To appear in Postproceedings, IFIP Working Conference on Verified Software: Theories, Tools, Experiments (Zurich, Switzerland, Oct 10–13, 2005)

Can We Build an Automatic Program Verifier? Invariant Proofs and Other Challenges

Myla Archer

Code 5546, Naval Research Laboratory,
Washington, DC 20375
archer@itd.nrl.navy.mil

Abstract. This paper reviews some common knowledge about establishing correctness of programs and the current status of program specification and verification. While doing so, it identifies several challenges related to the grand challenge of building a verifying compiler. The paper argues that invariants are central to establishing correctness of programs and that thus, a major part of an automatic program verifier must be automated support for verifying invariants, a significant problem in itself. The paper discusses where the invariants come from, what can be involved in establishing that they hold, and the extent to which the process of finding and proving invariants can be automated. The paper also discusses several of the related challenges identified, argues that addressing them would make the significance to global program behavior of feedback from a verifying compiler clearer, and recommends that many of them should be included within the scope of the grand challenge.

1 Introduction

In undertaking to construct and exploit an automatic program verifier, one must first focus in on the problems to be solved. There are several natural questions that arise, e.g.:

- What does it mean to verify a program?
- What does it mean for a program to be correct? (Is "verified" sufficient?)
- Assuming program verification involves proving a set of properties:
  - What types of properties are to be established?
  - Where do the properties come from?
  - Are the properties capable of automatic proof?
- Finally, what support should be provided by the automatic program verifier to allow a user to best exploit it?

This paper will explore these issues, summarizing some common knowledge about program specification, verification, and correctness, and arguing that one major problem on which to focus is the automation of invariant proofs. The paper will then note several additional challenges related to establishing invariants

and other correctness properties of programs, and take the position that, given the numerous challenges, the notion of a verifying compiler as a grand challenge should be given a broad scope.

Sections 2 through 5 discuss the questions listed above. Section 6 discusses related challenges. Finally, Section 7 presents some conclusions and discusses our current and future research that relates to some of the challenges.

2 What Does It Mean to Verify a Program?

Program verification can be approached in more than one way. Two major approaches are *assertion-based verification* and *model-based verification*. As will be expanded on below, to be used for complete program verification, the second (model-based) approach needs to be combined with the first (assertion-based) approach.

Assertion-Based Verification. Since the seminal work in [12] and [7], program verification is often thought of in terms of assertions that can be proved to hold at various points in the program. In particular, for programs designed to run to completion while performing some computation, assertions at the beginning and end of the program can be used to specify the expected result of the computation. The approach is also valid for programs that run indefinitely; in this case, assertions (about input and output streams) before reads and after writes can be used to specify the expected visible behavior of the program.

Model-Based Verification. In model-based program verification, desired program properties are specified by way of a model. Certain programs, particularly ones intended to run indefinitely, are primarily intended to react to events in their environment. Such programs are often better specified not by providing assertions at points in the program but by giving an operational model, usually accompanied by invariant properties of the model. This is the approach used, for example, in SCR (Software Cost Reduction) [14], TIOA (Timed Input/Output Automata) [18, 13] and other software development tools. What then needs to be established in verifying the program is that it refines the specification in the sense that there is a refinement mapping or relation from a state machine representation of the program to the state machine of the operational model. When the operational model (specification) is intended to capture all or most of the intended behavior of the program, verification that the program refines the specification can be done by relating preconditions and postconditions in the model to assertions known to hold at appropriate locations in the program. Here, rather than serving as the program specification, the assertions serve as proof obligations about the program. The benefit of establishing a refinement relation from program to model is that any properties proved of the model will translate into properties of the program. The "refinement" approach can be generalized to allow alternative notions of implementation of a specification: e.g., forward simulation. This approach (establishing an implementation) is one version of model-based verification.

To many, the term "verification" principally means another form of model-based verification: model checking. Model checking has been used more often in the context of hardware verification than software verification, but recent advances such as automated abstraction refinement (see, e.g., [8]) have extended the degree of its applicability to software. When used for true verification rather than testing for counterexamples, model checking involves exhaustively examining a set of cases that covers all the reachable states of the model in some manner (individually, or intelligently grouped). For this reason, the size of the state space becomes a limitation, and model checking for program verification usually compares a very spare model of the program to a second model that captures one property (or a small set of properties) at a time.

