Are We There Yet? 20 Years of Industrial Theorem Proving with SPARK

Roderick Chapman[1] and Florian Schanda[2]

[1] Altran UK Limited, 22 St Lawrence Street, Bath BA1 1AN (United Kingdom)
`roderick.chapman@gmail.com`
[2] Altran UK Limited, 22 St Lawrence Street, Bath BA1 1AN (United Kingdom)
`florian.schanda@altran.com`

Abstract. This paper presents a retrospective of our experiences with applying theorem proving to the verification of SPARK programs, both in terms of projects and the technical evolution of the language and tools over the years.

Keywords Formal Methods, Program Verification, Theorem Proving, SPARK

1 Introduction

This paper reflects on our experience with proving properties of programs written in SPARK[2] - a programming language and verification toolset that we have designed, maintained, sold, and used with some success for nearly 20 years.

2 Projects and Technologies

The following sections present a retrospective on the use of theorem proving in SPARK, roughly alternating between technical developments in the language and tools and the experiences of various projects, coming from both our own experience and that of a selection of external SPARK users in industry.

2.1 Early Days - 1987ish

It all started in about 1987. Our predecessors at the University of Southampton and then PVL had designed and implemented a Hoare-logic based verification system for a subset of Pascal[9]. Their next goal seemed almost absurdly bold - to design a programming language and verification system that would offer sound verification for non-trivial programs, but be scalable and rich enough to write "real world" embedded critical systems. The base language chosen was Ada83, and through judicious subsetting, semantic strengthening, and the addition of contracts, SPARK (the "SPADE Ada Ratiocinative Kernel" - we kid you not) was born, with a first language definition appearing in March 1987.

The highest priority design goal was the provision of soundness in all forms of verification. This led to a need for a completely unambiguous dynamic semantics, so that the results of verification would be reliable for all compilers and target machines, and so that all valid SPARK programs would be valid Ada programs with the same meaning and also, at a stroke, removing the need for us to produce a "special" compiler for SPARK.

The toolset consisted of three main tools

– The Examiner. This consists of a standard compiler-like "front-end", followed by various analyses that check the language subset, aliasing rules, data- and information-flow analysis, then (finally) generation of verification conditions (VCs) in a language called FDL - a typed first-order logic that has a relatively simple mapping from SPARK. The VCs generated include those for partial correctness with respect to preconditions, postconditions and loop-invariants, but also (lots of) VCs for "type safety" such as the freedom from buffer overflow, arithmetic overflow, division by zero and so on.
– The Checker. This is an interactive proof assistant for FDL. It emerged from of one of the team's PhD research[19]. Written in PROLOG.
– The Simplifier[3]. This is an heuristic, pattern-matching theorem prover, based on the same core inference engine as the Checker. It started out as a way of literally "Simplifying" VCs (a bit) before the real fun could start with the Checker, but as we will see, it grew substantially in scope and power over the years.

2.2 SHOLIS Project

Our first attempt at serious proof of a non-trivial system came with the SHOLIS project in 1995, by which time PVL and the SPARK technology had been acquired by Praxis (now Altran UK). This was the first effort to meet the requirements of the (then) rather onerous Interim version of the UK's Def-Stan 00-55 for critical software at Integrity Level "SIL4".

SHOLIS is a system that assists Naval crew with the safe operation of helicopters at sea, advising on safety limits (for example incident wind vector on the flight deck, and ship's roll and pitch) for particular operations such as landing, in-air refuelling, crew transfer and so on.

The SHOLIS software[16] comprised about 27 kloc (logical) of SPARK code, 54 kloc of information-flow contracts, and 29 kloc of proof contracts, plus some tiny fragments of assembly language to support CPU start-up and system boot. There was no operating system and no COTS libraries of any kind - a significant simplification at this level of integrity, and a programming model that SPARK was explicitly designed to support.

The SHOLIS code generated nearly 9000 VCs, of which 3100 were for functional and safety properties, and 5900 for type safety. Of the 9000 total, 6800

[3] Not to be confused with Greg Nelson's better-known Simplify prover.

(75.5 %) were proven automatically by the Simplifier, with the remaining 2200 being "finished off" using the interactive Checker.

The experience was painful. Computing resource was scarce; one UNIX server (shared by the whole company) was used for all the proof work. Simplification times for a single subprogram were measured in hours or days.

A key output was the identification of the need to support state abstraction and refinement in SPARK proofs. This mechanism is used in SPARK to control the volume of states that are visible, and hence has a direct impact on the complexity and size of contracts and the "postcondition explosion" problem. This was implemented (too late for SHOLIS), but had a major impact on later projects.

