: Types as abstract interpretations. In: Conference Record of the Twentyfourth Annual ACM SIGPLAN--SIGACT Symposium on Principles of Programming Languages, Paris, France, January 1997, pp. 316–331. ACM Press, New York (1997)
6. Cousot, P.: Partial completeness of abstract fixpoint checking. In: Choueiry, B.Y., Walsh, T. (eds.) SARA 2000. LNCS (LNAI), vol. 1864, pp. 1–25. Springer, Heidelberg (2000)

7. Cousot, P.: Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. Theoretical Computer Science 277(1—2), 47–103 (2002)
8. Cousot, P.: Integrating physical systems in the static analysis of embedded control software. In: Yi, K. (ed.) APLAS 2005. LNCS, vol. 3780, pp. 135–138. Springer, Heidelberg (2005)
9. Cousot, P., Cousot, R.: Static determination of dynamic properties of programs. In: Proceedings of the Second International Symposium on Programming, Paris, France, pp. 106–130. Dunod, Paris, France (1976)
10. Cousot, P., Cousot, R.: 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, Los Angeles, California. United States, pp. 238–252. ACM Press, New York (1977)
11. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Conference Record of the Sixth Annual ACM SIGPLAN--SIGACT Symposium on Principles of Programming Languages, San Antonio, Texas, pp. 269–282. ACM Press, New York (1979)
12. Cousot, P., Cousot, R.: Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In: Bruynooghe, M., Wirsing, M. (eds.) PLILP 1992. LNCS, vol. 631, pp. 269–295. Springer, Heidelberg (1992)
13. Cousot, P., Cousot, R.: Temporal abstract interpretation. In: Conference Record of the Twentyseventh Annual ACM SIGPLAN--SIGACT Symposium on Principles of Programming Languages, Boston, Massachusetts, United States, January 2000, pp. 12–25. ACM Press, New York (2000)
14. Cousot, P., Cousot, R.: Systematic design of program transformation frameworks by abstract interrpetation. In: Conference Record of the Twentyninth Annual ACM SIGPLAN--SIGACT Symposium on Principles of Programming Languages, Portland, Oregon, United States, January 2002, pp. 178–190. ACM Press, New York (2002)
15. Cousot, P., Cousot, R.: An abstract interpretation-based framework for software watermarking. In: Conference Record of the Thirtyfirst Annual ACM SIGPLAN--SIGACT Symposium on Principles of Programming Languages, Venice, Italy, January 14–16, 2004, pp. 173–185. ACM Press, New York (2004)
16. Cousot, P., Cousot, R.: Grammar analysis and parsing by abstract interpretation. In: Reps, T., Sagiv, M., Bauer, J. (eds.) Program Analysis and Compilation, Theory and Practice. LNCS, vol. 4444, pp. 175–200. Springer, Heidelberg (2007)
17. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The Astrée analyser. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 21–30. Springer, Heidelberg (2005)
18. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: Combination of abstractions in the Astrée static analyzer. In: Okada, M., Satoh, I. (eds.) Eleventh Annual Asian Computing Science Conference, ASIAN 06, LNCS, Tokyo, Japan, December 6–8, 2006. pp. 6–8. Springer, Berlin (to appear, 2006)
19. Cousot, P., Halbwachs, N.: 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, Tucson, Arizona, pp. 84–97. ACM Press, New York (1978)
20. Feret, J.: Static analysis of digital filters. In: Schmidt, D. (ed.) ESOP 2004. LNCS, vol. 2986, pp. 33–48. Springer, Heidelberg (2004)
21. Feret, J.: The arithmetic-geometric progression abstract domain. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 42–58. Springer, Heidelberg (2005)

