Tool Integration for Reasoned Programming*

Andrew Ireland

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

Abstract. We argue for the importance of tool integration in achieving the Program Verifier Grand Challenge. In particular, we argue for what we call *strong integration*, *i.e.* a co-operative style of interaction between tools. We propose the use of an existing planning technique, called *proof planning*, as a possible basis for achieving strong integration.

1 Introduction

The renewed interest in the mechanical verification of software, we believe, can be attributed in part to the following three factors:

- A focus on property based verification, rather than full functional verification.
- Progress in terms of mechanizing abstractions.
- Greater integration of tools.

Below we highlight some software verification projects in which these factors played a key role:

- SLAM [1] provides an integrated toolkit for checking safety properties of software interfaces written in C. SLAM has been applied very successfully to the validation of device driver software. Predicate abstraction and model checking are used to identify potential defects. Using a theorem prover, the potential defects are then refined to identify true defects.
- ESC/Java [12] is a tool for identifying defects in Java programs. Using a theorem prover, ESC/Java can verify that a program is free of run-time exceptions. In general, annotations are required in order to support the theorem proving. In order to address this annotation burden, ESC/Java has been integrated with the Houdini [11] *annotation assistant*. Houdini is based upon predicate abstraction, and uses refutations to refine candidate annotations.
- Caveat [3] is a static analysis tool for software written in C, and was used during the development of the flight-control software for the Airbus A380. Caveat includes a theorem prover that supports the verification of annotated C programs. A tool called Cristal supports the automatic generation of annotations (preconditions) for run-time exception freedom proofs. Currently, abstract interpretation [26] is being explored as a basis for generating loop invariants [25].
- NuSPADE[1] [9, 10, 20] builds upon the SPARK approach to high integrity software development [2]. The SPARK approach has been used extensively on safety [22] and security [15] critical applications. The NuSPADE project developed an integrated approach to program reasoning, based upon the use of proof-failure analysis to constrain the generation of program annotations. NuSPADE focused in particular on automation for run-time exception freedom proofs.

* The work discussed was supported in part by EPSRC grants GR/R24081 and GR/S01771. We are grateful for feedback on this position paper from Alan Bundy. Thanks also goes to Praxis High Integrity Systems Ltd, in particular Peter Amey and Rod Chapman for their support.

[1] More details can be found at `http://www.macs.hw.ac.uk/nuspade`

The above list is by no means complete. The aim is simply to highlight the role of property-based verification, mechanized abstract and tool integration within current software verification projects. The remainder of this position paper focuses on the importance of tool integration for software verification.

2 Tool integration

The importance of tool integration for software verification is not a new observation. For instance, the potential benefits of having a close relationship between heuristic guidance, *i.e.* annotation generation, and theorem proving were anticipated by Wegbreit in his early work on program verification [31]. Achieving a "close relationship", what we will refer to as *strong integration*, requires a co-operative style of interaction between tools. Note that strong integration is closely related to the notion of *tightly coupled integration* presented in [8]. The use of counterexamples in guiding the search for program annotations is an example of strong integration. As an aside, the importance of counterexamples within the context of software verification is discussed in more detail in [30]. This is in contrast to a black box style of integration, or *weak integration*, where interaction between tools is minimal, *e.g.* success and failure.

In terms of automated reasoning, the benefits of strong integration are illustrated in [4] where Boyer and Moore report on the experimental integration of their theorem prover with a decision procedure for linear arithmetic. They found that the decision procedure was directly applicable to very few subgoals generated by the theorem prover – so weak integration gave poor performance. In contrast, strong integration, *i.e.* allowing the theorem prover and decision procedure to interact co-operatively, gave significant performance improvements. However, the customization associated with such strong integration is costly. Boyer and Moore reported that implementing strong integration was time-consuming, involving extensive and complex changes to both the theorem prover and decision procedure. An in-depth discussion of the trade-offs that need to be considered when addressing the challenge of tool integration can be found in [8].