2.3 C130J Project

The Lockheed-Martin C130J is the most recent generation of the enormously successful "Hercules" military transport aircraft. The Mission Computer application software is written in SPARK, and was subject to a large verification effort in the UK as part of the acquisition of the aircraft by the UK RAF.

Verification consisted of full-blown SPARK analysis, including verification of partial correctness for most critical functions with respect to the system's functional specification, which was expressed using the "Parnas Tables"[20] notation.

Unusually, the "proof" component of the work was performed in the UK in the late 1990s *after* the formal testing (to meet the objectives of DO-178B Level A) of the Mission Computer.

Results from the development phase of the project are best reported in [18] while the results of the later proof work (and comparisons of the SPARK code with other systems and programming languages) can be found in [13].

2.4 Improving the VCG and Simplifier

Both the SHOLIS and C130J projects led to a serious analysis of how we could improve the completeness of the Simplifier. Analysis identified a number of key areas where improvement was sorely needed:

– Tactics for unwrapping and instantiation of universally quantified conclusions, especially those that commonly arise from arrays in SPARK.
– Modular (aka "unsigned") arithmetic - very common in low-level device driver code, ring buffers, cryptographic algorithms and so on.
– Tracking the worst-case ranges of integer expressions.

These were implemented in 2002. We also spent effort on improving the VC Generator (VCG) itself - it turns out it's only too easy for the VCG to produce a VC that omits some vital hypothesis so that *no* prover could prove it, no matter how clever. Ever more detailed and precise modelling of the language semantics inside the VCG continues to this day.

Finally, we noted one other factor that critically impacted the usefulness of the proof system - the "proof friendliness" of the code under analysis. This seemed to correlate with common software engineering guidance - simplicity, low information-flow coupling, and proper use of abstraction to control the name- and state-space of any one subprogram. In short - we were learning how to write provable programs, so we started to set a goal for projects to "hit" a particular level of automatic proof (e.g. 95 % of VCs discharge automatically), making the proof a design-level challenge rather than a retrospective slog.

2.5 Tokeneer Project

Tokeneer is an NSA-funded demonstrator of high-security software engineering. We were given a clean-sheet to work from, so the project deployed various forms of formal methods, including a system specification and security properties in Z, and implementation in SPARK[1].

This was the first time we had attempted to prove non-trivial security properties of a software system with SPARK. Owing to the budget, the system was small (only about 10 kloc logical, producing 2623 VCs), but critically, 2513 of those were proven automatically (95.8 %), with only 43 left to the Checker and 67 discharged by review (i.e. we looked at them really hard).

Unusually (and some years later, in 2008) the NSA granted a licence that effectively allowed a fully "open source" release of the entire Tokeneer project archive. It remains the focus of various research efforts[28].

2.6 Speeding Up and Going FLOSS

With the emergence of cheap multi-core CPUs, we implemented an obvious improvement to the Simplifier in 2007 - the ability to run the prover on several VCs at once in parallel. It turns out the VCG generates *lots* of small, simple VCs which are completely independent of one another, so are ripe for parallelization. This almost wasn't a conscious design goal in 1990ish, but came as a pleasantly surprising benefit.

The results are dramatic. On a modern (2013-era) quad-core machine, the entire SHOLIS software can be re-proven from scratch in about 11 minutes, with only 440 undischarged VCs (compare with weeks and 2200 undischarged VCs in the original project).

Secondly, in 2009 we took the dramatic step, in partnership with AdaCore, to "go open source", with the entire toolset moving to a FLOSS development model and GPL licence, as a way of promoting interest, teaching and research with the language.

2.7 User-defined rules

In response to customer demand, we implemented an approach for users to write and "insert" additional rules or lemmas into the Simplifier to "help" it with particularly tricky VCs, or in areas where its basic reasoning power proved insufficient.

This approach opens an obvious soundness worry (users can write nonsensical rules), but does offer an attractive middle-ground between the onerous use of the Checker and the rather relaxed idea of just “eyeballing” the undischarged VCs to see if they look OK. Additionally, a single well-written rule can be written once but used thousands of times by the Simplifier, so the effort to get the rules right should pay off. Finally, the Checker be used to verify the soundness of a user-defined rule from first principles if required.

2.8 iFACTS Project - Scaling up

Starting in 2006, the implementation of the NATS iFACTS system is the most ambitious SPARK project to date.

iFACTS augments the tools available to en-route air-traffic controllers in the UK. In particular, it supplies electronic flight-strip management, trajectory prediction, and medium-term conflict detection for the UK’s en-route airspace, giving controllers a substantially improved ability to plan ahead and predict conflicts in a sector[21].

