ASTRÉE: VERIFICATION OF ABSENCE OF RUN-TIME ERROR*

Laurent Mauborgne
École Normale Supérieure
Paris, France
astree@ens.fr
http://www.astree.ens.fr

Keywords: Static analysis, abstract interpretation, verification, critical software.

1. Introduction

Everybody knows about failure problems in software: it is an admitted fact that most large software do contain bugs. The cost of such bugs can be very high for our society and many methods have been proposed to try to reduce these failures. While merely reducing the number of bugs may be economically sound in many areas, in critical software (such as found in power plants or aeronautics), no failure can be accepted.

In order to achieve an unfailing critical software, industrials follow very strict production patterns and must also certify the absence of errors through state-of-the-art verification. When old verification methodologies became intractable in time and cost due to the growth of code size, the Abstract Interpretation Team[2] of École Normale Supérieure started developing Astreé [Blanchet et al., 2002]. The object of Astrée is the automatic discovery of all potential errors of a certain class for critical software. As most critical software don't (or won't) have any error, the main challenge was to be exhaustive and very selective, that is yielding few or no false alarms on the software, so as to reduce the cost of verifying those alarms.

In this paper, we show how Astrée is based on sound approximations of the semantics of C programs, tailored to be very accurate on a class of embedded

*Work supported in part by the Réseau National de recherche et d'innovation en Technologies Logicielles (RNTL) exploratory 2002 project ASTRÉE.

[2]The team that developed Astrée is composed of Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux and Xavier Rival. They are supported by CNRS, École Normale Supérieure and École Polytechnique.

synchronous critical software. In section 5, we describe how these abstractions can also be applied or augmented to deal with a wider class of critical software.

2. Related Work

The first method which was used to try and give some confidence in the absence of errors in programs was testing. It consists in running a program on a set of inputs and checking that it behaves as intended. For critical software, the coverage of the tested inputs should be very high in order to achieve a high confidence. The number of possible inputs for real time embedded systems is nearly infinite, so testing is at the same time very expensive (hundreds of man years) and not fully satisfying as some unwanted behaviors may have escaped detection by testing.

Because testing is so expensive, one can use bug finders which will detect some common programming mistakes or report on suspicious codes. Such programs are usually fully automatic, so their cost is very low. They do find bugs in many codes, but they don't give good results on critical softwares: they report too many false alarms and they may overlook some unpredicted bugs.

Formal methods on the other hand can give exhaustive results. Some of them can prove very complex properties of the software, but usually at the cost of heavy human interaction and expertise. This is the case for proof assistants, which may be useful on small parts of the code but cannot scale to full systems. Other formal methods can be more automated, such as software model checkers.

The main problem of many formal methods based tools is that they perform the proof on a model of the code. In order to be tractable, this model cannot be too big, so either they are restricted to a small part of the code or they are restricted to some aspects of the semantics of the code. In general, models concentrate on the logical design and forget about abstruse machine implementation aspects of the software. In the case of critical software, where the logical design is well-mastered, potential errors are more likely to lurk in the machine implementation (such as the rounding errors introduced by floating-point arithmetics).

The theory of abstract interpretation [Cousot, 1978] makes it possible to analyze the actual semantics of real programs while performing sound abstractions to give correct results. Some industrial code analyzers are based on this theory and they give exhaustive results. So far, they discovered bugs in some applications, but they have not been precise enough on critical software: they yield too many false alarms to be useful. The cost of formally proving that all these alarms are indeed false is way too high even for a selection rate[3] of 1%.

[3]The selection rate is the ratio (number of lines with alarms)/(total number of lines).

