The Importance of Non-theorems and Counterexamples in Program Verification

Graham Steel

School of Informatics,
University of Edinburgh,
Edinburgh, EH8 9LE, Scotland.
graham.steel@ed.ac.uk
http://homepages.inf.ed.ac.uk/gsteel

Abstract. We argue that the detection and refutation of non-theorems, and the discovery of appropriate counterexamples, is of vital importance to the Grand Challenge of a Program Verifier.

1 Introduction

In this essay, we make a case for the inclusion of non-theorem (i.e. incorrect conjecture) detection and counterexample generation as a core theme in the research program of the Grand Challenge of a Program Verifier. We will argue that:

- Research in program verification technology will be hindered if counterexample generation research does not catch up and keep pace. We will give the reasons for this in §2.
- Detecting false conjectures and generating counterexamples to them is a fascinating scientific challenge in itself. We will argue this in §3.
- Deduction based approaches for verification, which offer perhaps the best chance of achieving the goals of the grand challenge, must improve in their handling of non-theorems if they are to compete with model checking approaches, which are already able to give counterexamples to false verification conditions.

Our arguments will be followed in §4 by a look at previous and current research on the topic, including our own efforts in the Mathematical Reasoning Group at the University of Edinburgh.

Since the areas of application are the same, and since the problems of program verification and counterexample generation are in a sense 'dual', it seems logical that they should be thought of as part of the same Grand Challenge. Note that we fully support the view that the final goal of the project must be a program verifier, not a system that just finds more and more bugs. However, we will argue in this paper that to achieve this goal, the dual problem of non-theorem detection and counterexample generation must be given due attention.

2 The Importance of Counterexample Generation

Even the most diligent and expert programmer will very rarely write a bug-free first version of a program. The majority of calls to a program verifier will therefore involve an incorrect program. If we want program verification tools to become widely used, they must deal with buggy programs in a competent manner. Simply presenting failed verification conditions or open subgoals is not sufficient, since it leaves the user with no idea whether a bug has been found, or whether the verification system is simply unable to dispose of that particular proof obligation. What is required is a system which can not only detect an incorrect conjecture, but also supply a counterexample to allow the user to locate the flaw.

Non-theorem detection is also important to the internal processes of a verification tool. Automated proof attempts, particularly when induction is involved, will frequently require the conjecturing of lemmas and generalisations. Often these conjectures will be false, and it is vital that we detect these cases and prune our search space appropriately.

There is a further, pragmatic argument for the inclusion of counterexample generation within the Grand Challenge. Great scientific problems, from landing a man on the moon to producing a computer to beat a grand master at chess, have been solved by making iterative improvements to a prototype and learning from failures. To apply this methodology to our Grand Challenge, we must encourage participation from industry, in order to ensure a supply of case study material, to get feedback on the tools we produce, and where appropriate, to try to influence software engineering practice. For this, we need to ensure that we are able to make a financial argument for the use of our tools even when they are at a prototype stage, and unable to deliver a fully verified end product. Being able to detect and present counterexamples that allow bugs to be identified is a way of ensuring some payback to our industrial partners. The current industrial preference for model checking over theorem proving can partly be explained by the ability of model checkers to present counterexample traces.

3 The Challenge of Counterexample Generation

The non-theorems that arise in program verification can sometimes be easy to find. They often occur close to the base case of a recursive data type, and evaluating the conjecture at some small values can be sufficient to detect the bug. However, there are a large number of cases where the counterexample is a far more subtle object, and it is here that the scientific challenge lies. An example of a success story in this area is the application of formal methods to discovering attacks on security protocols. Here, the attacks are counterexamples to conjectures about the security of a system. The counterexamples may be quite large, for example up to 14 messages in [18], and may require an intruder to exploit quite subtle flaws in the protocol. Further challenges remain: for example, taking into account the mathematical properties of the cryptofunctions in use.

