– Discussion Paper –

Must Program Verification Systems and Calculi Be Verified?

Bernhard Beckert and Vladimir Klebanov

University of Koblenz-Landau
Institute for Computer Science
beckert@uni-koblenz.de, vladimir@uni-koblenz.de

Abstract. With this paper, we want to provoke discussion on whether verification tools and calculi themselves need to be formally verified. We argue that though verifying the verifier is useful, it is not an absolute necessity. The (limited) resources in academia may be better spent on improving verification systems in other ways.

1 The Question

Do program verification systems and calculi need to be verified? Many people, both within and outside of the formal methods research community, seem to think so.[1] Is a verification calculus without a formal soundness proof useful? And if we do not use our own product, why should our customers? So far, academia has put a lot of effort into soundness proofs of verification calculi.[2]

In this paper we argue that though it is useful to formally verify a verification system, it is not a necessity. The (limited) resources in academia may be better spent on improving verification systems in other ways.

2 Why Verifying the Verifier is Not a Necessity

2.1 The Different Artefacts

Before we go on, we clarify which different artefacts we have to distinguish, using deductive program verification as an example:

[1] "A Hoare logic that is unsound would be useless since its very purpose is to verify correctness of programs. Thus after giving a Hoare logic the proof of its soundness is obligatory, in particular when—like in our case—the rules are rather involved and thus their correctness is by far not obvious." [16]

[2] "The proof rules are specified in KIV and their correctness with respect to the [own] semantics has been proved. [...] All 57 rules have been proved correct. The specification and verification effort required several months of work. [...] As can be imagined several errors were found during verification. Most of them are errors only for type incorrect programs." [13]

– The program language specification. Note that, in the case of Java, this is written in English. It is precise but not formal. There is no official *formal* Java semantics. This is the only informal artefact that we have to consider.
– A formal specification of the program language semantics.
– The verification calculus (e.g., a Hoare logic calculus).
– The system that executes the calculus (e.g., a tactical theorem prover).
– The program that is to be verified.

2.2 Correctness is Not the Only Criterion—Verification as an Engineering Tool

If we want our program verification systems to be used in practice, we have to consider the practitioners' viewpoint:

– Practitioners know that verified software may still fail.[3] This can happen because the software is part of a larger system that fails (compiler, operating system, hardware); or because the specification has an error or does not cover an important aspect, such as security, quality of service, or, in particular, fault tolerance of the whole system. Thus, correctness is never an absolute. Nothing is 100% correct in the sense that it does never fail.
– Practitioners know that formal methods are not the only way to reach a high level of dependability. High dependability of a verification tool used in practice can be achieved with testing and experience from long-term use as well. For example, people do believe in the correctness of Isabelle even though the implementation is not verified (neither is the implementation of ML verified, etc.). That shows that at some point, even formal methods people stop verifying things that are well tested.
– Practitioners not only want dependable software but software that they can *trust* to be dependable. Trust is a social process. Thus, introducing a technology (such as a verification calculus or system) cannot be done abruptly but requires a step-wise process. The new technology has to be evaluated in practice, even if it has been formally proven correct. When it is introduced, it has to be compared to and supported by well-known and trusted techniques (such as testing).
– Practitioners are interested in other things besides correctness: How efficient the verification mechanisms and tools are; how well they are integrated into their software development tools and processes; the tools' coverage of the programming language and its interesting features; and how well the mechanism is suited for uncovering errors apart from proving their absence.

As a consequence of these considerations, correctness of the verification mechanism is not an absolute but there is a trade-off between degree/probability of correctness and other properties of the verification system. For this reason,

[3] For example, the current airborne software regulations DO-178B [11] (authoritative for the whole aviation industry) state that "formal methods are complementary to testing".

developers of verification tools should think hard, whether verifying the verification tool and calculus is worth the effort or whether that effort is better spent improving the calculus and tools in other respects such as usability, coverage, and efficiency. To make our point, we compare this scenario:

> *Researcher:* I have a verification calculus covering 75% of all features of your programming language. It is 100% correct.
> *Practitioner:* So what?! I can't use it in practice.

