A Practical Perspective on the Verifying Compiler Proposal*

Andrew Ireland
School of Mathematical and Computer Sciences
Heriot-Watt University
Edinburgh, Scotland, UK
a.ireland@hw.ac.uk

Abstract

A personal perspective of the verifying compiler proposal is presented. I outline what I see as the key practical issues that need to be addressed. I focus in particular on theorem proving issues and the role that proof planning can play in building the verifying compiler.

1 Introduction

The dream of being able to routinely verify programs has a long history dating back to 1947 when Goldstine and von Neumann first recognised the potential for assertion based reasoning [22]. However, it was Floyd's inductive assertion method [19] and Hoare's axiomatic basis for programming [30] that popularised program verification research and laid the foundations for today's *verifying compiler* proposal. In this short paper I present a personal perspective on the verifying compiler proposal. I begin with a historical survey of program verification systems and then go on to discuss current research trends. Building upon this background material, I highlight practical issues that need to be addressed. In particular I focus on theorem proving, outlining the role *proof planning* can play in meeting the verifying compiler challenge.

2 A historical survey of program verification systems

Based upon Floyd's inductive assertion method, King [46] was the first to build a program verifier. Others followed [9, 24], but it was the Stanford Pascal Verifier project that produced the first verification system to target a real programming language [51]. A related system called Runcheck [20], provided support in verifying the absence of common run-time errors in Pascal programs. Not surprisingly, this property-based style of verification was more tractable than full verification, *i.e.* partial and total correctness. During this early phase of the research, considerable effort was applied to the problem of generating loop invariants [15, 65, 43, 44, 66, 21, 6]. However, although the generation of loop invariants are a crucial bottle-neck for program verification, little of this work was carried forward.

The early verification systems highlighted issues of complexity that arise when specifying nontrivial programs. To address these issues, the Gypsy Verification Environment [23], AFFIRM [54] and the Hierarchical Development Methodology (HDM) [48] all supported abstraction within their specification languages to varying degrees. In terms of applications, Gypsy and HDM (later extended to EHDM [12]) were two of the most prominent systems to come out of the 1970's. Both systems were applied to significant security projects. Gypsy was the first system to address concurrency issues while HDM supported verification within a broader approach to the software development process. These projects were relatively successful in academic terms. They showed, however, that the goal of being able to

*Thanks to Alan Bundy and Benjamin Gorry for their feedback on an earlier version of this paper. The support of EPSRC (GR/R24081) is also acknowledged, as is the support of our industrial collaborator, Praxis Critical Systems.

routinely verify programs was some way off. A key outcome from this early work was the realisation that the effective integration of decision procedures was crucial to achieving large scale verifications. Probably the most influential contribution was the work by Shostak [61] and Nelson-Oppen [55] on combining decision procedures.

The 1980's gave rise to systems that focused on Ada [50, 28, 2], however, the late 1980's and 1990's saw a decline in program verification research. This decline also marked increased emphasis on design level verification, as popularised by theorem provers such as PVS [59] and ACL2 [45], and model checkers like SMV [53] and SPIN [31]. A notable exception to this trend was SPARK [2], an annotated subset of Ada. SPARK has achieved significant industrial success with safety and security critical applications [47, 29]. The SPARK toolset supports static analysis, including formal verification. Similar static analysis techniques can be found in MALPAS [64]. In terms of verification, the SPARK toolset has been finely tunned for the task of verifying the absence of run-time errors, so called "exception freedom proofs" [7].

3 Current trends in formally based program analysis

The success of model checking and constraint solving techniques has revitalised formal analysis of program code. SLAM [1], Pathfinder [4], Bandera [8] and Alloy [40] are some of the key tools to have emerged. The strength of these tools lies in their ability to find defects. Moreover, they typically focus on pre-defined properties, *e.g.* in the case of SLAM, safety properties that express "API usage rules". As a consequence, programmers are insulated from assertion level specification. In terms of theorem proving, the Extended Static Checker project, and ESC/Java in particular, is worth noting [18]. ESC/Java provides annotations that support static analysis and makes use of a theorem prover to find defects, rather than verify correctness.

Another significant trend is the increased use of commercial automatic code generation tools [3, 11, 52, 62]. While automatic code generation tools can significantly reduce "time to market" they also raise issues of correctness. For commercial and practical reasons, the formal verification of code generation tools is not usually an option. Instead, emphasis has been placed upon proving the correctness of each individual run of the code generator. Pnueli calls this "translation validation" and has applied it to the validation of reactive systems that are restricted to single loops [60]. QinetiQ's Systems Assurance Group, based at Malvern, follow a similar approach and have achieved a high degree of proof automation in certifying Simulink generated code for advanced avionics systems [57]. These successes can, in part, be attributed to the restricted nature of the programs and properties being considered.