The project has a formal functional specification (again expressed mostly in Z), and the majority of the code (about 250 kloc logical lines of code, as counted by GNATMetric) is implemented in SPARK. Figure 1 illustrates that the proportion of SPARK contracts is minor compared to the bulk of the executable code and comments (and whitespace).

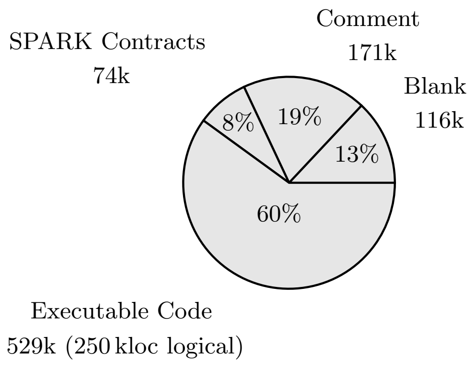

Fig. 1. Project size in physical lines of code (counted with `wc -l`).

Proof concentrates on type-safety, but not functional correctness, since the system has stringent requirements for reliability and availability - in short, the software must be proven “crash proof”.

The current operational build produces 152927 VCs, of which 151026 (98.76 %) are proven entirely automatically by the Simplifier alone. User-defined rules are used for another 1701 VCs, with only 200 proved "by review". This remains the highest "hit rate" for automatic proof with SPARK that we have ever encountered.

The proof is reconstructed daily (and overnight), and developers are under standing orders that all code changes *must* prove OK before code can be committed into the CM system. Developers who "break the proof" receive a terse notification the following morning.

2.9 Reaching out - SMT and Counterexamples

Licensing SPARK under the GPL has also made it much easier to collaborate with academia, and we have seen two useful improvements for SPARK 2005 as a result of this.

Alternative provers. The existing automatic theorem prover for SPARK was good (98.76 % on well written code), but due to its nature had difficulties discharging certain VCs. Paul Jackson from the University of Edinburgh has written Victor[15], a translator and prover driver to allow SMT solvers to be used with SPARK. As SMT solvers are fundamentally different from rewrite systems such as the Simplifier, they are able to easily discharge many of the VCs the Simplifier cannot deal with.

For some projects, such as SPARKSkein, using a modern SMT solver allowed 100 % of all VCs to be discharged automatically. Victor is now shipped with SPARK.

Counterexamples. The existing SPARK proof tools were all geared towards proving VCs. As a consequence it was difficult (and thus time-consuming, and thus expensive) to distinguish between true VCs that could not be proved due to prover limitations and false VCs due to specification or programming errors.

Riposte [24] is the counter-example generator for SPARK that was developed under a joint project with Martin Brain (University of Bath, now Oxford). This tool provides helpful counter-examples, and can even be used as a proof tool as it is sound, so the lack of a counter-example guarantees none exist (although the tool is incomplete, so sometimes a counter-example might be generated where none exists).

2.10 SPARKSkein Project - Fast, Formal, Nonlinear

In 2010, we implemented the Skein[25] hash algorithm in SPARK. The goal was to show that a "low-level" cryptographic algorithm like Skein could be implemented in SPARK, be subject to a complete proof of type safety, be readable and "obviously correct" with respect to the Skein mathematical specification, and also be as fast or faster than the reference implementation offered by the designers, which is implemented in C.

The proofs of type safety turned out to be quite tricky. Firstly, finding the correct loop invariants proved difficult, and this was compounded by the plethora of modular types and non-linear arithmetic in the VC structures. Of the 367 VCs, 23 required use of the Checker to complete the proof - not bad but these still required a substantial effort to complete. Full results from the project are reported in [7]. All sources, proofs and tests have been released under GPL and can be obtained here[26].

One final unexpected side-effect was the discovery of a subtle corner-case bug in the designers' C implementation (an arithmetic overflow, which leads to a loop iterating zero times, which leads to an undefined output).

2.11 CacheSimp - Speeding up even more

In an industrial context, verification time is not just a number, it has a significant qualitative effect on how verification tools are used and thus on project management. If verification takes 4 hours, then a developer has to organise their working day around this activity. If the same activity can be done in 30 minutes or less then this has a significant and positive impact on how the tools are used (and often mistakes are found much earlier).

Figure 2 shows that a single proof run of iFACTS takes around 3 hours (green line with squares) on a fast desktop computer, regardless of how big the actual change is (blue bars). We implemented a very simple caching system [6] in around 250 lines of code using memcached [12], where each invocation of the proof tools first checks if the result is already known. As the memcached server was run on a separate computer accessible to all developers, this system was both incremental and distributed, leading to an average 29-fold speedup (red line with circles) and a strong correlation to the size of the change.