Models used to capture just one or a small set of properties rather than the full functional behavior of a program can be termed *property models*. In verification using model checking, the specification to be met is given as a set of properties that are represented (directly or by transformation) as property models. Thus model checking for full program verification requires 1) a sufficient set of property models to adequately specify the program; 2) for each property model, an abstract model of the program that can be given as input to the model checker together with the property model to establish a refinement; and 3) some form of proof that each abstract program model used is a correct abstraction of the program—i.e., that the program refines or implements the abstract model. The use of property models is not limited to the model checking context: Property models are also important for capturing properties of full operational models of programs (as, for instance, in TIOA—see, e.g., [22, 23]).

Verification, Specification, and Correctness. Both verification methods discussed above show that a program *satisfies a specification*. However, as will be argued in the next section, this is not necessarily the same as showing that the program is "correct".

3 What Does It Mean for a Program to Be Correct?

As noted in Section 2, the term "program verification" in any sense means establishing that the program satisfies a specification. The specification may be defined by assertions associated with various points in the program, by an operational model to which the program must conform, or by other assertions about the program as a whole (such as liveness, or absence of deadlock or livelock), which often are captured as specialized property models. However, *correctness* of a program means that the program's behavior matches a set of (behavioral) *requirements*. Thus, for verification of a program to be equivalent to establishing the program's correctness, it is necessary for the specification against which it is verified to capture these requirements.

For some programs, the requirements are clear. For example, a program that sorts a list needs to take a list as input and produce a sorted version of the list as output. For simple programs of this type, assertion-based verification is an appropriate approach.

For a very complex program, e.g., a graphical editor, the requirements for completely correct behavior are equally complex, and it can even be unclear what correctness ought to mean, precisely. A specification, even an operational model, that captures all the desired behavior can be so complex as to make reasoning about it in full detail an intractable problem. For such programs, one may be most interested in only some particular subset of the required behavior. This subset may cover "good" behavior from the user's and operating system's point of view: Will the program terminate unexpectedly due to a segmentation fault? Are there possible buffer overflows or deadlocks? Many properties of this type can be captured in a straightforward way as program assertions.

Another subset of required behavior that is of interest as a program correctness criterion (short of full functional correctness) is security relevant behavior. For cases in which only the security relevant behavior is considered critically important, correctness can mean conformance to a particular security model, and thus model-based verification is especially appropriate.

Whatever the verification method used, verification will only establish correctness of a program if the specification the program is verified against correctly captures the requirements for the program. But, it is only possible to capture requirements in a well-formed specification if they are consistent. Completeness of the requirements can also be particularly important, e.g., if all exceptional behavior must be described. Thus, analysis of well-formedness properties of requirements specifications has a role in the overall effort of establishing the correctness of programs.

4 Properties: Formulation and Proof

4.1 What Types of Properties Are to Be Established?

All the program correctness properties mentioned above can be formulated as invariants of some state machine. The simplest category from the point of view of proof is that of state invariants: program assertions, absence of deadlock, and many specified properties of models fall in this category. Conformance to a model can also be cast as a state invariant of a composition automaton (representing some composition of the model and the program). Almost as simple are safety properties, which involve at most a bounded sequence of transitions. More difficult to prove are liveness properties, which can involve reasoning about an unbounded sequence of states and may involve some fairness assumptions.

4.2 Where Do the Properties Come From?

In the interest of separation of concerns, one can assume, in tackling the challenge of building an automatic program verifier, that the properties that must be established of the program are given. However, it is clear that an automatic verifier will not be much help in establishing program correctness if properties that imply its correctness have not been formulated by someone. Thus, a related

challenge is to persuade developers (or other stakeholders in a piece of software) to specify in some form what the software is to do (i.e., its required behavior). A further related challenge is to create a tool that, given appropriate information, can derive assertions about a program to be used by an automatic program verifier from assertions about an abstract model of the program's behavior.

In the context of asserted programs, there has been some work [11] on dynamically discovering likely program invariants that could produce some of the needed assertions in a program (which would then be subject to proof). There has also been work on generating known invariants, starting from [9] and [10], which consider program assertions. Later work includes [4], which also considers program assertions, and [15], which considers invariant properties of specifications. Although these approaches can help furnish some of the assertions, the connection between the assertions and program correctness would need to be established by someone who understands what the program is supposed to do, or how a model is supposed to behave. Creating automated support for generating program assertions from assertions about a model appears to be an open problem.

4.3 Are the Properties Capable of Automatic Proof?

Some program assertions can be established without induction: e.g., input assertions can be assumptions, other assertions can be established through weakest precondition computations, and further assertions can be established from existing ones by the application of decision procedures. A challenge in this connection is to develop additional decision procedures to be integrated into existing ones that can handle data types (beyond numerical, boolean, and enumerated types) for which many assertions are decidable.