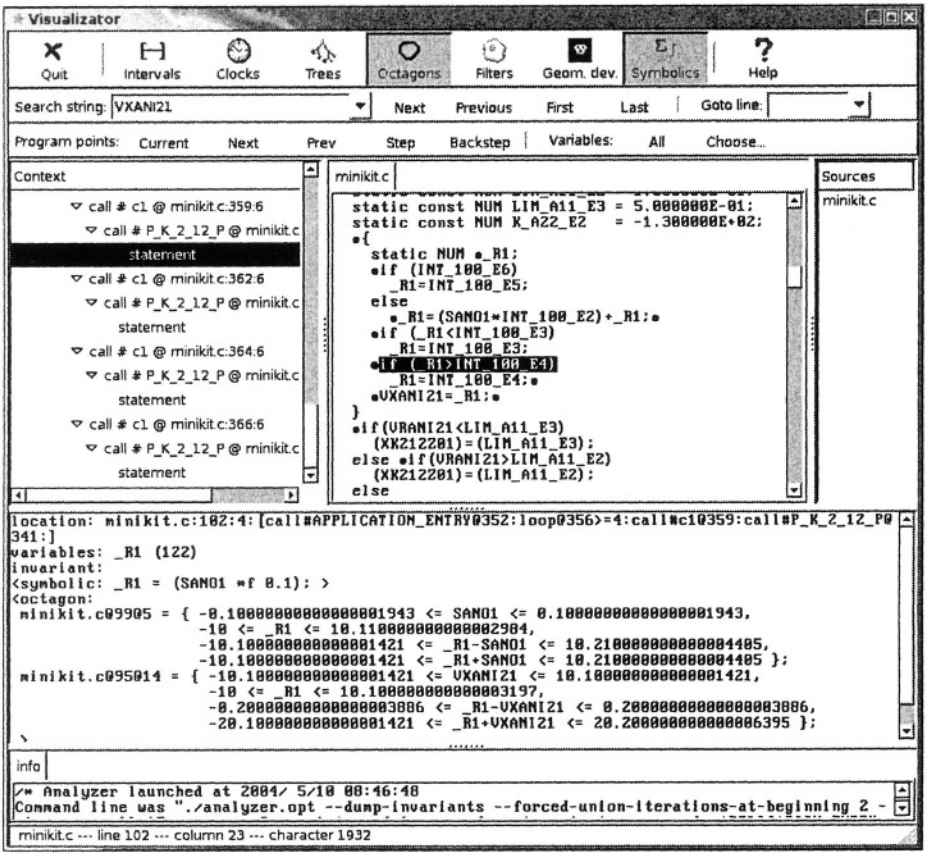

Figure 1. An example visualization of the results of Astrée.

3. What Astrée Does

Astrée is a static analyzer that automatically computes supersets of the possible values in synchronous C programs at every program points. Thus, if Astrée does not report any bad behavior, it proves that no such behavior can happen whatever the inputs of the C program.

Errors detected by Astrée

Once having a superset of the possible values of all program variables at each program point, Astrée can automatically report on a number of errors. The kind of errors which are currently reported by Astrée stems from the first end-user requirements. They wanted to see what could be proved without going through the expensive process of producing formal specifications. The least

one can expect from a critical software is that the code never produces fatal errors, such as divisions by zero. Another common requirement is that the language is never used in cases where the result is stated as "undefined" in the norm of the language [JTC 1/SC 22, 1999]. For example, this is the case of out-of-bound array accesses, or integer overflows.

The errors which are currently reported are:

- out-of-bound array accesses,
- integer division by zero,
- floating point operations overflows and invalid operations (resulting in IEEE floating values Inf and NaN),
- integer arithmetics wrap around behavior (occurring mainly in overflows),
- casts that result in wrap around operations (when the target type is too small to contain a value).

In addition, Astrée can use some user-defined known facts and report on arbitrary user defined assertions (written in C) on the software.

Some Characteristics of Astrée

Astrée was developed to prove the absence of run-time errors for a specific class of synchronous C programs. As expected, it will be quite efficient and precise on the difficulties raised by this class of programs and may be weak on other aspects of the language.

One restriction of the class of C programs for which Astrée was originally designed is that it does not contain any dynamic memory allocation, string manipulation and very restricted pointers. That allows for a fast and precise memory analysis which would not be possible otherwise.

On the other hand, the class of analyzed C programs contains large programs (hundreds of thousands of lines of code), with a huge number of global interdependent variables (about 10 000 for a 100 000 lines program). This makes it hard to be efficient and precise, and specific algorithms and heuristics have been developed to keep the complexity of Astrée low (not far above linear in the number of lines of codes and the number of global variables).

