# Verified Squared: Does Critical Software Deserve Verified Tools?

Xavier Leroy *
INRIA Paris-Rocquencourt
Xavier.Leroy@inria.fr

## Abstract

The formal verification of programs have progressed tremendously in the last decade. Principled but once academic approaches such as Hoare logic and abstract interpretation finally gave birth to quality verification tools, operating over source code (and not just idealized models thereof) and able to verify complex real-world applications [6, 8, 15, 18]. In this talk, I review some of the obstacles that remain to be lifted before source-level verification tools can be taken really seriously in the critical software industry: not just as sophisticated bug-finders, but as elements of absolute confidence in the correctness of a critical application.

***Categories and Subject Descriptors*** D.2.4 [*Software Engineering*]: Software/Program Verification; D.3.1 [*Programming Languages*]: Formal Definition and Theory; D.3.4 [*Programming Languages*]: Processors; F.3.1 [*Logics and meanings of programs*]: Specifying and Verifying and Reasoning about Programs; F.3.2 [*Logics and meanings of programs*]: Semantics of Programming Languages

***General Terms*** Languages, Verification

## Extended abstract of invited talk

***Critical software*** Software is critical when human lives are at stake and there is no simple, safe failure mode. I take the paradigmatic example of electronic flight control systems ("fly-by-wire") in aircraft, which provides a fascinating glimpse of what software perfection may look like. Impressive reliability has been achieved so far—but at tremendous costs—by following meticulous development and certification processes, as codified in DO-178 regulations for instance. These processes rely on (qualitative) review, (quantitative) analysis, and (black-box) testing at multiple levels of the design, complemented with strong traceability between high-level requirements and actual implementation. In this domain, verification tools have great potential to facilitate and strengthen the results of analysis phases, while at the same time remove the need for some costly tests [15].

***Trust in compilers and code generators*** With a few notable exceptions, most verification tools operate over the source code: generally C code, possibly Scade/Simulink block diagrams. The guarantees provided by these tools are therefore vulnerable to *miscompilation*: compiler bugs that cause wrong executable code to be silently generated from correct sources. DO-178 recognizes this issue and mandates that either the compiler/code generator is qualified to the same level of assurance as the code it compiles, or the generated code is certified as if hand-written [9]. Neither requirement can be fully achieved in the case of a conventional C compiler. Even when possible, e.g. for Scade-to-C code generators, qualification of a compiler generally comes at the cost of inefficiencies in the generated code.

An obvious alternative is compiler verification: apply mechanized program proof to the compiler itself and prove semantic preservation (the generated code behaves as prescribed by the semantics of the source program). For example, CompCert [10, 11] is a lightly-optimizing, multi-pass compiler for a large subset of C that was programmed and proved semantically preserving using Coq. From the results of the CompCert experiment, the formal verification of realistic compilers appears within reach of today's proof technology, and generates (arguably) unprecedented confidence in the compilation process and, indirectly, in source-level verification. At the same time, much remains to be done in this area, such as verifying more of the "first miles" (preprocessing, parsing) and the "last miles" (assembing and linking, or even connections with micro-architecture verification), and proving compilers for source languages significantly different from C.

***Trust in verification tools*** While unsound verification tools still have value as bug-finders, verification-based certification of critical software demands formal evidence that the results of the verification tools are sound with respect to concrete executions. The general principles for stating and proving these soundness results are well known; some approaches, such as classic abstract interpretation, even ensure soundness by construction of the analysis. Mechanization of these principles appear well within reach [2, 5, 17]. Perhaps the biggest challenge to extend these efforts to realistic verification tools is the need to prove some of the fairly complex algorithms involved in the implementation of abstract domains and theorem provers.

***Verified vs. validated*** Not all parts of a compiler or verifier need to be proved: only those parts that affect soundness, but not those part that only affect termination, precision of the analysis, or efficiency of the generated code. Leveraging this effect, complex algorithms can often be decomposed into an untrusted implementation followed by a formally-verified validator that checks the computed results for soundness and fails otherwise. (Failure is not an option in flight, but is an option at compile-time and verification-time.) In static analysis, for example, it is much easier to prove the code that checks that an abstract value $X$ is a post-fixpoint of an operator $F$ than to prove the correctness of the fixpoint iteration

* Partially supported by ANR grant Arpège U3CAT.


*POPL '11* January 26–28, Austin, TX, USA
ACM 978-1-4503-0490-0/11/01.

that computes $X$. Likewise, in a compiler, translation validation of the results of an untrusted compilation pass can provide soundness guarantees as strong as formal verification of this pass, provided the validator is proved sound [12, 13, 16]. While often effective to reduce the overall proof effort, validation a posteriori is not a silver bullet either: many compiler passes are no easier to validate than to prove correct once and for all. Between full compiler verification and full translation validation lies a continuum of combined approaches that remain to be systematically explored.

***Mechanized semantics*** Formal verification of compilers and program verifiers does not eliminate all sources of uncertainty, but reduces the issue of trusting these tools to (primarily) the issue of trusting the formal semantics used for the source and target languages. Extensive manual reviews and testing of these semantics remains a necessity. As a further difficulty, different tasks favor different styles of semantics: big-step semantics or definitional interpreters are perhaps the easiest to review by hand, while verification of type-checkers, abstract interpreters, and compilers favor Wright-Felleisen small-step semantics, collecting semantics, and transition semantics with explicit contexts, respectively. Finally, there is also a strong tension between low-level operational semantics, easy to mechanize but difficult to work with, and higher-level approaches such as type- or step-indexed logical relations that provide much nicer reasoning principles but are expensive to set up [1, 4]. Agreement on one reference semantics for, say, the C language is therefore unlikely to happen. Instead, we should strive for systematic, mechanized proofs of equivalence or implication between these various semantics. Such proofs, as well as tool verification in general, considerably strengthen the confidence we can have in these semantics.