Other problems are well outside the scope of current techniques. For example, the book 'Numerical recipes in C' contains Knuth's code for a (pseudo-)random number generator, [14, p. 283]. The code is designed to return a floating point number between 0 and 1. However, if certain very large seed values are used, the code can return a number outside of this range. With seed value 752 005 709[1], 13 out of the first 10,000 calls return a value very significantly outside the required range. The large seed value may seem ridiculous, but if you were seeding your generator on the number of seconds since 1 Jan 1970, this value would have occurred as a seed during 1993. The generation of these counterexamples remains well beyond the capabilities of current tools, and presents a substantial scientific challenge.

4 Survey of Counterexample Generation

Given how important the detection and presentation of counterexamples is to applications of formal methods, it is surprising how comparatively little attention it has received. A community of researchers interested in the subject is now beginning to emerge, with the second 'Workshop on Disproving' due to be held at CADE 2005[2].

Much past research has focused on fast generation of counterexamples to conjectures which are in some sense 'obviously' false. Model generators like MACE, [9], and FINDER, [17], can be used to enumerate finite domains to search for counterexamples. The Isabelle theorem prover, [11], now includes a tactic to search for small counterexamples, called quick-check, [5]. For infinite data structures with recursive definitions, methods have been proposed by Protzen, [15], and Reif, [16]. Both can deal with small problems quite effectively, but are not suitable for large counterexamples, or for domains that cannot be easily enumerated. Model generation has also been proposed as a method for refuting non-theorems, [1, 20].

Model checking can be a very effective method for finding counterexamples to false verification conditions, particularly in finite domains. As we have already remarked, the fact that model checking can produce counterexamples as well as provide guarantees is one reason for its increasing popularity in industry. Many researchers are now working on extending model checkers to non-finite domains, using 'lazy' and 'on-the-fly' techniques to construct the infinite models as they are checked (e.g. [4]), with good results. Another large branch of current model checking research is in counterexample guided abstraction refinement, [6, 3]. Here, the processing of counterexamples is used to guide management of the level of detail that is taken into account when attempting verification of a program, balancing tractability of the model checking problem against over-abstraction. Theorem provers can be used to check the feasibility of counterexam-

[1] Knuth stated in the specification that the seed value can be any (large) number under 1 000 000 000. This is a known bug - the Numerical Recipes in C website contains a patch to fix the code.

[2] `http://www.cs.chalmers.se/~ahrendt/cade20-ws-disproving/`

ple traces, [2]. A further current direction involves passing unproven conjectures from a theorem prover to a model checker to search for counterexamples, [13].

Our work in Edinburgh has lead to the development of the CORAL system[3], which refutes incorrect inductive conjectures using a first-order theorem prover adapted to follow the 'proof by consistency strategy' of Comon and Nieuwenhuis, [7]. Its major successes have been in the field of protocol analysis, where it has been used to discover 6 previously unknown attacks on 3 group protocols, [18, 19]. The protocols are modelled inductively following Paulson, [12], and the attacks found as counterexamples to security conjectures. CORAL proved to be particularly suitable for group protocols because they can be formalised very naturally in an inductive model. This is something that rival approaches, such as model checking, struggle with. In future, we plan to experiment with CORAL in other areas of formal verification.

Many of the big problems remain unsolved. For example, how to deal adequately with arithmetic, or how to explore very large or non-trivially enumerable spaces for counterexamples. Some current directions include trying to use more information from failed proof attempts to guide the counterexample search, [8], and to suggest patches for incorrect conjectures, [10].

5 Summary

We have argued that disproof and counterexample generation are vital areas for research in the development of practical program verification systems. There remain many exciting open problems, and milestones to pass, such as the automatic generation of counterexamples for Knuth's random number bug, described above. Non-theorem detection therefore deserves to be included as a core theme in the Grand Challenge of a Program Verifier.

