Verification by theorem proving: issues and challenges

PROVISIONAL EXTENDED ABSTRACT
(warning: written in a hurry before the talk was prepared)

Michael J. C. Gordon

University of Cambridge Computer Laboratory
William Gates Building, JJ Thomson Avenue, Cambridge CB3 0FD, U.K.
Email: `mjcg@cl.cam.ac.uk` Web: http://www.cl.cam.ac.uk/~mjcg

A short and very partial history

General purpose theorem proving systems have been under development for a long time. ACL2 and HOL are based on developments that started 30 years ago, and Isabelle and PVS on work started about 20 years ago. Other major systems like Nuprl, Coq, ProofPower and EVES all have a long history.

The early research with systems like these often focused on heroic proofs of large hardware or software systems or difficult mathematical theorems. By the early 1990s it was clear that, given enough effort, just about anything could be represented and proved. It also became clear that, at least then, the effort and timescales needed for the complete formal verification of most industrial scale systems was too great to be cost-effective. However, a lasting and positive by-product of the early proofs was the creation of considerable infrastructure for modelling and automated reasoning in a wide variety of domains. Much of standard discrete and continuous mathematics is now formalised and supported by powerful tools.

During the 1980s and 1990s there were dramatic advances in automatic verification (e.g. decision procedures, model checking, BDDs, SAT). These methods required less heroic efforts for their deployment and quickly found industrial applications, sometimes as "point tools" for a narrow range of problems. From the mid 1990s, starting with pioneering work using PVS, there has been a concerted effort to combine automatic methods with user guided theorem proving. This built on earlier experience gained from integrating decision procedures into the Boyer-Moore system. A currently hot application is using theorem proving to abstract from undecidable infinite state models to decidable finite state ones, which can then be model checked. Another application is to employ theorem proving to glue together the results of hundred of model checking runs to ensure all cases are covered. Intel's work using Forte to manage STE checking is an example of an industrial success story in combining theorem proving with automatic verification.

Connecting logics

ACL2 uses first-order logic plus induction schemes, HOL, ProofPower and Isabelle/HOL use classical higher order logic, PVS uses classical higher order logic augmented with dependent subtypes and products, Isabelle/ZF and Eves use ZF set theory, Coq and Nuprl use constructive type theories. It seems that all these logics are adequate for modelling systems, though some might be more natural than others for certain background mathematics (e.g. real analysis). I think it is only realistic to "let one thousand flowers bloom", but each of the major theorem prover communities is built on hundreds of person-years of development, and it would be good if we could all exploit each others work. There have been some preliminary experiments on connecting Nuprl to HOL and ACL2 to HOL which are promising. Should we develop new logic interchange formats (e.g. XML schemas) or try to exploit existing work like OpenMath, MathML or MBase?

Theorem provers as tool integration platforms

A lot of exciting recent work aims to combine traditional non-formal verification methods like simulation with formal methods. It is claimed that having a standard property specification language (PSL) and a standard set of directives for invoking checking will enable verification tools to be more easily connected together. Should we also have standard ways of interfacing theorem provers to EDA and software engineering tools? Projects such as PROSPER and MathWeb have addressed the middleware problem of connecting theorem provers together and connecting provers to other tools.

Specialised verification methods use specifications in different formats, each with its own semantics. If several methods have been used on different parts of a design it might be hard to tell exactly what has been verified, and whether there are coverage gaps. One role a general theorem prover can play is to provide an integration platform. By describing the specification logics of the various methods in a uniform general logic, one can hope to combine the results of the methods to deduce an overall picture of what has been proved. Thus a theorem prover can be a kind of logic programming system, but it need not be based on a single uniform method like resolution, but can exploit heterogeneous combinations of formal and conventional verification. In 1973 Pat Hayes said:

> "... the usual sharp distinction which is made between the processes of computation and deduction, is misleading. An interpreter for a programming language, and a theorem-proving program for a logical language, are structurally indistinguishable. Important benefits, both practical and theoretical, are obtained by combining the best of both methodologies."

Thirty years later this idea is still worth thinking about.

Proof of correctness versus debugging

Theorem proving, and other formal verification methods, are sometimes marketed as a kind of advanced debugging. If the goal is to fit into the normal debugging flow, then speed and ease of use are paramount and soundness is less important (a bug is a bug, whether or not the tool that found it is sound). One would like to know when one can stop looking for more bugs, and correctness tells us this more reliably than coverage metrics, but a list of bugs is better than a failed correctness proof. Should we try to develop theorems provers that are simultaneously optimised for debugging and for proof of correctness, or are these activities best supported by different kinds of tools?

Efficiency versus soundness

Theorem provers derived from ideas originating in Robin Milner's LCF system (e.g. HOL, ProofPower, Isabelle, Nuprl) take an extreme position on soundness: theorems can only be created by applying sequence of primitive inference rules. This is called "LCF-style" or "fully-expansive" proof. Other systems, such as ACL2 and PVS, allow oracles, such as decision procedures, to determine the truth of classes of formulas. In practise, the disciplined use of oracles is quite safe and it can lead to simpler and more efficient implementations. There may be a few niche applications (e.g. by paranoid government agencies) where the extreme soundness of LCF-style proof is motivated, but for normal commercial applications it seems excessive. In fact, modern versions of HOL and Isabelle allow oracles to be used, though theorems created with them are marked so their logical provenance is explicit. Although the pure soundness motivation may be hard to defend to engineers, LCF-like provers provide a foolproof way of scripting new proof tools. If non-logicians are going to write proof scripts, then having guaranteed soundness is reassuring. Perhaps the fully-expansive discipline is best thought of as analogous to a type discipline, preventing logical errors like type-checking prevents type errors. Relatively efficient methods for programming in a fully expansive style have been developed, so now performance may well be adequate for many tasks.

Case studies and grand challenges

Even though we know that with sufficient effort we can verify anything, there is still value in calibrating the state-of-the-art to provide data for estimating how long verifications will take. This is particularly important for theorem proving, which is getting much more efficient, but has not yet become widely adopted in industry. For example, using established methods, Anthony Fox recently verified a refinement from a programmers model of an ARM6 processor to a micro-architecture model. This took a year. Is that cost-effective? This is data ARM can use to determine if theorem proving is useful.

It is also vital to continue to explore new kinds of example to expand the repertoire of techniques we have available for immediate deployment. In 1990 we had little experience with specifying and verifying complex pipeline architectures. Today we have many methods and exemplar proofs on the shelf. We need to continue to extend the range of examples and techniques, so that the scope of verification by theorem proving continues to broaden and mature. To this end J Moore recently proposed, as a Grand Challenge, the formal verification of a complete embedded system from transistors to software.

Conclusion

General purpose theorem proving is thriving, but it has not yet found many commercial applications. Traditional large scale proofs might become cost effective as methods, infrastructure and processor speeds improve, but there are also other roles for theorem provers besides just proving theorems.