However, for certain classes of assertions, induction is required. For example, induction is generally needed to establish loop invariants. Induction in some form is also generally needed to establish liveness properties. For a finite model, one can sometimes avoid induction: properties of finite models can (if state explosion is manageable) be established by exhaustive search (model checking). However, establishing invariant properties of infinite (and sometimes, very large finite) models requires theorem proving, and, typically, induction.

Thus, even though some program properties can be established by other means, a general truly *automatic* program verifier would need to be able to do induction proofs automatically. A completely general approach to doing this is not possible, because the general problem of establishing whether an assertion is an invariant is undecidable. In principle, provided the induction scheme is known (as is the case, e.g., for state invariants, where induction is over the reachable states), and provided the base and induction cases can be stated in first order logic, *valid* invariants can be established by induction automatically. However, efficiency is an issue; so is the problem that some properties being checked are false—as may be the case for the induction step when one is trying to prove a possibly true invariant by proving that it is inductive. In particular, proofs by induction of invariants also often require strengthening of the invariants, a process

that is not always automatable. Strengthening *can* be automated, to a degree, as has been illustrated in SCR. Note that an equivalent approach to strengthening is the introduction of additional invariants as lemmas. In the context of SCR, it has been possible to create an induction proof strategy that uses automatically generated invariants [15] as lemmas and that proves many properties of SCR specifications automatically without user guidance; see, e.g., [20].

Thus, automating induction proofs of program properties is itself a challenge. The goal would be to create a technique that would cover the kinds of assertions that normally arise in practice. Techniques such as proof planning with rippling [6, 5] have had some success, but are still not sufficiently universal.

Mechanical proofs—by induction or otherwise—of correctness properties of abstract models are often best constructed interactively. This is because for abstract models, correctness properties can contain quite complex predicates (e.g., the `Authenticated` predicate in the basic TESLA protocol model in [2] involves existential quantifiers and is recursively defined) and are potentially higher-order. As shown by our experience with TAME [3], efficient interactive construction of proofs can be made more feasible if an appropriate special domain tool or prover interface is provided. (TAME is discussed further in Section 5, and in more detail in Section 7.)

5 How to Exploit an Automatic Program Verifier?

As has been noted above, an automatic verifier presupposes some form of specification against which to verify the program. A user better equipped to specify is thus better equipped to verify. But such a user is also better equipped to *test*. To state the obvious: the user should test the assertions before using the program verifier, because verification is expensive; only after one has evidence that a set of properties is likely correct should one undertake to prove the properties. Thus, a program verifier is best used in conjunction with a testing tool.

Equally important to knowing that a program has certain properties is knowing *why* it has those properties. For example, one usually does not want a property to be vacuously true, as might happen (in a program) for the postcondition of an unintentionally nonterminating loop, or (in a model) when all preconditions of transitions are false. Thus, in addition being able to prove properties, it is desirable for the verifier to produce some degree of proof explanation. A variety of theorem proving techniques provide some form of explanation. Several automatic proof techniques provide proof explanation; examples include ACL2 [16, 17] and approaches based on proof planning such as [19]. Our tool TAME [3] provides explanations for invariant proofs produced with interactive guidance.

However explanations are produced, the same techniques used in proof explanation can be adapted to provide some explanation of proof failure—i.e., what point and proof goal did the proof reach when the automatic verifier was unable to continue? When the automatic verifier is unable to verify a program, the next action needs to be either modification of the program or modification of the specification. While performing the proper corrective action is an art form, un-

derstandable feedback from the automatic verifier is an important prerequisite to making the correction.

Addressing some other challenges related to the automatic program verifier would allow the automatic verifier to be exploited as fully as possible as a tool for establishing program correctness. The next section discusses a number of these challenges and notes how addressing them would help.

6 Related Challenges

As noted above, the challenge of building an automatic invariant prover is a part of the challenge of building an automatic program verifier. The challenge of improving and expanding the scope of decision procedures also falls into this category. But there are other challenges that, if addressed, would increase the usefulness of an automatic verifier. Two have already been mentioned.

First, it would be helpful if software developers could be convinced to provide some form of specification of what the software is supposed to do. With respect to low level specification, this is not an unreasonable hope: for example, the inclusion of assertions with C and Java code is provided for and beginning to come into practice. It is likely unrealistic to hope that *all* software developers will provide requirements specifications that capture the intended behavior of the code. However, when the correctness of the code is essential, such specifications are more likely to be developed.