References

1. W. Ahrendt. Deductive search for errors in free data type specifications using model generation. In A. Voronkov, editor, *18th Conference on Automated Deduction*, volume 2392 of *Lecture Notes in Computer Science*, pages 211–225. Springer, 2002.
2. T. Ball, B. Cook, S. Lahiri, and L. Zhang. Zapato: Automatic theorem proving for predicate abstraction refinement. In Rajeev Alur and Doron Peled, editors, *CAV*, volume 3114 of *Lecture Notes in Computer Science*, pages 457–461. Springer, 2004.
3. T. Ball and S. Rajamani. The SLAM toolkit. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *CAV*, volume 2102 of *Lecture Notes in Computer Science*, pages 260–264. Springer, 2001.
4. D. Basin, S. Mödersheim, and L. Viganò. An on-the-fly model-checker for security protocol analysis. In *Proceedings of the 2003 European Symposium on Research in Computer Security*, pages 253–270, 2003. Extended version available as Technical Report 404, ETH Zurich.

[3] `http://homepages.inf.ed.ac.uk/gsteel/coral`

5. Stefan Berghofer and Tobias Nipkow. Random testing in isabelle/hol. In *2nd International Conference on Software Engineering and Formal Methods (SEFM 2004)*, pages 230–239, 2004.
6. Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the Association for Computing Machinery*, 50(5):752–794, 2003.
7. H. Comon and R. Nieuwenhuis. Induction = I-Axiomatization + First-Order Consistency. *Information and Computation*, 159(1-2):151–186, May/June 2000.
8. L. A. Dennis. The use of proof planning critics to diagnose errors in the base cases of recursive programs. In W. Ahrendt, P. Baumgartner, and H. de Nivelle, editors, *IJCAR 2004 Workshop on Disproving: Non-Theorems, Non-Validity, Non-Provability*, pages 47–58, 2004.
9. W. McCune. A Davis Putnam program and its application to finite first order model search. Technical report, Argonne National Laboratory, 1994.
10. R. Monroy. Predicate synthesis for correcting faulty conjectures: The proof planning paradigm. In *Automated Software Engineering*, pages 247–269, 2003.
11. L.C. Paulson. The foundation of a generic theorem prover. *JAR*, 5:363–397, 1989.
12. L.C. Paulson. The Inductive Approach to Verifying Cryptographic Protocols. *Journal of Computer Security*, 6:85–128, 1998.
13. L. Pike, P. Miner, and W. Torres. Model checking failed conjectures in theorem proving: a case study. Technical Report NASA/TM–2004–213278, NASA Langley Research Center, November 2004. Available at `http://www.cs.indiana.edu/~lepike/pub_pages/unproven.html`.
14. William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, 1992.
15. M. Protzen. Disproving conjectures. In D. Kapur, editor, *11th Conference on Automated Deduction*, pages 340–354, Saratoga Springs, NY, USA, June 1992. Published as Springer Lecture Notes in Artificial Intelligence, No 607.
16. W. Reif, G. Schellhorn, and A. Thums. Flaw detection in formal specifications. In *IJCAR'01*, pages 642–657, 2001.
17. J. Slaney. *FINDER: Finite Domain Enumerator*. Australian National University, 1995. Available from `ftp://arp.anu.edu.au/pub/papers/slaney/finder/finder.ps.gz`.
18. G. Steel and A. Bundy. Attacking group multicast key management protocols using CORAL. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 125(1):125–144, 2004. Also available as Informatics Research Report EDI-INF-RR-0241. Presented at the ARSPA workshop 2004.
19. G. Steel, A. Bundy, and M. Maidl. Attacking a protocol for group key agreement by refuting incorrect inductive conjectures. In D. Basin and M. Rusinowitch, editors, *Proceedings of the International Joint Conference on Automated Reasoning*, number 3097 in Lecture Notes in Artificial Intelligence, pages 137–151, Cork, Ireland, July 2004. Springer-Verlag Heidelberg.
20. T. Weber. Bounded model generation for isabelle/hol. In *IJCAR 2004 Workshop on Disproving - Non-Theorems, Non-Provability*, pages 27–36, Cork, Ireland, 2004.