Proving the Absence of Run-Time Errors in Safety-Critical Avionics Code

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

ABSTRACT

We explain the design of the interpretation-based static analyzer ASTRÉE and its use to prove the absence of run-time errors in safety-critical codes.

Categories and Subject Descriptors

D.2.4 [**Software**]: Software Engineering—*Software/Program Verification*; F.3.1, F.3.2 [**Theory of Computation**]: Logics and Meanings of Programs—*Specifying and Verifying and Reasoning about Programs, Semantics of Programming Languages*

General Terms

Reliability, Languages, Verification

Software engineering has focussed on methods for designing larger and larger computer applications but software quality has not followed this dramatic progression. Poor software quality is not acceptable in safety and mission critical applications. An avenue is therefore opened for formal methods which are product-based (as opposed to development process-based). Abstract interpretation [3, 5] is a formal method for software verification which is having significant industrial applications, in particular in the context of avionics [8, 15, 26, 30].

We explain program correctness proofs by static analysis and the design of a static analyzer by abstract interpretation of a program semantics. This is illustrated with the ASTRÉE abstract-interpretation-based static analyzer which, after six years of academic development, is progressing towards industrial acceptance for the validation of software intensive applications such as safety-critical avionics code.

The ASTRÉE static analyzer [8, 1, 2, 9, 16] aims at formally proving the *absence of runtime errors* in C programs (such as buffer overruns, pointer misuses, arithmetic overflows, etc including additional user requirements). It is specialized for synchronous, time-triggered, real-time, safety critical, *embedded software* as found in earth transportation, nuclear energy, medical instrumentation and aerospace applications. Such programs have no recursion, no dynamic memory allocation and no library call (the only undefined function is the synchronization on the clock) but otherwise may have all the difficulties found in the analysis of C programs (including pointer arithmetics).

The analysis performed by ASTRÉE is *static* that is based on source code inspection only (as opposed to *dynamic* analysis involving a recording of program execution for online or later offline inspection or replay). In theory, static analysis is an exhaustive exploration of a *semantic model* of the full program execution space. The hard problems are to fully *automate* the analysis and to *scale up* to industrial applications, a general grand challenge for all formal methods.

ASTRÉE *always terminates* (even if actual executions do not), is *efficient* (requiring typically one to two hours of computations per 100 000 LOCs) and *does scale up* (to millions of LOCs). A parallel implementation enhances the performances (although slow communication costs may rapidly surpass the computing power gained by increasing the number of processors) [25].

The static analysis overestimates the *program trace semantics* that is set of all run-time execution traces of the program with unknown dynamic inputs (which ranges may be limited by an optional configuration file) for an execution duration which may be bounded or not (as indicated in the configuration file). None of the actual executions of the program being omitted, abstract-interpretation-based static analyzers have no false negatives hence are *sound* by design. As opposed to bug-finders, static analyzers will never omit to signal an error that can appear at runtime in some execution environment [11].

ASTRÉE is designed according to the *theory of abstract interpretation* [3, 5].

The *correctness proof* has two phases. In the first *analysis phase*, the program trace semantics is computed iteratively. From a purely mathematical point of view, the set of all execution traces can in principle be formally constructed starting from initial states, then extending iteratively the partial traces from one state to the next one according to the program transition steps until termination on final or error states or passing to the limit for infinite traces (corresponding to non-terminating executions). The *verification phase* then checks that none of these execution traces can reach a state in which a runtime error can occur.

In general this set of traces of interest is neither computer-representable nor computable (by undecidability). Abstract interpretation exploits the facts that an *overapproximation* of the program set of traces is sound: when considering more possibilities, no actual execution can ever be omitted. The theory is used to design sound approximations of the mathematical structures involved in the formal description of this set of traces. This includes an *iterator* for approximating the step by step iterative computation of traces [5] and *ab-*

stract domains representing the effect of program steps and passage to the limit (widening/narrowing [5]).

The first abstraction underlying ASTRÉE is *trace partitioning* [17]. It collects at each program point a set of trace suffixes leading to that program point. This is therefore a reachability analysis enriched by some history of the computations leading to each state. It follows that information about the execution order and the concrete data flows and control paths is not completely lost. The partitioning criterion is based on data and control and corresponds to a case analysis starting and finishing on demand, according to partitioning directives inserted in the program. Further analyzes have been developed to automate the insertion of these partitioning directives directly by the analyzer according to criteria dictated by the program properties to be proved.

Further *abstractions* are needed to get computer-representable and computable overapproximations of the partitioned sets of trace suffixes leading to a program point.

A classical general-purpose non-relational abstraction is involved in the *interval abstract domain* [4]. The interval abstraction collects for each program point and each numerical variable a lower bound and an upper bound of the values of this variable along all states of all execution traces of the program. Interval analysis is *non-relational* in that knowing an overapproximation of the interval of variation of the numerical values computed by the program does not provide any information on how these values do relate at runtime.

Most other general-purpose and domain-specific abstractions are *relational* and allow e.g. for the discovery of invariant relations between values computed by the program (e.g. values of program variables at a program point on a subset of the traces).