To this one:

> *Researcher:* I have a verification calculus covering 100% of all features of your programming language. It is 99.9% correct.
> *Practitioner:* Interesting. Show me!

This point resounds in [5]. Even though the author has verified the rules of her Hoare Logic w.r.t. her own semantics in both PVS and Isabelle/HOL, she ponders extensively and takes a contrary stand on a related issue—the correctness of the underlying theorem prover.[4]

Also, in model checking and SAT-based approaches, most of the systems are not verified.[5] Nonetheless these formal methods are among the most successful in practice.

2.3 The Sorry Lack of an Official Formal Language Semantics

There is no official formal semantics of the Java programming language. Sun Microsystems, the holder of the Java trademark, decides what constitutes a valid Java implementation within the framework of the Java Community Process. It is assumed that every such implementation adheres to the Java Language Specification, which is a precise but informal document. A similar statement holds for most programming languages used in practice today.

Lately, several research groups have come up with a number of formal semantics for (fragments of) the Java language. Ultimately, there is no way to judge whether any of these semantics is adequate, i.e., reflects the official informal specification correctly. Verifying a calculus against these formal semantics is helpful, but some doubt will always remain about whether the calculus is correct w.r.t. the official language specification and its implementations (compilers, virtual machines), which is what counts in practice. Consequently, other methods such as testing the calculus using a large number of programs (e.g., a compiler test suite like [6]) can lead to the same—or even a higher—degree of correctness w.r.t. the informal language specification as a formal proof.

[4] "Of course, a theorem prover should be sound. [...] However, also efficiency is an important consideration in the design. If a tool is sound, but too slow, it is not useful for verifications of larger systems. Also, as explained above, even though PVS contains soundness bugs, it is still a great help in specification and verification, since most of the time it works 'correctly'." [5]

[5] As far as we know, there are currently only two verified model checker implementations available [10, 12], none of which is in wide practical use.

2.4 Localized Effect of Errors

In verification calculi for programming languages like Java, most of the (many) rules correspond to particular features of the language. Therefore, the effects of an error in most cases remain localized and do not lead to catastrophe. That is, verification proofs for (parts of) programs not containing the particular feature are not affected.

However, this argument does not hold for the generic parts of the calculus that do not correspond to a particular program language feature (for example, an error in the induction rule). Thus, correctness proofs for the core calculus are a different story. On the other hand, the core usually has much fewer rules, so that its correctness proof is usually less work and may even be done with paper and pencil (e.g. [3]).

2.5 Different Kinds of Errors Require Different Validation Methods

Assume that a calculus has an unsound rule, where that rule reflects the semantics of a particular feature of the programming language.

Such an error can have different reasons. It may result from a simple oversight, or the designer of the calculus may have misunderstood how the program language feature actually works. Also, one has to distinguish rules for programming language features that occur frequently from those for obscure features rarely used in practice. And, finally, there are errors that lead to an unsound proof only if the bug in the program is "compatible" to the bug in the rule, as opposed to errors that can lead to an unsound proof whenever the corresponding program language feature occurs.

Verifying the calculus is well-suited for finding most kinds of errors but is prone to fail if an unsoundness results from a misunderstanding of how the language features works, as then there is a high probability that the same problem occurs in the formal language semantics as well (in particular, if both the formal semantics and the calculus are constructed by the same person).

Testing the calculus by applying it to many example programs may fail to find errors in rules for obscure features rarely occurring in the examples. On the other hand, testing is good for uncovering misunderstandings. It is much easier to do a cross-validation with test programs written by different people than to cross-verify a calculus w.r.t. different formal semantics (see Section 3). Note, however, that it is important to use both correct and incorrect programs for testing a calculus.

Also, it is easy to automatically redo tests when the system or calculus is modified, while reusing a verification proof may be difficult and require interactions and/or a proof-reuse mechanism.

Thus, both approaches to validating a calculus (formal correctness proofs and testing) have their strengths and weaknesses; neither one is inherently superior.