As is necessary for many critical software, Astrée deals well with complex control using thousands of boolean variables. In addition, Astrée makes a sound analysis of floating values computations (as described in [IEEE Computer Society, 1985]), taking into account all possible rounding errors [Miné, 2004]. Astrée is even able to prove tight invariants for a variety of numerical filters implemented with floating numbers [Feret, 2004].

4. Inside Astrée

In order to understand what Astrée proves and how to use it, we describe the basic techniques used in the analyzer.

How Sets of Values can be Approximated

Astrée is a static analyzer based on abstract interpretation [Cousot and Cousot, 1979]. Following this theory, Astrée will proceed by approximating the set of all possible inputs into a symbolic representation. Then the program to analyze will be interpreted on this set of values, approximating each basic instruction when necessary to keep the sets of values representable. Approximation mechanisms are also necessary to find the sets of all possible values at a given point inside a loop, as the problem of finding the exact set is in general undecidable. The main mechanism is the so-called widening which allows extrapolating this set.

There is usually a balance between precision and efficiency in abstract interpretation. This balance can be tuned in two main categories: the widening strategy and the symbolic representation of sets of values. Thanks to the abstract interpretation theory, the symbolic representation can be split into a number of so-called abstract domains, each abstract domain being specialized on certain shapes of sets of values, and all abstract domains communicating to obtain as precise information as possible. Knowing which abstract domains are used in a static analyzer, one can have an idea of its potential precision.

Basic Abstract Domains. The less expensive abstract domain for numerical values is the domain of intervals, as described in [Cousot and Cousot, 1976]. In Astrée, an interval is associated with each variable, with a possibility of distinguishing each cell in arrays, or only the union of all cell values if the array is too big.

Octagon Abstract Domain. This domain, described in [Miné, 2001], will capture relational sets of values. The shape of these relations is of the form $X \pm Y \in \text{interval}$. The complexity of manipulating groups of variables linked through an octagon is cubic in the number of variables. Although it is the relational domain with the best complexity, we cannot afford the complexity of one big octagon relating all pairs of variables in the program. Instead, variables are grouped into small octagons according to user directives or pattern matching of predefined program schemata.

Digital Filters Abstract Domains. Linear filters are widely used in control software. The problem is that although their ideal versions (on real numbers) are well studied, the effect of computing on IEEE floating numbers may change

the stability of the filters. Also, with many filters, it is not possible to bound the output stream in the presence of retroactions by using classical linear abstract domains (even the more powerful polyhedra of [Cousot and Halbwachs, 1978]). [Feret, 2004] developed for Astrée a way of designing very precise and efficient abstract domains to deal with linear filters on floating point numbers.

Decision Trees Abstract Domain. In order to represent precisely the effect of complex control based on boolean variables, Astrée uses decision trees such that the decisions are based on the boolean variables, and the leaves of the trees are numerical abstract domains. This gives very precise informations about booleans, but the complexity is exponential in the number of boolean variables. So we group some boolean variables and some numerical ones in the same way as for octagons: either through user directives or pattern matching.

Unions. Unions of sets of possible values must be performed each time we merge the two branches of an `if` or each time we loop when computing the invariants of `while` loops. In addition to being a costly operation, for all the abstract domains used in Astrée unions imply a loss of precision. In order to keep more precision, at least locally, it is possible to delay the unions. The effect is to partition the traces of executions. Such partitioning, which can be extremely costly if the unions are too much delayed, can be introduced by the user or automatically through pattern matched program schemata.

Choosing Parameters for the Analysis

If Astrée always used all its most precise abstract domains and strategies on all program points and variables, the time and memory consumption of the analysis would be intractable. Luckily, no critical software required that much precision to prove their absence of run-time error so far. Astrée provides a lot of opportunities for the end-user to tune different parameters, so that the analyzer will be precise where it matters. As tuning the analyzer might be difficult for a non-expert, Astrée comes with a number of automatic decision procedures to compute default parameters. Still, it can be useful to know where these parameters can be taken into account.

The different phases of Astrée, after parsing, are:

Preprocessing. Preprocessing is decomposed in three passes. In the first one, the code is simplified, computing constant expressions (in a sound way with respect to floating point computations) and removing unused variables. Then the analyzer uses various pattern matched program schemata to determine where to put partitioning directives, in addition to those specified by the user in the source code. In the third phase, some variables are put together to be later incorporated into octagons or decision trees. The end-user can choose

to put some packs or influence the parameterization, choosing for exemple the maximum number of boolean variables in a decision tree.