4 Practical issues relating to the verifying compiler proposal

The gap between current successes and the goal of building the verifying compiler is significant. Below I focus on practical issues that need to be addressed if this gap is to be bridged.

4.1 Language issues

The choice of programming language(s) targeted by the verifying compiler will have a significant effect on the chances of success. Some programming language features are well known as being problematic to formal reasoning, *e.g.* the volatile keyword in C. The pragmatic approach of identifying useful subsets of an existing languages that are amenable to formal analysis has many advantages, as demonstrated by SPARK. Relatively recent developments in terms of a semantics for C [56] and methods for reasoning about dynamic structures [58] may also prove useful in tackling this issue.

In terms of specifications, assertion based programming is a key feature of the verifying compiler proposal. As noted above, verification that involves pre-defined properties means that programmers are not required to write assertions. However, if the verifying compiler is to support full verification then programmers will be required to provide assertions. In general, however, assertion based programming is still not widely accepted. Testing may provide a handle on some aspects of the problem,

i.e. unit tests may provide a basis for generating assertions. This may seem rather perverse, however, from a practical stand point, programmers understand how to construct unit tests. A novel approach to generating "likely invariants" from program traces has been implemented in a system called Daikon [16]. Note that the quality of the invariants generated by Daikon is strongly dependent on the quality of the unit tests used in generating the program traces. It is unclear whether or not this kind of approach will scale-up. The issue of assertion generation has also been addressed to a limited extent within Houdini [17], a tool based upon predicate abstraction for generating annotations within the context of ESC/Java. More generally, as noted earlier, much work has already be done on the problem of generating invariants. As the sophistication of assertions increases, so does the number of auxiliary properties (lemmas) that need to be generated and proved. This has implications for the deductive capabilities of the verifying compiler, as will be discussed below.

4.2 Deductive issues

Generating verification proofs is hard. Proof should been seen as the final step in building confidence in a program and only attempted after conventional, light-weight, static analysis has been used to eliminate as many defects as possible. Moreover, experience has shown that static analysis should be applied routinely throughout program construction, and not left until unit testing [29]. Without this pro-active approach to static analysis, I believe, the challenge of the verifying compiler may prove to be too difficult. This raises questions about whether the verifying compiler should be developed with particular light-weight static analysis techniques in mind. And whether these techniques should be integrated within the verifying compiler.

Theorem proving techniques and strategies have advanced significantly since the early days of program reasoning. However, theorem provers are still the tool of the specialist. Talk, for instance, to any experienced ACL2 user and they will explain the failure-driven process that they engage with during a verification effort. An interactive process that involves the user in generalising conjectures, providing additional lemmas and identifying non-theorems. Programmers do not understand failed proof attempts. Where the limits of current deductive techniques are reached, the verifying compiler will have to provide meaningful feedback, *i.e.* an explanation that assists any follow-up debugging activity and which is expressed in programming terms, and not in terms of deductive failures. To achieve this, I believe a more integrated view of program reasoning is called for: one in which knowledge of the program and its specification are used to provide heuristic guidance to the theorem prover. Traditionally, such knowledge is distilled-out via the process of verification condition generation. This is not a new observation. The potential benefits of having a closer relationship between heuristic guidance and the theorem prover were anticipated early on [66].

As noted above, early experience in program verification also raised the importance of having an effective integration of decision procedures. This directly effected the design of PVS and ESC/Java, and has had an impact on the development of HOL [25]. Decision procedures will have a significant role to play within the verifying compiler.

Finally, the evolution of dependable systems gives rise to an evolving verification task. Consideration therefore needs to be given to how the verifying compiler can support this task, *i.e.* proof reuse and transformation.

5 A proof planning perspective

Proof automation and support for debugging will have a significant impact on the acceptability of the verifying compiler within the wider community. Here I outline the extent to which the theorem proving technique known as *proof planning* [5] addresses these issues. Proof planning uses high-level proof outlines, known as *proof plans*, to guide proof construction. Proof planning extends the tactic-based theorem proving paradigm [26], *i.e.* while tactics guarantee soundness, proof plans guide search. Decoupling these aspects of the theorem proving task allows for a schematic style of proof construction. Access to an explicit proof plan coupled with this schematic style of proof has been exploited with significant effect through the *proof critics* mechanism [32, 34]. Proof critics support

failure-driven proof, *i.e.* the automatic analysis and patching of failed proofs. The critics mechanism has been successfully used in patching proofs that require conjecture generalisation, lemma discovery, induction rule revision and case analyses [33, 34, 35]. With regard to the verifying compiler proposal, I believe there are a few key areas where proof planning has a valuable role to play:

Failure-driven proof: proof critics have also been applied to the problem of loop invariant discovery. Many of the informal methods for developing loop invariants presented within the structured programming literature [10, 27, 42] can be represented in terms of critic-based proof patching [38, 63, 39]. More recently we have extended these ideas within the context of SPARK. In particular we are integrating proof planning with program analysis techniques, thus making progress towards the more integrated view of program reasoning mentioned above [13, 14, 36].