2.6 Scalability of Validation Approaches

When validating a piece of software (by testing or verification), there are two things that can make it hard to find a particular error: the size of the program and the "complexity" of the error, i.e., how complex a test case has to be to uncover the error and how probable it is that a particular test case will do so. Both dimensions (size of program, complexity of error) have to be considered when we talk about scalability of a validation method.

When validating a verification system or calculus, the situation is different in that the size of the calculus and the system is (more or less) fixed. The method has to be able to handle a calculus or system of that particular size; no scaling up is needed. On the other hand, scaling along the other dimension (complexity of the error) is still of great importance.

A measure for the complexity of errors in a verification calculus is the number of program language features that are involved. For example, the rule for handling while-loops in Java has to consider the possibility that the loop body throws an exception and, thus, terminates abruptly. If the loop rule is erroneous and does not cover the case of abrupt termination, then that problem can only be found with a test case involving both features: while-loops and exception throwing.

Finding very complex errors is difficult to do by testing. This problem is limited, however, by the fact that programming languages are designed in such a way that their features are as independent as possible (since otherwise they are hard to understand and use for programmers). Altogether, the question of scaling along the complexity dimension is an argument in favor of verifying the verification system—but not a very strong one.

3 Cross-Verification: A Better Way

In most cases when verification calculi are verified, the formal language specification, the calculus, and the correctness proof are all done by the same people (or group of people). That increases the probability that the formal language specification is inadequate and the discrepancies to the informal specification remain undetected. That is, if you have not understood the informal specification correctly and, thus, the formal semantics is not adequate, a formal correctness proof does not help.[6]

Thus, we argue, that if verification calculi are verified, they should be cross-verified against other people's specifications of the programming language semantics. With cross-verification, the probability of uncovering errors is much higher.

[6] The UK Defence Standard 00-55 "Requirements for Safety Related Software in Defence Equipment" [9] demands that "[...] there should be at least a peer review of the proof obligations and formal arguments [by a member of the team] other than the author [...]".

Another interesting approach to increase system reliability that is based on cross-validation is described in [14]. To check the correctness of the popular model checker SPIN, random test-based cross-checking was performed between different implementations of the LTL formula translation step. It helped uncover several bugs in SPIN.

4 The Situation in the KeY Project

The authors of this paper have been working on a Java verification system within the KeY project. The KeY tool [1, 7] is a mature and established verification system for Java. Its Dynamic Logic calculus covers 100% of Java Card, including—among other things—full support for dynamic object creation (with static initialization), efficient aliasing treatment, full handling of exceptions and method calls, Java-faithful arithmetics, Java Card transactions, etc.

The KeY team has refrained from stating its own formal semantics of the Java language (besides the rules of the calculus). Thus, the KeY semantics of Java is given by the calculus itself. Resources saved on this were instead spent on further improvement of the system. At the same time, the KeY project performs ongoing cross-verification against other Java formalizations to ensure the faithfulness of the calculus.

One such effort compares the KeY calculus with the Bali semantics [16], which is a Java Hoare Logic formalized in Isabelle/HOL. KeY rules are translated manually into Bali rules. These are then shown sound with respect to the rules of the standard Bali calculus. The published result [15] describes in detail the examination of the rules for local variable assignment, field assignment and array assignments. These rules are of particular importance to every Java calculus.

A different approach is described in [2]. It takes as a reference another Java semantics [4], which is formalized in Rewriting Logic [8] and mechanized in the input language of the Maude system. This semantics is an executable specification, which together with Maude provides a Java interpreter for free. Considering the nature of this semantics, we concentrated on using it to verify our program transformation rules. These are rules that decompose complex expressions, take care of the evaluation order, etc. (about 45% of the KeY calculus). For the cross-verification, the Maude semantics was "lifted" in order to cope with schematic programs like the ones appearing in calculus rules. The rewriting theory was further extended with means to generate valid initial states for the involved program fragments, and to check the final states for equivalence. The result is used in frequent completely automated validation runs, which is beneficial, since the calculus is constantly extended with new features.