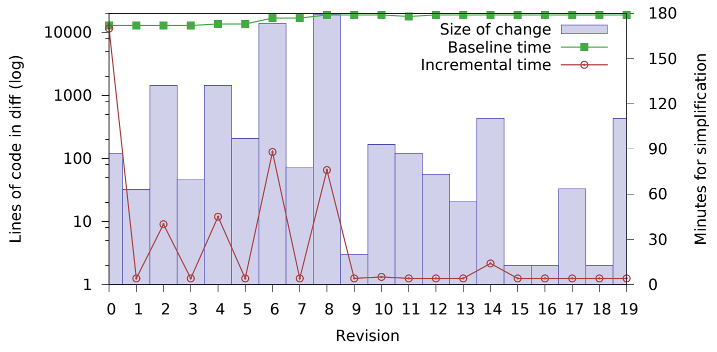

Fig. 2. Results showing the effects of using a simple caching system

2.12 Reaching out - Interactive Provers

One particular SPARK user, secunet, kept pushing the boundaries of what was reasonable to achieve using automated verification, and what was expressible using the simple first order logic constructs available in SPARK annotations. Stefan Berghofer (from secunet) implemented a plug-in for Isabelle/HOL that allows one to express much richer properties using SPARK proof functions [4] and complete the proof in Isabelle. They have kindly released their work under a free software license, including a fully verified big-number library which they have used to implement RSA. Given the recent OpenSSL "Heartbleed"[17] bug, this importance of this contribution should not be underestimated.

2.13 Muen Project

This project[27] span off from the work of secunet. Muen is a FLOSS separtion-kernel for the x86_64 architecture, but unusually, almost the entire kernel is written in SPARK - something we might have considered impossible some years ago. The kernel code is subject to an automated proof of type-safety. We look forward to further results from this group.

3 Future Trends

This final section reflects on two topics - the future of SPARK, and the role that theorem-proving evidence can play in the wider context of regulation and acceptance of critical software.

3.1 Technologies and Languages - SPARK 2014

Despite many positive experiences, SPARK and the use of the proof tools remain a challenge for many customers - the "adoption hurdle" is often perceived as too high. Secondly, it became clear to us that improving the Simplifier had become a game of diminishing returns, and it was time to move to more modern proof technologies. Finally, the arrival of Ada 2012 brought contracts into the mainstream Ada syntax, so it was time to "reboot" both the language design and underlying technologies.

Since 2012, we have been working with AdaCore to produce SPARK 2014. The language is based on Ada 2012, and uses its native "aspect" notation for all contracts. The language subset permitted is much larger, including variant records, generic units, dynamic types, and so on. The new toolset is based on the full GNAT Pro Ada front-end (part of the GCC family), a new information-flow analysis engine (based on analysis of program-dependence graphs[11, 14]), and a new proof system that uses the Why3 language and VCG[5] and modern SMT solvers such as Alt-Ergo[8] and CVC4[3]. The new tools also bring a significant improvement in the area of floating point verification: where the old tools used an unsound model using real numbers, the new tools employ a sound encoding;

current versions use an axiomatised rounding function. We feel that a transition to the upcoming SMTLIB floating-point standard will bring many benefits, in particular preliminary experiments (with the University of Oxford) using SMT solvers implementing ACDL [10] have shown promise.

SPARK 2014 also (perhaps unusually) takes the step of unifying the dynamic (i.e. run-time) and static (i.e. for proof) semantics of contracts, so that they can be proved, or tested at run-time (as in, say, Eiffel), or both - providing some interesting possibilities for mixing verification styles and/or mixing languages (i.e. programs partly written in SPARK, Ada, C, or anything else) in a single program.

Both the “Pro” (supported) and “GPL” (free, unsupported) versions of the SPARK 2014 toolset will be available before this conference.

3.2 Assurance and Acceptance for Critical Systems

Over the years, both customers and regulators have taken a variety of stances on the use of strong static analysis and theorem proving in critical software. Some regulators remain sceptical, perhaps owing to the novelty of the idea or the perceived unreliability (i.e. unsoundness) of common “bug finding” style static analysis tools.

The future looks bright, though, in the aerospace with the advent of DO-178C[22] and its formal methods supplement DO-333[23] which explicitly allows “formal methods” as a combination of an unambiguous language and analysis methods which can be shown to be sound. Additionally, DO-178C allows later verification activities (e.g. testing) to be reduced or eliminated if it can be argued that formal analytical approaches have met the required verification objective(s). This supports (we hope) a strong economic incentive for the adoption of more formal and static approaches.