Programmer-oriented feedback: the hierarchical nature of proof plans makes them suitable for communicating high-level proofs [49], while proof critics provide an opportunity to explain the choices that arise when a proof fails [37]. Integrating proof plans with program and specification knowledge, could provide a basis for explaining proof failures in terms familiar to programmers.

Decision procedures: proof planning has been used to develop a general framework for combining and augmenting decision procedures [41]. Proof planning therefore provides a basis for integrating theorem proving and decision procedures.

6 Conclusion

By way of surveying the history of verification systems and current trends, I have set out what I feel are some of the issues that will have to be addressed in building the verifying compiler. I believe that broadening the definition of the verifying compiler to include light-weight static analysis will be crucial. An integrated view of program reasoning is also called for in which both the program and its specification provide guidance for proof discovery. I believe proof planning has a contribution to make in achieving this integrated view. Building the verifying compiler will be challenging, however, the time is ripe to attempt this challenge!

References

[1] T. Ball and S.K. Rajamani. The SLAM project: Debugging system software via static analysis. In *Conference Record of POPL'02: The 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–3, Portland, Oregon, 2002.

[2] J. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley, 2003.

[3] BEACON. *Applied Dynamics International*. `http://www.adi.com/beacpg1.htm`.

[4] G. Brat and W. Visser. Combining static analysis and model checking for software analysis. In *Proceedings of the 16th IEEE Conference on Automated Software Engineering (ASE 2001)*, San Diego, CA, 2001. IEEE.

[5] A. Bundy. The use of explicit plans to guide inductive proofs. In R. Lusk and R. Overbeek, editors, *9th International Conference on Automated Deduction*, pages 111–120. Springer-Verlag, 1988. Longer version available from Edinburgh as DAI Research Paper No. 349.

[6] M. Caplain. Finding invariant assertions for proving programs. In *Proceedings of the International Conference on Reliable Software*, Los Angeles, CA, 1975.

[7] R. Chapman and P. Amey. Industrial strength exception freedom. In *Proceedings of ACM SigAda*. 2002.

[8] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, R.S. Laubach, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *22nd International Conference on Software Engineering*, pages 439–448, Limerick, Ireland, June 2000. IEEE Computer Society.

[9] L.P. Deutsch. *An Interative Program Verifier*. PhD thesis, University of California, Berkeley, 1973.

[10] E. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[11] EASY5. *Boeing*. `http://www.boeing.com/assocproducts/easy5/`.

[12] User guide for the EHDM specification language and verification system, version 6.1. Technical report, SRI Computer Science Laboratory, 1993.

[13] B.J. Ellis and A. Ireland. Automation for exception freedom proofs. In *Proceedings of the* 18^{th} *IEEE International Conference on Automated Software Engineering*, pages 343–346. IEEE Computer Society, 2003. Available from School of Mathematical and Computer Sciences, Heriot-Watt University, Technical Report HW-MACS-TR-0010.

[14] B.J. Ellis and A. Ireland. An integration of program analysis and automated theorem proving. Technical Report HW-MACS-TR-0014, School of Mathematical and Computer Sciences, Heriot-Watt University, 2004. Also to appear in the Proceedings of the 4th International Conference on Integrated Formal Methods 2004 (IFM-04).

[15] D. Elspas, M.W. Green, K.N. Levitt, and R.J. Waldinger. Research in interactive program-proving techniques. In *SRI, Menlo Park, CA*. 1972.

[16] M.D. Ernst, J. Cockrell, W.G. Griswold, and D. Notkin. Dynamically discovering likely program invariant to support program evolution. *IEEE Trans. on Software Engineering*, 27(2):1–25, February 2001.

[17] C. Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for ESC/Java. In *Proceedings of FME 2001*. LNCS 2021, Springer-Verlag, 2001.

[18] C. Flanagan, K. Rustan M. Leino, M. Lillibridge, G. Nelson, J. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of PLDI*, 2002.