Furthermore, the KeY calculus is regularly tested against the compiler test suite Jacks [6]. The suite is a collection of intricate programs covering many difficult features of the Java language. These programs are symbolically executed

with the KeY calculus and the output is compared to the reference provided by the suite. This approach has also been taken by others.[7]

5 Conclusion

Industry people should employ more verification, while verification researchers should consider more testing (or other empirical methods).

References

1. Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H. Schmitt. The KeY tool. *Software and System Modeling*, 4:32–54, 2005.
2. Wolfgang Ahrendt, Andreas Roth, and Ralf Sasse. Automatic validation of transformation rules for Java verification against a rewriting semantics. In Geoff Sutcliffe and Andrei Voronkov, editors, *Proceedings, 12th International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Montego Bay, Jamaica*, volume 3835 of *LNCS*, pages 412–426. Springer, Dec 2005.
3. Bernhard Beckert and André Platzer. Dynamic logic with non-rigid functions: A basis for object-oriented program verification. In U. Furbach and N. Shankar, editors, *Proceedings, International Joint Conference on Automated Reasoning (IJCAR), Seattle, USA*, LNCS. Springer, 2006. To appear.
4. Azadeh Farzan, Feng Chen, José Meseguer, and Grigore Roşu. Formal analysis of Java programs in JavaFAN. In Rajeev Alur and Doron Peled, editors, *CAV*, volume 3114 of *Lecture Notes in Computer Science*, pages 501–505. Springer, 2004.
5. Marieke Huisman. *Reasoning about Java Programs in Higher Order Logic with PVS and Isabelle*. PhD thesis, University of Nijmegen, 2001.
6. Jacks (Jacks is an automated compiler killing suite). At `http://www.sourceware.org/mauve/jacks.html`.
7. The KeY project homepage. At `http://www.key-project.org/`.
8. José Meseguer and Grigore Rosu. Rewriting logic semantics: From language specifications to formal analysis tools. In D. Basin and M. Rusinowitch, editors, *Automated Reasoning, Second International Joint Conference, IJCAR 2004, Cork, Ireland, Proceedings*, LNCS 3097, pages 1–44. Springer, 2004.
9. UK Ministry of Defence. Requirements for safety critical software in defence equipment (part 1: Requirements, part 2: Guidance). Defence Standard 00-55, Issue 1, Directorate of Standardisation, MoD, 1997.
10. Wolfgang Reif, Jürgen Ruf, Gerhard Schellhorn, and Tobias Vollmer. Do you trust your model checker? In *FMCAD '00: Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design*, pages 179–196, London, UK, 2000. Springer-Verlag.

[7] "However, both semantics and calculus could be wrong. It is possible to validate the semantics by 'running' test programs in KIV (automatically applying the proof rules) and comparing the output with a run of a Java compiler and JVM (currently 150 examples), and this certainly increases confidence in the semantics [...]" [13] (the author goes on to argue that both testing and verification of the calculus are needed).

11. RTCA. Software considerations in airborne systems and equipment certification. Guideline DO-178B, Radio Technical Commission for Aeronautics, 1992.
12. Christoph Sprenger. A verified model checker for the modal μ-calculus in Coq. In *Proc. TACAS '98*, volume 1384 of *LNCS*, pages 167–183. Springer, 1998.
13. Kurt Stenzel. *Verification of Java Card Programs*. PhD thesis, Universität Augsburg, 2005.
14. Heikki Tauriainen and Keijo Heljanko. Testing LTL formula translation into Büchi automata. *International Journal on Software Tools for Technology Transfer (STTT)*, 4(1):57–70, October 2002.
15. Kerry Trentelman. Proving correctness of JavaCard DL taclets using Bali. In B. Aichernig and B. Beckert, editors, *Proceedings, 3rd IEEE International Conference on Software Engineering and Formal Methods (SEFM)*, 2005.
16. David von Oheimb. *Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic*. PhD thesis, Technische Universität München, 2001.