If one accepts strong integration as an important factor in addressing the task of software verification, then alleviating the costs associated with strong integration is an important milestone on the road to meeting the Program Verifier Challenge. We believe that approaches that support the kind of "customization" outlined above will play a vital role in alleviating such costs. We propose planning, and in particular *proof planning* [5], as a possible approach to achieving the level of customization that is required in order reduce the cost of strong integration.

Proof planning is a computer-based technique for automating the search for proofs. At the core of the technique are high-level proof outlines, known as *proof plans*. Proof planning builds upon tactic-based reasoning [14]. Starting with a set of general purpose tactics, plan formation techniques are used to construct a customized tactic for a given conjecture. A key feature of proof planning is that it separates proof search from proof checking. This gives greater flexibility in the strategies that can be used in guiding proof search as compared to conventional proof development environments. An example of this greater flexibility is the *proof critics* mechanism [16, 18] that supports the automatic analysis and patching of proof planning failures. Proof critics have been very successful in automating the generation of auxiliary lemmas, conjecture generalizations and loop invariants [17–19, 29, 21].

Inspired by [4], the value of proof planning as a basis for strong integration was first observed in [6], where part of a decision procedure was rationally reconstructed as a proof plan. The modularity imposed by the proof plan enabled flexibility in

the application of the decision procedure, *e.g.* auxiliary information such as lemmas, could be easily incorporated. In terms of tool integration, the value of proof planning as a basis for a co-operative style reasoning has been demonstrated through the Clam-HOL [28] and NuSPADE projects, the details of which are outlined below.

In the case of Clam-HOL, the Clam proof planner [7] was integrated with the Cambridge HOL interactive theorem prover [13]. The Boyer and Moore integration example, highlighted above, was re-implemented within the Clam-HOL framework with positives results [27].

Within the NuSPADE project, proof plans were used to increase the level of proof automation available via the SPARK toolset. Part of this effort involved the development of new proof plans, as well as the reuse of existing proof plans, *i.e.* proof plans developed for mathematical induction. The NuSPADE project also broadened the role of proof plans, *i.e.* proof patching was extended to incorporate light-weight program analysis. That is, common patterns of proof-failure were identified with constraints on missing properties. These constraints were used by our program analyzer to guide the introduction of auxiliary program annotations, *e.g.* loop invariants. It should be noted that the program analyzer also initiated interactions with the proof planner, *i.e.* the program analyzer called upon the proof planner to discharge simple equational reasoning goals. In terms of automation for run-time exception freedom proofs, NuSPADE was evaluated on a number of industrial applications, including SHOLIS [22], the first system developed to meet the UK Ministry of Defence Interim Defence Standards 00-55 [24] and 00-56 [23]. Our techniques are aimed at verification conditions that arise in loop-based code. While industrial strength critical software systems are engineered to minimize the number and complexity of loops, we found 80% of the loops that we encountered were provable using our techniques. That is, our program analysis, guided by proof-failure analysis, automatically generated auxiliary program annotations that enabled subsequent proof planning and proof checking attempts to succeed.

3 Conclusion

Tool integration is prevalent within current software verification projects. We have argued for the value of strong integration, *i.e.* a co-operative style of tool interaction, within the context of software verification. To achieve strong integration, we have proposed the use of proof planning, an approach which has a track-record in the development of reasoning systems which embody a co-operative style of interaction. We believe that strong integration will accelerate the development and sharing of tools and techniques on the road to achieving the Program Verifier Grand 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. P. Baudin, A. Pacalet, J. Raguideau, D. Schoen, and N. Williams. Caveat: A tool for software validation. In *International Conference on Dependable Systems and Networks (DSN02).* IEEE Computer Society, 2002.
4. R. S. Boyer and J S. Moore. Integrating decision procedures into heuristic theorem provers: A case study of linear arithmetic. In J. E. Hayes, J. Richards, and D. Michie, editors, *Machine Intelligence 11*, pages 83–124, 1988.