Next, as noted in Section 2, when an operational specification of the required behavior is available, one approach to establishing that a program refines this specification is to relate assertions in the specification (e.g., pre- and postconditions associated with transitions) to assertions in the code. This leads to a second challenge: automating as far as possible the mapping of assertions in the specification to assertions in the code, based on information provided by the user that relates program states to abstract (specification-level) states and program segments to transitions in the specification. Even if there are no decidability issues here, deriving code assertions from assertions in the specification may not be straightforward in cases where there is not a direct relation between individual variables in the program to variables in the specification, or when the relation of program states to abstract states is not a simple mapping. The design and development of generic tool support for this part of the program verification process can thus be complicated problem.

There is a third, additional challenge related to the second: To develop sound procedures for transforming requirements specifications expressed in forms such as natural language or a set of logical properties into an operational requirements specification. Implicit in this challenge is the ability to analyze requirements specifications for such properties as consistency and completeness.

A further, fourth challenge is to develop methods that can be applied by developers in designing programs for verifiability, and induce the developers to use them. While some such guidance already exists (e.g., avoid certain constructs), this guidance is mostly of a "local" nature. An open question is whether guidance

can be provided for structuring programs so that particular properties (e.g., for security, separation of data) are easier to establish.

Addressing these challenges would help ensure first, that the automatic program verifier is proving properties of interest and second, that the automatic verifier's task is made as simple as possible.

7 Conclusions, Recommendations, and Plans for the Future

A verifying compiler that verifies assertions in programs is only part of the answer to the problem of producing verifiably correct programs. The challenge of building an automatic program verifier can be conceived more generally as covering not only a (possibly interactive) mechanical verifier of assertions in programs but a mechanical verifier (almost necessarily interactive!) that a program conforms to a model. For either the program-assertion-based or model-based verification style, the automation of (or, failing full automation, mechanized support for) proofs of invariants, and in particular induction proofs, will play a central role.

This paper has identified several related challenges to be met; some of them are directly implied by the challenge of building an automatic program verifier. Others are associated with additional parts of the process of establishing correctness properties of programs. Because addressing these others will increase the effectiveness of the automatic program verifier, it is worth considering including them as part of the overall challenge. Below is a summary of our current and future work that does (or will) address *some* of the related challenges.

The challenges identified that relate to requirements specifications are addressed in part by the SCR tool set [14]. In particular, the SCR tool set provides for consistency and completeness checking of SCR specifications, which define operational models. Moreover, SCR specifications can include an associated set of properties. While the tool set does not provide a means for transforming properties to an operational model, the associated set of properties can be used to express a property-based version of the specification, and several tools in the tool set can be applied to showing consistency between the properties and the operational specification.

Two of the other challenges identified above are addressed to some degree for model-based verification by the tool TAME (Timed Automata Modeling Environment) [1, 3], a specialized interface to PVS [24] for proving properties of timed I/O automata [21, 22]. In particular, by providing specification templates, TAME attempts to make specification of models easier.[1] It also partially automates proofs of invariants, including state invariants, transition invariants, and abstraction properties such as refinement and forward simulation [23] by providing a set of high level proof steps that allow a proof sketch to be mechanically checked. For SCR specifications, TAME can prove many invariant properties

[1] The SCR tool set also aims at simplifying the specification of models, but uses a quite different approach based on tables.

automatically. TAME provides user feedback for failed proofs both inside the prover at the point of a proof dead end and in saved TAME proofs through structure, proof step names, and automatically generated comments. A prototype proof tool that translates TAME proofs into English has been implemented. Work is continuing on improving TAME in all these areas.

It is also planned to extend the work on TAME by increasing the degree to which proofs of invariants can be automated. This will be done by 1) developing techniques that can prove more invariants automatically by building on previously proved invariants, by finding useful alternative instantiations of the inductive hypothesis, and so on; and 2) exploring the possible use of techniques such as rippling in proving invariants of TAME models.

Other plans for the near future include:

- The use TAME or a similar "special domain" PVS interface to model some medium-sized programs and establish their correctness. The goal is to build on the techniques used in TAME to permit program verification on a level nearer to the level of program assertions.
- Development of prototype automated support for translating assertions at the model level into assertions at the program level.

Some interesting lessons, and perhaps some new associated challenges, are likely to result from these efforts.