[19] R. W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematics 19*, pages 19–32. American Mathematical Society, 1967.

[20] S.M. German. Automating proof of the absence of common runtime errors. In *Proceedings of 5th ACM Conference on Principles of Programming Languages*. 1978.

[21] S.M. German and B. Wegbreit. A synthesizer of inductive assertions. *IEEE Trans. on Software Engineering*, SE-1(1):68–75, 1975.

[22] H.H. Goldstine and J. von Neumann. Planning and coding of problems for an electronic computing instrument. In A.H. Taub, editor, *J. von Neumann: Collected Works*, pages 80–151. Pergamon Press, 1963. Originally, part II, vol. 1 of a report of the U.S. Ordinance Department 1947.

[23] D. I. Good. Mechanical proofs about computer programs. In C. A.R. Hoare and J. C. Shepherdson, editors, *Mathematical Logic and Programming Languages*, chapter 3, pages 55–75. Prentice-Hall, 1985.

[24] D.I. Good, R.L. London, and W.W. Bledsoe. An interactive program verification system. *IEEE Trans. on Software Engineering*, SE-1(1):59–67, 1975.

[25] M. Gordon and K.F. Larsen. Combining the Hol98 proof assistant with the BuDDy BDD package. Technical Report UCAM-CL-TR-481, University of Cambridge, Computer Laboratory, 15 JJ Thomson Avenue, Cambridge CB3 0FD, United Kingdom, phone +44 1223 763500, December 1999.

[26] M. J. Gordon, A. J. Milner, and C. P. Wadsworth. *Edinburgh LCF - A mechanised logic of computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[27] D. Gries. *The Science of Programming*. Springer-Verlag, New York, 1981.

[28] D. Guaspari, C. Marceau, and W. Polak. Formal verification of Ada programs. *IEEE Trans. on Software Engineering*, 16(9):1058–1075, September 1990.

[29] A. Hall and R. Chapman. Correctness by construction: Developing a commercial secure system. *IEEE Software*, 19(2), 2002.

[30] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–583, 1969.

[31] G. J. Holzmann. *The Spin Model Checker, Primer and Reference Manual*. Addison-Wesley, Reading, Massachusetts, 2003.

[32] A. Ireland. The Use of Planning Critics in Mechanizing Inductive Proofs. In A. Voronkov, editor, *International Conference on Logic Programming and Automated Reasoning – LPAR 92, St. Petersburg*, Lecture Notes in Artificial Intelligence No. 624, pages 178–189. Springer-Verlag, 1992. Also available from Edinburgh as DAI Research Paper 592.

[33] A. Ireland and A. Bundy. Extensions to a Generalization Critic for Inductive Proof. In M. A. McRobbie and J. K. Slaney, editors, *13th International Conference on Automated Deduction*, pages 47–61. Springer-Verlag, 1996. Springer Lecture Notes in Artificial Intelligence No. 1104. Also available from Edinburgh as DAI Research Paper 786.

[34] A. Ireland and A. Bundy. Productive use of failure in inductive proof. *Journal of Automated Reasoning*, 16(1–2):79–111, 1996. Also available as DAI Research Paper No 716, Dept. of Artificial Intelligence, Edinburgh.

[35] A. Ireland and A. Bundy. Automatic Verification of Functions with Accumulating Parameters. *Journal of Functional Programming: Special Issue on Theorem Proving & Functional Programming*, 9(2):225–245, March 1999. A longer version is available from Dept. of Computing and Electrical Engineering, Heriot-Watt University, Research Memo RM/97/11.

[36] A. Ireland, B.J. Ellis, and T. Ingulfsen. Invariant patterns for program reasoning. Technical Report HW-MACS-TR-0011, School of Mathematical and Computer Sciences, Heriot-Watt University, 2004. Also to appear in the Proceedings of the Mexican International Conference on Artificial Intelligence 2004 (MICAI-04).

[37] A. Ireland, M. Jackson, and G. Reid. Interactive Proof Critics. *Formal Aspects of Computing: The International Journal of Formal Methods*, 11(3):302–325, 1999. A longer version is available from Dept. of Computing and Electrical Engineering, Heriot-Watt University, Research Memo RM/98/15.

[38] A. Ireland and J. Stark. On the Automatic Discovery of Loop Invariants. In *Proceedings of the Fourth NASA Langley Formal Methods Workshop – NASA Conference Publication 3356*, 1997. Also available from Dept. of Computing and Electrical Engineering, Heriot-Watt University, Research Memo RM/97/1.