***Multiple languages*** Many applications combine sources written in several languages, such as Scade/Simulink, C, and assembly for control/command systems. Also, implementations of high-level languages combine compiled code with run-time systems written in a lower-level language. Verifying the correctness of the whole system requires a delicate combination of proofs, some coming from the verification of the individual sources, others coming from the verification of the compilers involved. In this situation, whole-program semantic preservation result such as those of CompCert do not directly apply, and finer-grained approaches are needed: possibly proof-preserving compilation [3, 14] combined with separation logics.

***Floating-point woes*** "It makes me nervous to fly an airplane since I know they are designed using floating-point arithmetic." A. Householder, the author of this famous quote, should be even more nervous because modern airplanes *fly* using floating-point arithmetic. Floating-point is, today, the biggest source of infidelities between the semantics used for verification and the code that actually runs, owing to fundamental aspects of floating-point arithmetic (limited precision, limited range) that are further aggravated by sloppy language specifications (C leaves unspecified the precision of intermediate floating-point results) and dubious compiler optimizations (reassociation during vectorization, for instance). Good static analyzers manage to conservatively approximate most of these floating-point artefacts, at significant cost. Program logics and deductive program provers still have some way to go in this direction: just specifying simple numerical routines in a floating-point accurate manner is already a challenge [7].

***Conclusions*** The formal verification of development and verification tools for critical software appears worthwhile: first, to strengthen the confidence we can have in the results of verification tools, and therefore to make a stronger case for their adoption; second, because it raises scientifically-challenging issues, shedding new light on well-researched areas and exposing new problems at their frontiers. The POPL community has, obviously, much to contribute to this endeavor: feel free to join!

## References


[1] A. J. Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In *Programming Languages and Systems, 15th European Symposium on Programming, ESOP 2006*, volume 3924 of *Lecture Notes in Computer Science*, pages 69–83. Springer, 2006.

[2] A. W. Appel and S. Blazy. Separation logic for small-step Cminor. In *Theorem Proving in Higher Order Logics, 20th Int. Conf. TPHOLs 2007*, volume 4732 of *Lecture Notes in Computer Science*, pages 5–21. Springer, 2007.

[3] G. Barthe, B. Grégoire, C. Kunz, and T. Rezk. Certificate translation for optimizing compilers. *ACM Transactions on Programming Languages and Systems*, 31(5), 2009.

[4] N. Benton and C.-K. Hur. Biorthogonality, step-indexing and compiler correctness. In *International Conference on Functional Programming 2009*, pages 97–108. ACM Press, 2009.

[5] F. Besson, D. Cachera, T. P. Jensen, and D. Pichardie. Certified static analysis by abstract interpretation. In *Foundations of Security Analysis and Design*, volume 5705 of *Lecture Notes in Computer Science*, pages 223–257. Springer, 2009.

[6] 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. In *Programming Language Design and Implementation 2003*, pages 196–207. ACM Press, 2003.

[7] S. Boldo, J.-C. Filliâtre, and G. Melquiond. Combining Coq and Gappa for certifying floating-point programs. In *Intelligent Computer Mathematics, Calculemus/MKM 2009*, volume 5625 of *Lecture Notes in Computer Science*, pages 59–74. Springer, 2009.

[8] G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: formal verification of an operating-system kernel. *Communications of the ACM*, 53(6):107–115, 2010.

[9] A. J. Kornecki and J. Zalewski. The qualification of software development tools from the DO-178B certification perspective. *CrossTalk*, Apr. 2006.

[10] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[11] X. Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

[12] G. C. Necula. Translation validation for an optimizing compiler. In *Programming Language Design and Implementation 2000*, pages 83–95. ACM Press, 2000.

[13] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *Tools and Algorithms for Construction and Analysis of Systems, TACAS '98*, volume 1384 of *Lecture Notes in Computer Science*, pages 151–166. Springer, 1998.

[14] Z. Shao, V. Trifonov, B. Saha, and N. Papaspyrou. A type system for certified binaries. *ACM Transactions on Programming Languages and Systems*, 27(1):1–45, 2005.

[15] J. Souyris, V. Wiels, D. Delmas, and H. Delseny. Formal verification of avionics software products. In *FM 2009: Formal Methods*, volume 5850 of *Lecture Notes in Computer Science*, pages 532–546. Springer, 2009.

[16] J.-B. Tristan and X. Leroy. Verified validation of Lazy Code Motion. In *Programming Language Design and Implementation 2009*, pages 316–326. ACM Press, 2009.

[17] H. Tuch, G. Klein, and M. Norrish. Types, bytes, and separation logic. In *34th symposium Principles of Programming Languages*, pages 97–108. ACM Press, 2007.

[18] J. Yang and C. Hawblitzel. Safe to the last instruction: automated verification of a type-safe operating system. In *Programming Language Design and Implementation 2010*, pages 99–110. ACM Press, 2010.