EPiC Series in Computing
Volume 51, 2017, Pages 37–41
ARCADE 2017. 1st International Workshop on Automated Reasoning: Challenges, Applications, Directions, Exemplary Achievements

24 Challenges in Deductive Software Verification*

Reiner Hähnle[1] and Marieke Huisman[2]

[1] Technische Universität Darmstadt, Germany, haehnle@cs.tu-darmstadt.de
[2] University of Twente, Enschede, The Netherlands, M.Huisman@utwente.nl

Abstract

Deductive software verification aims at formally verifying that all possible behaviors of a given program satisfy formally defined, complex properties, where the verification process is based on logical inference. We list the most important challenges for the further development of the field.

1 Introduction

Deductive software verification aims at formally verifying that all possible behaviors of a given program satisfy formally defined, complex properties, where the verification process is based on some form of logical inference, i.e., "deduction". We consider deductive software verification, i.e. automated reasoning about programs, at least in part to be a subfield of automated deduction. This is witnessed by the long tradition of papers on deductive aspects of software verification at CADE and IJCAR. On the other hand, deductive verification is one of the most important consumers of automated deduction technology; it was and is a driver for many core research questions in automated deduction, including theory reasoning, SMT solving, tactical theorem proving, etc. For this reason we believe our discussion to be highly relevant for automated deduction in general.

Since its inception in the late 1960s the field of deductive verification came a long way from pen-and-paper proofs for programs in small, idealized languages to highly automated proofs of complex library or system code written in mainstream programming languages. The field has now reached a stage of maturity that permits to use deductive verification technology in an industrial setting. However, formidable challenges remain and we present an overview of what we consider the most important challenges in the area of deductive software verification.[1] We divide these into technical (Sect. 2) and non-technical (Sect. 3) aspects.

*The present paper is based on a longer article to appear in the LNCS 10000 anniversary volume.

[1]We focus on *contract-based, deductive verification of imperative and OO programs*. Hence, we do not explicitly discuss model checking, SMT solvers, general proof assistants, program synthesis, correctness-by-construction, runtime verification, or abstract interpretation. We also do not cover fully automated verification tools for generic safety properties. However, we expect that most of our conclusions are still valid in those fields.

2 Challenges: Technical

Specification. Deductive verification starts with specifying *what* should be verified. This is where specification languages come into play. In essence, expected program behaviour is described in the form of a method contract: a precondition specifies the assumptions under which a method may be called; a postcondition specifies what is achieved by its implementation: the computed result, or its effect on the global state. For the deductive verification community, the design of JML, the Java Modeling Language, has been a major achievement. JML is a rich specification language; complex specifications can be expressed in it. The design of JML has been influential in the design of other specification languages for deductive verification, such as the ANSI/ISO C Specification Langage (ACSL) and the Spec# specification language for C#.

A central problem of deductive verification is that specifications must spell out all details for the verification proof to succeed. This is problematic for the verification of large code bases and exacerbated by usage of off-the-shelf libraries. To improve this situation, we believe attention should be given to address the following two challenges:

S.1 Provide specifications for widely-used APIs—and verify reference implementations against them. This task is work-intensive and has little scientific reward. It is, therefore, difficult to find funding, see F.1.

S.2 Develop techniques to infer specifications from code in a (semi-)automated manner. Many specification details that have to be spelled out explicitly, actually can be inferred from the code. There is initial work on specification generation, but it is not integrated into deductive verification frameworks (I.9).

Integration. Integration aspects of formal verification appear on three levels. The most elementary is the software engineering aspect of tool integration and reuse. Then there is the aspect of integrating different methods and analyses with their complementary strengths. Finally, there is the challenge to integrate formal verification technology into an existing production environment such that added value is perceived. We discuss each aspect in turn.

Intermediate verification languages are good reuse candidates at the level of tools, because they are small and have a clear semantics. In addition, compilation is a well-understood, mainstream technology with excellent tool support. This makes it relatively easy to implement new frontends. On the other hand, tool reuse at the "user level", for example, for JML/Java or ACSL/C is much harder to achieve and we are not aware of any significant case.

I.1 Equip frontend (JML, Java, ACSL, C, ...) as well as intermediate (Boogie, Silver, Why, LLVM-IR, bytecode, ...) languages with precise, preferably formal semantics. In the case of complex frontend languages this involves identifying a "core" that must then be supported by all tools.

I.2 Equip formal verification tools with a clear, modular structure and offer their functionality in well-documented APIs. This is a work-intensive task with few scientific rewards and, therefore, closely related to Challenge F.1.

I.3 Establish and maintain a tool integration community, to foster work on reuse and increase its appreciation as a valuable contribution.

One of the largest, self-imposed stumbling blocks of formal methods has been the propagation of monolithic approaches. At least in deductive verification, it became very clear within

the last decade that software development, formal specification, formal verification, runtime verification, test case generation, and debugging are not separate activities, but they have to be done in concertation. At the same time, formal specifications have to be incrementally developed and debugged just as the pieces of code whose behavior they describe. This is now commonly accepted in the community, even if the infrastructure is not there yet.

I.4 Calls to auxiliary tools must return certificates, which must be re-interpreted in the caller's correctness framework. This is necessary to ensure correctness arguments without gaps.

I.5 The semantic assumptions on which different analysis methods are based must be spelled out, so that it is possible to combine different approaches in a sound manner.

I.6 A systematic classification and evaluation of possible methodological combinations should be developed..

I.7 Establish a research community working on method integration.