[39] A. Ireland and J. Stark. Proof planning for strategy development. *Annals of Mathematics and Artificial Intelligence*, 29(1-4):65–97, February 2001. An earlier version is available as Research Memo RM/00/3, Dept. of Computing and Electrical Engineering, Heriot-Watt University.

[40] D. Jackson and M. Vaziri. Finding bugs with a constraint solver. In *Proceedings of ISSTA'00*, Portland, Oregon, 2000.

[41] P. Janičić and A. Bundy. A general setting for flexibly combining and augmenting decision procedures. *Journal of Automated Reasoning*, 28(3):257–305, April 2002.

[42] A. Kaldewaij. *Programming: The Derivation of Algorithms*. Prentice Hall, 1990.

[43] S.M. Katz and Z. Manna. A heuristic approach to program verification. In *Proceedings of IJCAI-73*. International Joint Conference on Artificial Intelligence, 1973.

[44] S.M. Katz and Z. Manna. Logical analysis of programs. *Communications of the ACM*, 19(4):188–206, 1976.

[45] M. Kaufmann and J. Moore. An Industrial Strength Theorem Prover for a Logic Based on Common Lisp. *IEEE Transactions on Software Engineering*, 23(4):203–213, 1997.

[46] J. King. *A Program Verifier*. PhD thesis, Carnegie-Mellon University, 1969.

[47] S. King, J. Hammond, R. Chapman, and A. Pryor. Is proof more cost effective than testing? *IEEE Trans. on SE*, 26(8), 2000.

[48] K.N. Levitt, P.G. Neumann, and L. Robinson. Computer science and technology: The SRI heirarchical development methodology (hdm) and its application to the development of secure software, 1980.

[49] H. Lowe and D. Duncan. XBarnacle: Making theorem provers more accessible. In William McCune, editor, *14th International Conference on Automated Deduction*, pages 404–408. Springer-Verlag, 1997.

[50] D. Luckham, F. von Henke, B. Krieg-Brückner, and O. Owe. *ANNA: A Language for Annotating Ada Programs*. Springer, 1987. Lecture Notes in Computer Science, Volume 260.

[51] D. C. Luckham, S.M. German, F.W. v.Henke, R.A. Karp, P.W. Milne, D.C. Oppen, W. Polak, and W.L. Scherlis. Stanford pascal verifier user manual. Research Report CS-TR-79-731, Stanford University, Department of Computer Science, 1979.

[52] MATRIXx. *WindRiver*. `http://www.isi.com/products/matrixx/autocode/`.

[53] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell Massachusetts, 1993.

[54] D.R. Musser. Abstract data type specification in the AFFIRM system. *IEEE Trans. on Software Engineering*, SE-6(1):24–32, January 1980.

[55] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, October 1979.

[56] M. Norrish. *Formalising C in HOL*. PhD thesis, Computer Lab., University of Cambridge, 1998.

[57] C. O'Halloran. Acceptance based assurance. In *Proceedings of the 16th IEEE Conference on Automated Software Engineering (ASE 2001)*, San Diego, CA, 2001. IEEE.

[58] P. O'Hearn, J. Reynolds, and Y. Hongseok. Local reasoning about programs that alter data structures. In *Proceedings of CSL'01*, volume 2142 of *Lecture Notes in Computer Science*, Paris, 2001.

[59] S. Owre, N. Shankar, and J. Rushby. Pvs: A prototype verification system. In D. Kapur, editor, *CADE11*, number 607 in LNAI, 1992.

[60] A. Pnueli, E. Shtrichman, and M. Siegel. Translation validation for synchronous languages. In *ICALP: Annual International Colloquium on Automata, Languages and Programming*, 1998.

[61] R. E. Shostak. Deciding combinations of theories. *Journal of the ACM*, 31(1):1–12, January 1984.

[62] Simulink. *Mathworks*. `http://www.mathsworks.com/products/rtw/`.

[63] J. Stark and A. Ireland. Invariant discovery via failed proof attempts. In P. Flener, editor, *Logic-Based Program Synthesis and Transformation*, LNCS 1559, pages 271–288. Springer-Verlag, 1998. An earlier version is available from the Dept. of Computing and Electrical Engineering, Heriot-Watt University, Research Memo RM/98/2.

[64] J T Webb and D Mannering. MALPAS - verification of a safety critical system. *SARSS'87: Achieving Safety and Reliability with Computer Systems*, page 44, 1987.

[65] B. Wegbreit. Heuristic methods for mechanically deriving inductive assertions. In *Proceedings of IJCAI-73*. International Joint Conference on Artificial Intelligence, 1973.

[66] B. Wegbreit. The synthesis of loop predicates. *Comm. ACM*, 17(2):102–122, 1974.