Program Analysis for Software Engineering: New Applications, New Requirements, New Tools

Daniel Le Métayer
Inria/Irisa, Campus Universitaire de Beaulieu, Rennes

In order to play a larger role in software engineering tools, static analysis techniques must take into account the specific needs of this application area, in particular in terms of interaction with the user and scalability. This new perspective requires a reexamination of the field of static program analysis both internally and in connection with related areas like theorem proving and debugging.

Despite the very general nature of its principles [2], most of the applications of static analysis so far have been targeted towards optimizing compilers. The efficiency of many implementations of languages on sequential and parallel machines relies in a crucial way on sophisticated static analyzers and there is no reason why this importance should decrease in the future. However, we believe that static analysis should play a much bigger role in another large and very demanding application field, namely software engineering. In this position paper, we argue that:

—A large variety of tasks in the software engineering process could benefit from techniques akin to static program analysis.
—This application field places new demands on static analyzers that should have an impact on their design at both the conceptual and the practical level.
—This new perspective requires reexamination of the field of static program analysis both internally and in connection with related areas that have developed completely independently so far.

1. APPLICATIONS OF STATIC ANALYSIS TO SOFTWARE ENGINEERING

We start with a quick review of applications of static program analysis techniques in the field of software engineering:

—Static debugging [5; 9; 1]: program slicing (for better understanding of the code and the origin of the bugs); automatic proof of properties on definition-use locations of variables, on the range of array indexes, or on pointer dereferences (to isolate potential errors in memory access), detection of dead code.

©ACM, 1996. ACM Computing Surveys Volume 28, Number 4es,
http://www.acm.org/pubs/citations/journals/surveys/1996-28-4es/a167-le_metayer/.

—Testing [3]: static analysis of programs to help construct effective test sets or to optimize non-regression and integration tests.
—Proof of simple correctness properties [6; 4; 8]: functional (such as dependencies between *aspects* of variables or invariants on the shape of data structures) or non-functional (such as confidentiality or integrity for security-critical applications).
—Understanding and documenting programs, isolating differences between two versions of a program and dependencies within applications to ease maintenance, software management, and help reverse engineering [5].

These problems are obviously interrelated and the solutions rely on a bulk of common underlying techniques. Most of these applications have already been studied in the literature, but they have received much less attention than the traditional compiler technology area and have not led to widely used tools. There are at least two significant reasons why this situation should change in the near future:

—The proliferation of software, in particular in embedded and safety-critical systems, imposes new demans for formal verification of code.
—The development of very large and complex applications over a long period of time requires more sophisticated tools for design, testing, understanding and maintenance of software.

Because of its firm theoretical foundations and its mechanical nature, static program analysis is in a good position to help tackle these problems. But it should be recognised that software engineering imposes new demands on analyzers that must be taken seriously in future research in this area.

2. NEW REQUIREMENTS AND NEW TOOLS

The most important requirement imposed by applications in software engineering results from the fact that programmers are faced with the result of the analysis. They should be able to understand it and to relate it to the source program (to detect the origin of potential errors, to modify the program to improve the result of the analysis, ...). More generally, the user should be able to interact with the analyzer, a demand that creates quite a different situation from the traditional use of program analyzers as hidden components of automatic compilers. In terms of models, it is crucial that the specification of the analysis (the property verified by the analyzer) can be described independently from the algorithms used to implement the analyzer (optimized fixed-point computations, symbolic or elimination methods, graph-based techniques, ...). Another important requirement is that some degree of parametricity be offered in order to let the user customize the tool to his own needs. In other words, we would like generic analyzers allowing the user to specify the properties of interest in a restricted logic language. This is to be compared with the use of temporal logics in model checking [10].

Let us stress that even the problem of producing analysis results that are easy to understand is far from trivial. Presentations of analyzers in terms of type inference (with an algorithm that is supposed to be correct and complete with respect to the inference system) may be a step in the right direction, but it is not *per se* the ultimate solution. It is well-known that the typing errors produced by ML polymorphic type checkers can be quite difficult to understand. This situation is not an exception, but a consequence of the fact that the interaction with the user is usually an afterthought rather than a primary design criterion. This factor should be given a top priority in the context of software engineering and should have an impact on the design of program analyzers.

Among the other challenges that must be taken up by the static analysis community, let us mention the issues of scalability and the treatment of industrial programming languages. Analysers should be designed with modularity as a major requirement in order to be of some use in a real context, and they must apply to the languages that are widely used today (or are expected to be in the future). It seems that these languages are mostly object-oriented and distributed. It is quite unfortunate that very few contributions in the field of static program analysis have tackled the specific features of these classes of languages.

3. REEXAMINATION

The requirements expressed in the previous section clearly show that more effort should be put on the study of the relationships between various approaches, both within the static program analysis community and in connection with other areas:

—Various analysis methods have been proposed for different languages, semantics, properties and with different presentations and optimization techniques. It is necessary at this stage to understand the relationship between all these approaches, their similarities, limitations, and complementarity. This study is a prerequisite for the design of generic tools, which would be more versatile and should include as much as possible the sophisticated techniques developed in specific contexts.

—The requirements imposed by the applications in the field of software engineering clearly show that static program analysis in this context would greatly benefit from cross-fertilization with other research areas, such as: interactive theorem proving, proof explanation, model checking, dynamic analysis, debugging, and programming-language design. Just to take two striking examples:
  (1) The borderline between sophisticated generic program analyzers and theorem provers is likely to get fuzzier and a better understanding of their integration could be mutually beneficial.
  (2) The advent of languages better suited to the description of large applications like coordination languages or software architecture languages [7] should help the design of modular and scalable analysis techniques.

REFERENCES

[1] F. Bourdoncle, *Abstract debugging of higher-order imperative languages*, in Proceeding of the ACM SIGPLAN'93 Conf. on Programming Language Design and Implementation, 1993.

[2] P. Cousot and R. Cousot, *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints*, in Proceedings 4^{th} ACM POPL, 1977, pp. 238-252.

[3] E. Duesterwald, R. Gupta and M.-L. Soffa, *A demand-driven analyser for data flow testing at the integration level*, in Proceedings 18^{th} Int. Conf. on Software Engineering, IEEE, 1996, pp. 575-584.

[4] P. Fradet and D. Le Métayer, *Shape types*, in Proceedings 24^{th} ACM POPL, 1997, to appear.

[5] S. Horwitz and T. Reps, *The use of program dependence graphs in software engineering*, Proc. int. Conference on Software Engineering, ACM, pp. 392-411, 1992.

[6] D. Jackson, *Aspect: an economical bug-detector*, in Proceedings of 13^{th} International Conference on Software Engineering, May 1994, pp. 13-22.

[7] D. Le Métayer, *Software architecture styles as graph grammars*, proc. ACM SIGSOFT'96 Symposium on the Foundations of Software Engineering, 1996, pp. 15-23.

[8] M. Mizuno and D. Schmidt, *A security flow control algorithm and its denotational semantics correctness proof*, Formal Aspects of Computing, Vol. 5-3, 1992.

[9] L. Osterweil, *Using data-flow tools in software engineering*, in *Program flow analysis: Theory and applications*,S. Muchnick and N. Jones (Eds), Prentice-Hall software series, 1981, pp. 237-263.

[10] B. Steffen, *Generating data flow analysis algorithms from modal specifications*, Science of Computer Programming, Vol. 21, 1991, pp. 115-139.