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. A. Bundy. The use of proof plans for normalization. In R. S. Boyer, editor, *Essays in Honor of Woody Bledsoe*, pages 149–166. Kluwer, 1991. Also available from Edinburgh as DAI Research Paper No. 513.
7. A. Bundy, F. van Harmelen, C. Horn, and A. Smaill. The Oyster-Clam system. In M. E. Stickel, editor, *10th International Conference on Automated Deduction*, pages 647–648. Springer-Verlag, 1990. Lecture Notes in Artificial Intelligence No. 449. Also available from Edinburgh as DAI Research Paper 507.
8. L. de Moura, S. Owre, H. Rueb, J. Rushby, and N. Shankar. Integrating verification components: The interface is the message. 2005. See `http://www.csl.sri.com/users/shankar/shankar-drafts.html`.
9. 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. Also available from the School of Mathematical and Computer Sciences, Heriot-Watt University, as Technical Report HW-MACS-TR-0010.
10. B.J. Ellis and A. Ireland. An integration of program analysis and automated theorem proving. In E.A. Boiten, J. Derrick, and G. Smith, editors, *Proceedings of 4th International Conference on Integrated Formal Methods (IFM-04)*, volume 2999 of *Lecture Notes in Computer Science*, pages 67–86. Springer Verlag, 2004. Also available from the School of Mathematical and Computer Sciences, Heriot-Watt University, as Technical Report HW-MACS-TR-0014.
11. C. Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for ESC/Java. In *Proceedings of FME 2001*. LNCS 2021, Springer-Verlag, 2001.
12. 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.
13. M. J. Gordon. HOL: A proof generating system for higher-order logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*. Kluwer, 1988.
14. 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.
15. A. Hall and R. Chapman. Correctness by construction: Developing a commercial secure system. *IEEE Software*, 19(2), 2002.
16. 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.
17. 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.
18. 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.
19. 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.
20. A. Ireland, B.J. Ellis, A. Cook, R. Chapman, and J. Barnes. An integrated approach to high integrity software verification. *Journal of Automated Reasoning: Special Issue on Empirically Successful Automated Reasoning*, 36(4):379–410, 2006.

21. 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.
22. S. King, J. Hammond, R. Chapman, and A. Pryor. Is proof more cost effective than testing? *IEEE Trans. on SE*, 26(8), 2000.
23. MoD. Hazard analysis and safety classification of the computer and programmable electronic system elements of defence equipment. Interim Defence Standard 00-56, Issue 1, Ministry of Defence, Directorate of Standardization, Kentigern House, 65 Brown Street, Glasgow G2 8EX, UK, April 1991.
24. MoD. The procurement of safety critical software in defence equipment (part 1: Requirements, part 2: Guidance). Interim Defence Standard 00-55, Issue 1, Ministry of Defence, Directorate of Standardization, Kentigern House, 65 Brown Street, Glasgow G2 8EX, UK, April 1991.
25. T. Nguyen and A. Ourghanlian. Dependability assessment of safety-critical system software by static analysis methods. In *International Conference on Dependable Systems and Networks (DSN03)*. IEEE Computer Society, 2003.
26. PolySpace-Technologies. http://www.polyspace.com/.
27. K. Slind and R. Boulton. Iterative dialogues and automated proof. In *Proceedings of the Second International Workshop on Frontiers of Combining Systems (FroCoS'98)*, Amsterdam, The Netherlands, October 1998.
28. K. Slind, M. Gordon, R. Boulton, and A. Bundy. System description: An interface between CLAM and HOL. In C. Kirchner and H. Kirchner, editors, *15th International Conference on Automated Deduction*, volume 1421 of *Lecture Notes in Artificial Intelligence*, pages 134–138, Lindau, Germany, July 1998. Springer. Earlier version available from Edinburgh as DAI Research Paper 885.
29. 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.
30. G. Steel. The importance of non-theorems and counterexamples in program verification. 2005. Submitted to the IFIP Working Conference on Verified Software: Theories, Tools, Experiments.
31. B. Wegbreit. The synthesis of loop predicates. *Comm. ACM*, 17(2):102–122, 1974.