We look forward to the day when software will be delivered with its proofs, which can be re-generated at will by the customer or regulator, perhaps even by diverse verification tools. That would surely move us towards a claim to being a true engineering discipline.

References

1. Barnes, J., Chapman, R., Johnson, R., Widmaier, J., Cooper, D., Everett, B.: Engineering the Tokeneer enclave protection software. In: 1st IEEE International Symposium on Secure Software Engineering (March 2006) (2006)
2. Barnes, J.: SPARK: The Proven Approach to High Integrity Software. Altran Praxis (2012)
3. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: Cvc4. In: Computer aided verification. pp. 171–177. Springer (2011)
4. Berghofer, S.: Verification of dependable software using SPARK and isabelle. In: SSV. pp. 15–31 (2011)

5. Bobot, F., Filliâtre, J.C., Marché, C., Paskevich, A.: Why3: Shepherd your herd of provers. In: Boogie 2011: First International Workshop on Intermediate Verification Languages. pp. 53–64. Wrocław, Poland (August 2011), http://proval.lri.fr/publications/boogie11final.pdf
6. Brain, M., Schanda, F.: A lightweight technique for distributed and incremental program verification. In: Verified Software: Theories, Tools, Experiments, pp. 114–129. Springer (2012)
7. Chapman, R., Botcazou, E., Wallenburg, A.: SPARKSkein: a formal and fast reference implementation of skein. In: Formal Methods, Foundations and Applications, pp. 16–27. Springer (2011)
8. Conchon, S., Contejean, E., Kanig, J.: Ergo: A theorem prover for polymorphic first-order logic modulo theories (2006), http://ergo.lri.fr/papers/ergo.ps
9. Cullyer, W., Goodenough, S., Wichmann, B.: The choice of computer languages for use in safety-critical systems. Software Engineering Journal 6(2), 51–58 (1991)
10. D'Silva, V., Haller, L., Kroening, D.: Abstract conflict driven learning. In: POPL. pp. 143–154 (2013)
11. Ferrante, J., Ottenstein, K.J., Warren, J.D.: The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst. 9(3), 319–349 (Jul 1987), http://doi.acm.org/10.1145/24039.24041
12. Fitzpatrick, B., et al.: memcached - a distributed memory object caching system. http://memcached.org (2003)
13. German, A.: Software static code analysis lessons learned. Crosstalk 16(11) (2003)
14. Horwitz, S., Reps, T., Binkley, D.: Interprocedural slicing using dependence graphs. In: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation. pp. 35–46. PLDI '88, ACM, New York, NY, USA (1988), http://doi.acm.org/10.1145/53990.53994
15. Jackson, P.B., Passmore, G.O.: Proving SPARK verification conditions with smt solvers. Tech. rep., Technical Report, University of Edinburgh (2009)
16. King, S., Hammond, J., Chapman, R., Pryor, A.: Is proof more cost-effective than testing? Software Engineering, IEEE Transactions on 26(8), 675–686 (2000)
17. Mehta, N.: Cve-2014-0160 (4 2014)
18. Middleton, P., Sutton, J.: Lean Software Strategies: Proven Techniques for Managers and Developers. Productivity Press (2005)
19. O'Neill, I.: Logic Programming Tools and Techniques for Imperative Program Verification. Ph.D. thesis, University of Southampton (1987)
20. Parnas, D.L., Madey, J., Iglewski, M.: Precise documentation of well-structured programs. IEEE Transactions on Software Engineering 20(12), 948–976 (1994)
21. Rolfe, M.: How technology is transforming air traffic management. http://nats.aero/blog/2013/07/how-technology-is-transforming-air-traffic-management
22. RTCA: DO-178C: Software considerations in airborne systems and equipment certification (2011)
23. RTCA: DO-333: Formal methods supplement to do-178c and do-278a (2011)
24. Schanda, F., Brain, M.: Using answer set programming in the development of verified software. In: ICLP (Technical Communications). pp. 72–85 (2012)
25. Schneier, B., Ferguson, N., Lucks, S., Whiting, D., Bellare, M., Kohno, T., Walker, J., Callas, J.: The skein hash function family. Submission to NIST(Round 3) (2010)
26. www.skein-hash.info
27. muen.codelabs.ch
28. Woodcock, J., Aydal, E.G., Chapman, R.: Reflections on the Work of CAR Hoare, chap. The Tokeneer Experiments, pp. 405–430. Springer (2010)