22. Hoare, C.A.R.: The verifying compiler, a grand challenge for computing research. Journal of the Association for Computing Machinary 50(1), 63–69 (2003)
23. Hoare, C.A.R.: The verifying compiler, a grand challenge for computing research. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, p. 78. Springer, Heidelberg (2005)
24. Leroy, X.: Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In: Conference Record of the Thirtythird Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, pp. 42–54. ACM Press, New York (2006)
25. Mauborgne, L.: Astrée: Verification of absence of run-time error. In: Jacquart, P. (ed.) Building the Information Society, ch. 4, pp. 385–392. Kluwer Academic Publishers, Dordrecht (2004)
26. Mauborgne, L., Rival, X.: Trace partitioning in abstract interpretation based static analyzer. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 5–20. Springer, Heidelberg (2005)
27. Miné, A.: The `Octagon` abstract domain library. `http://www.di.ens.fr/~mine/oct/`
28. Miné, A.: A new numerical abstract domain based on difference-bound matrices. In: Danvy, O., Filinski, A. (eds.) PADO 2001. LNCS, vol. 2053, pp. 155–172. Springer, Heidelberg (2001)
29. Miné, A.: A few graph-based relational numerical abstract domains. In: Hermenegildo, M.V., Puebla, G. (eds.) SAS 2002. LNCS, vol. 2477, pp. 117–132. Springer, Heidelberg (2002)
30. Miné, A.: Relational abstract domains for the detection of floating-point run-time errors. In: Schmidt, D. (ed.) ESOP 2004. LNCS, vol. 2986, pp. 3–17. Springer, Heidelberg (2004)
31. Miné, A.: Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In: Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems, LCTES '2006, June 2006, pp. 54–63. ACM Press, New York (2006)
32. Miné, A.: The octagon abstract domain. Higher-Order and Symbolic Computation 19, 31–100 (2006)
33. Miné, A.: Symbolic methods to enhance the precision of numerical abstract domains. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 348–363. Springer, Heidelberg (2005)
34. Monniaux, D.: The parallel implementation of the Astrée static analyzer. In: Yi, K. (ed.) APLAS 2005. LNCS, vol. 3780, pp. 86–96. Springer, Heidelberg (2005)
35. Rival, X.: Abstract interpretation based certification of assembly code. In: Zuck, L.D., Attie, P.C., Cortesi, A., Mukhopadhyay, S. (eds.) VMCAI 2003. LNCS, vol. 2575, pp. 41–55. Springer, Heidelberg (2002)
36. Rival, X.: Invariant translation-based certification of assembly code. International Journal on Software and Tools for Technology Transfer 6(1), 15–37 (2004)
37. Rival, X.: Symbolic transfer functions-based approaches to certified compilation. In: Conference Record of the Thirtyfirst Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Venice, Italy, pp. 1–13. ACM Press, New York (2004)
38. Rival, X.: Abstract dependences for alarm diagnosis. In: Yi, K. (ed.) APLAS 2005. LNCS, vol. 3780, pp. 347–363. Springer, Heidelberg (2005)
39. Rival, X.: Understanding the origin of alarms in Astrée. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, pp. 303–319. Springer, Heidelberg (2005)

A Discussion after Patrick Cousot's Presentation

Wolfgang Paul

You said that we couldn't make a formal semantics of a real programming language. If we would believe this, I would suggest we go home immediately, because then the thing has failed. Your example was IEEE floating-point arithmetic incorporated into C or a suitable subset, which is the only thing that is used in industry. And incorporating a floating-point arithmetic definition into a C semantics is no problem whatsoever, and I point you to people in France who can do it easily for you.

Patrick Cousot

No, my point was correct. There is no semantics of C, because the semantics says: If there is something going wrong, the behavior is undefined. But the program behavior is not undefined. It does something; it goes on after the error. If you make an overflow, it will do something, and maybe destroy the program. So, if you make an analysis, you have two things: Either you take the semantics where it can do anything and you get no information, or you make an hypothesis that it stops when it is undefined. But then, the result of your analysis that says that you have one error is not true. It is: one error, if you stop the program at the first error. So, when there is zero error, you can prove that the two things are well defined. But, for example, for the floating-point, if you run the program on an Intel and on a Macintosh, the result is completely different, even if it is always correct in the analyzer, but it is different. And one reason is, Intel computes 80 bits [within the processor] and the IEEE [standard] is 64 [bits]. So, if you have 80 bits in the register and you put it [somewhere] in memory, it will become 64 bits, but if you do not put it in memory, it is all done in 80 bits. The result is different and maybe incorrect.

Wolfgang Paul

You are completely right that things are not the way, they should be, and I am aware of all these things. But it does not mean that they cannot be fixed. If you say, they cannot be fixed inherently, then we must go home, and therefore, you should not say this.

Patrick Cousot

We fix it by checking that the assembler satisfies these bit things *[end of sentence missing].*

[Session chair Ganesan Ramalingam asks to postpone further discussion to the discussion session, and notes that Roderick Chapman had raised his hand for quite some time.]

Roderick Chapman, Praxis High Integrity Systems

My hand is nearly hitting the ceiling. Three things. Firstly, this is incredibly impressive work, and as an industrial application it's brilliant.

Secondly, in the discussion here, we must distinguish between the semantics of a programming language as is defined by ISO in the standard language definition in all their glory and the semantics of a programming language as implemented by one compiler, that is what you call a dialect of a language, and it's a completely different beast.

Thirdly, the formal semantics of SPARK, which are precise and very nearly complete, were constructed and written down by my colleagues almost ten years ago. And it is done, we did it, and you can have it, it is publicly available. So, there is an industrially available, used, formally defined programming language, and some people in this room, including Jim [Woodcock], reviewed those semantics, and they are available. Thank you.

Patrick Cousot

This was a comment *[rather than a question]*. I agree.