An example is the *octagon abstract domain* [19, 20, 23] to discover invariants of the form $\pm x \pm y \leq c$ where x and y denotes values computed by the program and c is a numerical constant discovered by the analysis. For floats, relational analyses must take rounding errors into account which is challenging but was solved in [21].

Another example of relational abstraction is the *decision tree abstract domain* [2, Sect. 6.2.4] to make different abstractions according to the values of variables taking finitely many values (e.g. booleans).

A last example of relational abstraction between values at different time instants is the *symbolic abstract domain* to abstract a set of traces by a linear symbolic expression relating initial and final values of numerical variables along the traces (and again, cumulating rounding errors for floating point computations) [24].

All these abstract domains are *parameterized* (e.g. maximal height of decision trees, maximal size of symbolic formulæ) to balance the cost/precision ratio. Some parameterizations have been *automated* (such as the determination of the packs of variables for which octagonal relations should be computed) according to criteria dictated by the program properties to be proved.

Besides these general-purpose abstractions, ASTRÉE has *domain specific abstractions* which are required to cope specifically with (synchronous) control command/applications.

The control of physical devices require floating-point computations which are subject to rounding errors which cumulate over time but must be proved to be bounded (e.g. by linear or exponential functions of the time). The *arithmetic-geometric progression abstract domain* is used for that purpose and more generally to approximate a set of traces by a time-dependent bound for the range of each variable (computation time can be bounded for some applications as specified in the configuration file). Another example is the *filter abstract domain* [13] to handle the digital filters used to smooth the variations of input data.

Following a fundamental result of abstract interpretation [7], ASTRÉE uses *infinitary abstract domains* (using finite symbolic representations of infinite mathematical objects) as opposed to finite abstractions which are provably less expressive. A counterpart is that the finite (and rapid) convergence of iterative computations must enforced using widening/narrowing *convergence acceleration techniques* [4, 5].

The use of many independent abstractions is necessary to master the design complexity of static analyzers. They can be easily extended or simplified by including or eliminating abstract domains. Typically, about twenty to thirty abstract domains are used in ASTRÉE. A counterpart of this *independent abstractions design principle* is that the cooperation between abstract domains must be organized [6]. An example is the *reduced product* [6]: information provided by an abstract domain (e.g. congruence information) can enhance the precision of another abstract domain (e.g. reduction of the bounds in interval analysis). The cooperation between abstract domains in ASTRÉE is described in [10]. It ensures interdomain reductions while preserving the convergence of widening under extension by new abstractions.

Because ASTRÉE overapproximates possible executions it is *incomplete* that is subject to false positives or *false alarms*. It may signal a potential error for an abstract execution corresponding to no actual concrete execution of the program. ASTRÉE was designed to be able to *tune the precision of the analysis* (by parametrization, [automated] analysis directives and addition of new abstract domains). This requires the analysis of the *origin of alarms* (either a program error for true alarms or an imprecision in the analysis for false alarms). This analysis is presently manual and should be automated [28, 27].

Thanks to the ability to reach 0 false alarm at a reasonable cost, ASTRÉE is now in industrial use for safety critical control/command programs in avionics [12, 29].

Such programs are generally mostly generated automatically from high-level specifications languages like SCADE$^{\text{TM}}$ or SIMULINK$^{\text{TM}}$ whence avoid tricky uses of pointer arithmetic and `union`. The initial memory model of ASTRÉE was therefore simple with variables, arrays, structures, pointers and aliases excluding untyped operations such as pointer arithmetics and overlapping unions. This simple memory model does not cover manually written, less critical applications, such as telecommunications. For this family of applications, ASTRÉE has been extended to cope with pointer arithmetic and `union` [22].

As the scope of application of ASTRÉE widens, we are faced with new challenges including complex data structures, modular analysis and parallel programs.

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. T. Mogensen, D. Schmidt, and I. Sudborough (Eds.), *The Essence of*

Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones, LNCS 2566, pp. 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. *Proc. ACM SIGPLAN '2003 Conf. PLDI*, pp. 196–207, San Diego, 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 (in French)*. Thèse d'État ès sciences mathématiques, University of Grenoble, 21 Mar. 1978.

[4] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. *Proc. 2nd Int. Symp. on Programming*, pp. 106–130, Paris, 1976. Dunod.

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

[6] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. *6th POPL*, pp. 269–282, San Antonio, 1979. ACM Press.

[7] P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation, invited paper. M. Bruynooghe and M. Wirsing (Eds.), *Proc. 4th Int. Symp. PLILP '92*, Leuven, LNCS 631, pp. 269–295, Springer, 26–28 Aug. 1992.

[8] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The Astrée Static Analyzer. `http://www.astree.ens.fr/`.

[9] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The Astrée analyser. M. Sagiv (Ed.), *Proc. 14th ESOP '2005*, Edinburg, LNCS 3444, pp. 21–30, Springer, 2–10 Apr. 2005.