It is very difficult to integrate software verification technology into a production environment. Some reasons have a non-technical nature and are discussed in Sect. 3. Another issue might be the lack of coverage, see below. First we concentrate on processes and work flows. How can formal software verification be usefully integrated into a software development process? The emerging integration of verification, test generation, and debugging aspects into single tool chains, as described above, is an encouraging development. We begin to see deductive verification tools that are intentionally presented as enhanced software development environments. Some verification tools support users in keeping track of open proof obligations after changes to the code or specification. However, to realize versioning and team-based development of verified software, it is necessary to generalize code repositories into proof repositories.

Most verification attempts fail at first. It requires often many tries to render a complex verification target provable. The nature of software development is incremental and evolutionary. Without contracts, in particular for library methods, deductive verification does not scale.

I.8 Implement proof repositories that support incremental and evolutionary verification and integrate them with verification tools.

I.9 Integrate automated specification generation (S.2) and bug finding techniques into the verification process.

Coverage Tool support for a realistic programming language entails verification techniques such as reasoning about integer types (including overflow), reference types, and exceptions. Properties that are to be verified can be functional as well as non-functional (complexity, information flow, ...). There is active research to investigate how to extend support for deductive verification to concurrent software. This opens up a whole new range of problems, because one has to consider all possible interleavings of the different program threads.

C.1 Precise verification of floating point numbers is essential for many algorithms, in particular in domains such as avionics.

C.2 Tool support for verification of concurrent software is still in its infancy. We need further developments in two directions: (1) automated support of functional properties of fine-grained concurrency, which does not require an overload of annotations, and can be used by non-experts, and (2) verification techniques for relaxed memory models that resemble realistic hardware-supported concurrent execution models.

C.3 Reasoning techniques for programs that use reflection are necessary for application scenarios such as the analysis of obfuscated malware, or of dynamic software updates.

C.4 The rapid evolution of industrial programming languages (e.g., substantial new features are added to Java every 2–3 years) is a challenge for tools that are maintained with the limited manpower of academic research groups. Translation to intermediate languages is one way out, but makes it harder to provide feedback at the source level.

C.5 Deductive verification technology is not merely applicable to software, but also to cyber-physical systems. There are some tools for deductive verification of hybrid systems, but its application to general computational models is an open challenge.

3 Challenges: Non-Technical

Usability Researchers working in formal verification are method- and tool-driven. The effectiveness of a novel method or new tool is usually simply claimed, at best, it is underpinned by citing execution statistics. The latter are often micro benchmarks carried out on small language fragments. The best case are industrial case studies which may or may not be representative and in nearly all publications these are performed by the researchers and tool builders themselves, not by the intended users. To convince industrial stakeholders of the usefulness of a formal verification approach, it is not only necessary to demonstrate that it can fit into the existing development environment, but also to argue that one can solve tasks more effectively or faster than with a conventional solution. This is only possible with the help of experimental user and usability studies. There are very few usability studies around formal verification tools. We know of *only one* experimental user study that investigated the impact of design decisions taken in a verification system on user performance.

U.1 Claims about increased effectiveness or productivity attributed to new methods or tools should be backed up by experimental user studies. This should answer such questions as in which manner interactive verification needs to be more automated.

U.2 Establish the paper category *Experimental User Study* as an acceptable kind of submission in formal verification conferences and journals.

Funding To support formal verification of industrial languages in real applications requires a sustained effort over many years. To formulate appropriate theoretical and methodological underpinnings took decades and the process is still not complete for complex aspects such as floating point types and weak memory models. It takes a long view, patience, and careful documentation to avoid repeated "re-invention of the wheel" or even regression. Tool building is particularly expensive and can take decades. To protect investments and to ensure measurable progress, long-term projects turned out to be very suitable. There are several long-term projects in deductive software verification that have sufficiently matured to enable industrial applicability. Unfortunately, neither the trend to embedded industrial research nor the current climate of academic funding are very well suited for this kind of enterprise. Further detrimental factors to long-time engineering-heavy projects are the publication requirements for tenured positions in Computer Science as well as the unrealistic expectations on short-term impact demanded from many funding agencies. Successful long-term research is not "disruptive" in its nature, but slowly and systematically builds on previous results.

F.1 The academic reward system should give incentives for practical achievements and for long-term success.

F.2 Large parts of Computer Science should be classified and treated as an Engineering or Experimental Science with an according funding model. Complex software systems should be viewed like expensive equipment that needs trained personnel to build and maintain it.

F.3 Grant proposals should foresee and include funding to carry out systematic experimental studies, also involving users.

Industrial and Societal Context The best prospects for industrial take-up of deductive verification technology is in areas characterized by high demands on software quality. This is clearly the case for safety- and security-critical domains that are regulated by formal standards overseen by certification authorities. With the ongoing global trend in digitalization, however, we might experience a surge in software that can be deemed as safety- or security-critical, in particular, in the embedded market (e.g., self-driving cars, IoT). An interesting side effect of digitalization is the arrival of companies on the software market that so far had no major stake in software. Here is an opportunity for formal methods and formal verification, in particular, since software verification tools are also applicable to cyber-physical systems (C.5).

New features of languages such as C/C++ or Java are not introduced with an eye on verifiability, making formal verification and coverage unnecessarily difficult.

CT.1 Researchers from formal verification should become involved in language standardization. Research in programming languages and formal methods must be better coordinated.

CT.2 Researchers from the formal verification area should become actively involved in the standardization efforts of certification authorities.

CT.3 Develop and apply specific quality assurance measures for verification tools: test coverage, incremental testing, external validation, etc.