Acknowledgements

I thank Elizabeth Leonard and Sandeep Shukla for helpful discussions, and Elizabeth for comments on an earlier version of this paper. I also thank the anonymous reviewer whose thoughtful observations have helped to improve the paper.

References

1. Myla Archer. TAME: Using PVS strategies for special-purpose theorem proving. *Annals of Mathematics and Artificial Intelligence*, 29(1-4):139–181, 2000. Published Feb. 2001.
2. Myla Archer. Proving correctness of the basic TESLA multicast stream authentication protocol with TAME. In *Workshop on Issues in the Theory of Security (WITS'02)*, Portland, OR, Jan. 14–15 2002.
3. Myla Archer, Constance Heitmeyer, and Elvinia Riccobene. Proving invariants of I/O automata with TAME. *Automated Software Engineering*, 9(3):201–232, 2002.
4. Nikolaj Bjørner, I. Anca Browne, and Zohar Manna. Automatic generation of invariants and intermediate assertions. *Theoretical Computer Science*, 173(1):49–87, 1997.
5. A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence*, 62:185–253, 1993.
6. Alan Bundy. The use of proof plans for normalization. In R. S. Boyer, editor, *Automated Reasoning: Essays in Honor of Woody Bledsoe*, volume 7 of *Automated Reasoning Series*, pages 149–166. Kluwer, 1991.

7. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, 1969.
8. Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Proc. 12th International Conference on Computer-Aided Verification (CAV)*, pages 154–169. Springer-Verlag, 2000.
9. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. 1977 Symp. on Principles of Programming Languages*, January 1977.
10. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables. In *Proc. 1978 Symp. on Principles of Programming Languages*, January 1978.
11. M. Ernst. *Dynamically Discovering Likely Program Invariants*. PhD thesis, Univ. of Washington, 2000.
12. R. W. Floyd. Assigning meanings to programs. In *Proceedings of Symposia in Applied Mathematics*, volume 19 of *Mathematical Aspects of Computer Science*, pages 19–32. American Mathematical Society, 1967.
13. Stephen Garland. TIOA User Guide and Reference Manual. Technical report, MIT CSAIL, Cambridge, MA, 2006.
URL `http://tioa.csail.mit.edu`.
14. C. Heitmeyer, M. Archer, R. Bharadwaj, and R. Jeffords. Tools for constructing requirements specifications: The SCR toolset at the age of ten. *International Journal on Computer System Science and Engineering*, 20(1):19–35, January 2005.
15. Ralph Jeffords and Constance Heitmeyer. Automatic generation of state invariants from requirements specifications. In *Proc. 6th International Symposium on the Foundations of Software Engineering (FSE-6)*, Orlando, FL, Nov. 1998.
16. M. Kaufmann, P. Manolios, and J Strother Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.
17. M. Kaufmann, P. Manolios, and J Strother Moore, editors. *Computer-Aided Reasoning: Case Studies*. Kluwer Academic Publishers, 2000.
18. D. Kaynar, N. A. Lynch, R. Segala, and F. Vaandrager. A mathematical framework for modeling and analyzing real-time systems. In *The 24th IEEE Intern. Real-Time Systems Symposium (RTSS)*, Cancun, Mexico, December 2003.
19. Manfred Kerber, Michael Kohlhase, and Volker Sorge. Integrating computer algebra into proof planning. *Journal of Automated Reasoning*, 21(3):327–355, 1998.
20. James Kirby, Jr., Myla Archer, and Constance Heitmeyer. SCR: A practical approach to building a high assurance COMSEC system. In *Proc. 15th Annual Computer Security Applications Conference (ACSAC '99)*. IEEE Comp. Soc. Press, Dec. 1999.
21. N. Lynch and M. Tuttle. An introduction to Input/Output automata. *CWI-Quarterly*, 2(3):219–246, Sept. 1989. Centrum voor Wiskunde en Informatica, Amsterdam, Netherlands.
22. N. Lynch and F. Vaandrager. Forward and backward simulations – Part II: Timing-based systems. *Information and Computation*, 128(1):1–25, July 1996.
23. Sayan Mitra and Myla Archer. PVS strategies for proving abstraction properties of automata. *Electronic Notes in Theor. Comp. Sci.*, 125(2):45–65, 2005.
24. N. Shankar, S. Owre, J. M. Rushby, and D. W. J. Stringer-Calvert. PVS Prover Guide, Version 2.4. Technical report, Comp. Sci. Lab., SRI Intl., Menlo Park, CA, Nov. 2001.