[10] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Combination of abstractions in the Astrée static analyzer, invited paper. M. Okada and I. Satoh (Eds.), *11th ASIAN 06*, Tokyo, 6–8 Dec. 2006. LNCS, Springer. To appear.

[11] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Varieties of static analyzers: A comparison with Astrée, invited paper. M. Hinchey, H. Jifeng, and J. Sanders (Eds.), *Proc. 1st TASE '07*, pp. 3–17, Shanghai, 6–8 June 2007. IEEE Press.

[12] D. Delmas and J. Souyris. Astrée: from research to industry. G. Filé and H. Riis-Nielson (Eds.), *Proc. 14th Int. Symp. SAS '07*, Kongens Lyngby, LNCS 4634, Springer, 22–24 Aug. 2007.

[13] J. Feret. Static analysis of digital filters. D. Schmidt (Ed.), *Proc. 30th ESOP '2004*, Barcelona, LNCS 2986, pp. 33–48, Springer, Mar. 27 – Apr. 4, 2004.

[14] J. Feret. The arithmetic-geometric progression abstract domain. R. Cousot (Ed.), *Proc. 6th Int. Conf. VMCAI 2005*, Paris, LNCS 3385, pp. 42–58, Springer, 17–19 Jan. 2005.

[15] É. Goubault, M. Martel, and S. Putot. Asserting the precision of floating-point computations: a simple abstract interpreter. D. Le Métayer (Ed.), *Proc. 11th ESOP '2002*, Grenoble, LNCS 2305, pp. 209–212, Springer, 8–12 Apr. 2002.

[16] L. Mauborgne. Astrée: Verification of absence of run-time error. P. Jacquart (Ed.), *Building the Information Society*, ch. 4, pp. 385–392. Kluwer Acad. Pub., 2004.

[17] L. Mauborgne and X. Rival. Trace partitioning in abstract interpretation based static analyzer. M. Sagiv (Ed.), *Proc. 14th ESOP '2005*, Edinburg, LNCS 3444, pp. 5–20, Springer, 2–10 Apr. 2005.

[18] A. Miné. The `Octagon` abstract domain library. `http://www.di.ens.fr/~mine/oct/`.

[19] A. Miné. A new numerical abstract domain based on difference-bound matrices. O. Danvy and A. Filinski (Eds.), *Proc. 2nd Symp. PADO '2001*, Århus, LNCS 2053, pp. 155–172, Springer, 21–23 May 2001.

[20] A. Miné. A few graph-based relational numerical abstract domains. M. Hermenegildo and G. Puebla (Eds.), *Proc. 9th Int. Symp. SAS '02*, Madrid, LNCS 2477, pp. 117–132, Springer, 2002.

[21] A. Miné. Relational abstract domains for the detection of floating-point run-time errors. D. Schmidt (Ed.), *Proc. 30th ESOP '2004*, Barcelona, LNCS 2986, pp. 3–17, Springer, Mar. 27 – Apr. 4, 2004.

[22] A. Miné. Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. *Proc. LCTES '2006*, pp. 54–63, ACM Press, June 2006.

[23] A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19:31–100, 2006.

[24] A. Miné. Symbolic methods to enhance the precision of numerical abstract domains. E. Emerson and K. Namjoshi (Eds.), *Proc. 7th Int. Conf. VMCAI 2006*, Charleston, LNCS 3855, pp. 348–363, Springer, 8–10 Jan. 2006.

[25] D. Monniaux. The parallel implementation of the Astrée static analyzer. K. Yi (Ed.), *Proc. 3rd APLAS '2005*, Tsukuba, LNCS 3780, pp. 86–96, Springer, 3–5 Nov. 2005.

[26] F. Randimbivololona, J. Souyris, and A. Deutsch. Improving avionics software verification cost-effectiveness: Abstract interpretation based technology contribution. *Proceedings DASIA 2000 – DAta Systems In Aerospace*, Montreal. ESA Publications, 22–26 May 2000.

[27] X. Rival. Abstract dependences for alarm diagnosis. K. Yi (Ed.), *Proc. 3rd APLAS '2005*, Tsukuba, LNCS 3780, pp. 347–363, Springer, 3–5 Nov. 2005.

[28] X. Rival. Understanding the origin of alarms in Astrée. C. Hankin and I. Siveroni (Eds.), *Proc. 12th Int. Symp. SAS '05*, London, LNCS 3672, pp. 303–319, Springer, 7–9 Sep. 2005.

[29] J. Souyris. Industrial experience of abstract interpretation-based static analyzers. P. Jacquart (Ed.), *Building the Information Society*, ch. 4, pp. 393–400. Kluwer Acad. Pub., 2004.

[30] S. Thesing, J. Souyris, R. Heckmann, F. Randimbivololona, M. Langenbach, R. Wilhelm, and C. Ferdinand. Abstract interpretation-based timing validation of hard real-time avionics software. *Proc. Int. Conf. DSN 2003*, San Francisco, pp. 625–634. IEEE Press, 22–25 June 2003.