Iterator. The actual abstract interpretation of the program starts from a user supplied entry point in the program, such as the `main` function. This interpretation follows the directives (relational packs and partitioning) introduced in the preprocessing phases. For each instruction, the iterator asks the abstract domains to compute a sound approximation of the result of the instruction (the abstract transfer function). The difficult point is then the analysis of the loops, where other parameters must be taken into account. For example, the end-user can choose the number of loop unrollings performed by the iterator, or the stages which will be used in the widening process [Blanchet et al., 2003].

5. Different Uses for Astrée

Although Astrée was designed to answer the specific needs of one end-user, many more end-users might find the analyzer useful.

The primary use of Astrée is the proof of absence of run-time errors. Because Astrée can also use known facts and report on violated assertions inserted in the source code to analyze, it is possible to prove complex user-defined properties. In the near future, we plan to add the possibility of specifying complex temporal properties, such as often required by critical, real-time software specifications.

In addition to reporting potential errors, Astrée can also output the possible sets of values of the variables which were computed to check for those potential errors. On the class of programs for which Astrée was developed, the analysis time is quite low: about one hour per 100 000 lines of program on a 2GHz PC. That makes it possible for using Astrée at the earlier stages of software development. Its high precision makes it likely to discover bugs, and to find their origin by inspecting the sets of possible values leading to that bug. This task will be eased in the future, when Astrée will incorporate some backward analysis which will allow to discover an approximation of the values which led to a failure.

6. Conclusion

Astrée is a static analyzer aiming at proving the absence of run-time errors of synchronous C programs. It is already successful on a class of large embedded programs, where the analysis time is as low as a few hours for hundreds of thousands of lines of code. As Astrée was developed for that class of programs, the Astrée team (B. Blanchet, P. Cousot, R.Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux and X. Rival) expects that some adjustments could be necessary to apply that tool to other families of programs. Our hope is that it

will be possible in a near future to prove the safety of all critical softwares at a reasonable cost.

In addition to analyzing more synchronous programs, we plan the evolution of Astrée in three directions. First, Astrée will propose the automatic proof that the compiled codes of the C programs are also correct by transferring automatically the analysis from source code to compiled code. Second we will add in the analyzer the possibility to perform a backward analysis. This will help to determine if an alarm is due to the imprecision of the analysis or if it is a true bug, and in both cases, it will help finding the source of the imprecision or cause of the bug. Finally, Astrée will be extended to analyze precisely asynchronous programs.

Acknowledgments. Thank you to Radhia Cousot, Antoine Miné and Patrick Cousot who helped me with useful comments and technical support.

References

Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., and Rival, X. (2002). Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In Mogensen, T., Schmidt, D.A., and Sudborough, I.H., editors, *The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones,* LNCS 2566, pages 85–108. Springer.

Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., and Rival, X. (2003). A static analyzer for large safety-critical software. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI'03),* pages 196–207, San Diego, California, USA. ACM Press.

Cousot, P. (1978). *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, France.

Cousot, P. and Cousot, R. (1976). Static determination of dynamic properties of programs. In *Proc. of the Second Int. Symp. on Programming,* pages 106–130. Dunod, Paris, France.

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

Cousot, P. and Cousot, R. (1979). Systematic design of program analysis frameworks. In 6^{th} *ACM POPL*, pages 269–282.

Cousot, P. and Halbwachs, N. (1978). Automatic discovery of linear restraints among variables of a program. In 5^{th} *ACM POPL*, pages 84–97.

Feret, J. (2004). Static analysis of digital filters. In *European Symposium on Programming (ESOP'04),* LNCS 2986. Springer.

IEEE Computer Society (1985). IEEE standard for binary floating-point arithmetic.

JTC 1/SC 22 (1999). Programming languages – C. Technical Report ISO/IEC 9899:1999.

Miné, A. (2001). The octagon abstract domain. In *IEEE AST in WCRE,* pages 310–319.

Miné, A. (2004). Relational abstract domains for the detection of floating-point run-time errors. In *European Symposium on Programming (ESOP'04),* LNCS 2986, pages 3–17. Springer.