Usable Auto-Active Verification

K. Rustan M. Leino and Michał Moskal

Research in Software Engineering
Microsoft Research, Redmond, WA, USA
{leino,micmo}@microsoft.com

Manuscript KRML 212, 2 November 2010.

Abstract. Program verification is unusable. But perhaps not useless. The quality of feedback produced by most verification tools makes it virtually impossible to use them without an extensive background in formal methods. This position paper proposes a research agenda that seeks to overcome these difficulties, aiming to enable well motivated users to take advantage of verification tools.

0 Background

The prospect of formally verifying a piece of software as it is being built, or after it has been built, has strong appeal. Applying program verification in this way may eliminate a number of otherwise latent defects in the final software product, because the verifier considers all possible inputs, all possible control paths, and all possible interactions with the environment. If the program verifier performs modular verification, one can verify a software module early in the development cycle, before the completion of other modules on which it will eventually rely. As is well known, defects are cheaper to correct earlier in the development cycle.

Everything comes at a price. To be useful, the application of a program verifier must stand out as a cost-effective alternative to other means of ensuring program reliability. In other words, the additional reliability that one hopes to gain from the verifier must outweigh the additional cost of using the tool. The dominant alternative today is various forms of testing.

Before considering the application of program verifiers further, let us settle the mere realizability of program verifiers. Here, we have good news. Mechanical tools for program verification have been built since the 1970s [0,9]. Because human users interact with the verifier, theoretical issues like undecidability are not show-stoppers in practice. In the last several decades, extensive theory and tools for various logics and decision procedures have been developed. Although research is still going on in these areas, the field has become mature enough that some recent prize-winning papers have primarily concerned the *application* of program verification [7, 11], rather than just technology building blocks used to construct program verifiers.

For a program verifier to work, one also needs to understand the semantics of the programming language and to provide a specification methodology that stands up to good modularity and information-hiding principles. The language semantics developed decades ago still provides a good foundation today [6, 10, 4]. In the last decade, several

specification methodologies have been developed to address the specification problem associated with pointers and modularity [5].

So, program verifiers can be built. How are they used? The verifier needs the program to be verified and it needs a specification of what it means for the program to be correct. If the specification is fixed, one can hope for a fully *automatic* verifier, for example one built using techniques of abstract interpretation or model checking.

More generally, the specification comes from a combination of usage rules imposed by the language (such as rules about accessing arrays within their bounds), additional usage rules imposed by the verifier (such as a discipline to avoid deadlocks), and user-supplied specifications (such as procedure postconditions). In addition, the verifier generally needs help in completing proofs, and this help may come, for example, in the form of manually supplied loop invariants and commands that invoke proof tactics.

A common and general pattern for architecting a program verifier is to first generate a set of logical *verification conditions (VCs)* from the given program and then process these by some kind of reasoning engine. This has the advantage of separating intricacies of the programming language and the modeling thereof from the underlying logic and its proof procedures. Given this architecture, it is meaningful to categorize program verifiers according to where user input is supplied. In particular, we may consider if user input is supplied before or after VC generation. If the user input is supplied after VC generation, which is the typical case when the reasoning engine is an interactive proof assistant, then the tool offers what is known as *interactive* verification. Tools where user input is supplied before VC generation therefore lie between automatic and interactive verification, which we will give the name *auto-active* verification.

In the 1990s, there was a shift toward auto-active verification, where all interaction with the program verifier was done with annotations supplied in the program text [3]. This was made possible by powerful automatic satisfiability-modulo-theories (SMT) solvers [2]. Another enabler is the fact that proofs often need many of the same cases that programs consider; for example, an if statement in a program usually goes hand-in-hand with a case split in the proof. This gives further credibility to the approach of supplying all human input before VC generation.

1 Bridging the Gap Between User and Tool

Despite these advances, we argue, based on our own experience in building and using auto-active program verification tools (*e.g.*, [1, 8]), that program verification is currently unusable, because current tools require too much expertise from the user. Stated differently, tools can understand our programs, but we cannot understand our tools. Input languages are too hard for the uninitiated to understand. It is too difficult to coax the reasoning engine into completing a proof. Error messages are too cryptic to decipher. And the time the user has to wait for feedback is too long. The problems we have mentioned are noticeable even for expert users, though experts may be able to cope with them. We are upping the ante to cover more users than just experts: we argue that before program verification can be called usable, it must provide a cost-effective solution to *serious non-expert users*.

To address these issues, we propose research along the following lines.

First, we argue that auto-active verification tools are the only ones that can stand a chance of ever being usable by non-experts. It is hard enough to learn the concepts of program verification and how they apply to programs written in a particular language. To also have to look at a verifier's logical encoding of the program semantics and learn effective commands that guide an interactive proof assistant to a proof is asking too much.

Second, any program verification system is likely to have hick-ups and non-obvious moments. Therefore, one should anticipate that users will apply a fair amount of trial and error, if for no other reason than that they did not read the manual carefully enough. To that end, it is essential that the verification time be as short as possible. Notice that the time for *failed* verifications is far more important to optimize than the time to construct or replay succeeding proofs, because the failed verifications are all done on the user's time.

To optimize these times, it helps to have a fast reasoning engine. Beyond that, it seems that one can obtain practically important speed-ups by keeping track of what needs to be re-proved after a change in a program. Such tracking may be done at the level of control paths in a procedure and may use previously completed proofs to determine dependencies.

It may also be possible to unleash the power of a prover in stages. For example, start by attempting the proof using a simple encoding or axiomatization. If the proof fails, report a tentative error to the user. Meanwhile, continue the proof search using more expensive but more powerful strategies.

Third, the process of deciphering error messages must be simplified. Beyond getting an indication of which proof obligation is failing, it is important to receive information about how the failure may occur, maybe even hints as to what the user can do to prevent the failure. We envision a debugging interface akin to dynamic debuggers, where one can step through program states (forwards and backwards), inspect values of program variables (including values obtained by dereferencing pointers in the program's memory), find out which program-declared properties have been applied in a proof attempt, and discover how supplied invariants may fail to be inductive and how other specifications may be too weak.

2 Conclusion

In summary, program verification technology has come a long way. We have a good idea of how to use programming-language semantics, specifications, VC generation, and automatic decision procedures to verify programs. However, to make program verification usable, it must reach a level where it can be comfortably be applied by motivated non-experts. We have proposed a research agenda for how to continue to make progress on verification tools in software engineering. The agenda reduces the emphasis on new advances in the core technologies of provers and semantics, areas where significant progress has already been made. The agenda instead places strong emphasis on user-interface considerations that let the user's attention stay within the context of the program being verified, prioritize speedy delivery of error messages relevant to the user's current focus, and give better explanations of the errors reports.

References

0. Robert S. Boyer and J Strother Moore. *A Computational Logic*. ACM Monograph Series. Academic Press, 1979.
1. Ernie Cohen, Markus Dahlweid, Mark A. Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A practical system for verifying concurrent C. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009*, volume 5674 of *Lecture Notes in Computer Science*, pages 23–42. Springer, August 2009.
2. David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *Journal of the ACM*, 52(3):365–473, May 2005.
3. David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Research Report 159, Compaq Systems Research Center, December 1998.
4. Edsger W. Dijkstra. *A Discipline of Programming*. Prentice Hall, Englewood Cliffs, NJ, 1976.
5. John Hatcliff, Gary T. Leavens, K. Rustan M. Leino, Peter Müller, and Matthew Parkinson. Behavioral interface specification languages. Technical Report CS-TR-09-01a, University of Central Florida, School of EECS, October 2010.
6. C. A. R. Hoare and N. Wirth. An axiomatic definition of the programming language PASCAL. *Acta Informatica*, 2(4):335–355, 1973.
7. Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: formal verification of an OS kernel. In Jeanna Neefe Matthews and Thomas E. Anderson, editors, *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009*, pages 207–220. ACM, October 2009.
8. K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *LPAR-16*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, April 2010.
9. D. C. Luckham, S. M. German, F. W. von Henke, R. A. Karp, P. W. Milne, D. C. Oppen, W. Polak, and W. L. Scherlis. Stanford Pascal Verifier user manual. Technical Report STAN-CS-79-731, Stanford University, 1979.
10. David C. Luckham and Norihisa Suzuki. Verification of array, record, and pointer operations in Pascal. *ACM Transactions on Programming Languages and Systems*, 1(2):226–244, October 1979.
11. Jean Yang and Chris Hawblitzel. Safe to the last instruction: automated verification of a type-safe operating system. In Benjamin G. Zorn and Alexander Aiken, editors, *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010*, pages 99–110. ACM, June 2010.