Industrial Experience with SPARK

Roderick Chapman
Praxis Critical Systems Limited,
20 Manvers Street,
Bath BA1 1PX, U.K.
+44 1225 466991
rod@praxis-cs.co.uk

1. ABSTRACT

This paper considers a number of large, real-world projects that are using SPARK—an annotated sublanguage of Ada that is appropriate for the development of high-integrity systems. Three projects are considered in some detail where SPARK has made a contribution to meeting the most stringent software engineering standards. The projects are the Ship/Helicopter Operational Limits Instrumentation System (UK Interim Defence Standard 00-55), the MULTOS CA (a high-security system developed to the standards of ITSEC level E6), and the Lockheed C130J Mission Computer (DO-178B Level A). A less successful project is also described. The lessons learnt from these projects show that SPARK offers a cost-effective approach for the construction of high-integrity software when it is deployed judiciously within an appropriate software development process.

1.1 Keywords

SPARK, Ada, static analysis, proof, Def. Stan. 00-55, ITSEC E6, DO-178B, industrial case studies.

2. INTRODUCTION

Most engineers in the Ada industry know something of SPARK, following a number of papers and tutorials in this and other conferences [1], and the publication of “The SPARK book”[2]. Simply knowing **what** SPARK is, though, is only half the story. Further pertinent questions include “Who’s using SPARK?”, “What factors separate successful from unsuccessful SPARK projects?”, and “How can SPARK help meet the various industry standards for critical software?”

SPARK is an annotated sublanguage of Ada that is suitable for the construction of high-integrity systems. The design of SPARK aims to eliminate ambiguity, erroneous behaviour, and implementation dependence. The language is also amenable to strong forms of static analysis, such as information flow analysis, and program proof, via its supporting tool—the SPARK Examiner.

This paper considers a number of real-world projects in an attempt to illustrate how SPARK meets the needs of various industries and standards.

3. SHOLIS

The Ship/Helicopter Operational Limits Instrumentation System (SHOLIS) is a ship-borne computer system that advises ship’s crew on the safety of helicopter operations under various scenarios. It is a fault-tolerant, real-time, embedded system and is the first system constructed to attempt to meet all the requirements of UK Interim Defence Standard (IDS) 00-55 [3] for safety-critical software. IDS 00-55 sets some bold challenges: it calls for formalized safety management and quality systems, formal specification of the system’s behaviour, formal proof (at both the specification and code levels), fully independent verification and validation, and static analysis of program properties such as information flow, timing and memory usage. The software for SHOLIS was specified, designed and developed by Praxis Critical Systems.

SHOLIS is by no means a trivial program, comprising some 13000 declarations and 14000 statements. Various forms of static analysis were used, according to a unit’s position in the system, its safety-integrity level, and the software hazards analysis. All software was subject to full information flow analysis and proof of freedom from predefined exceptions. The code which was designated as being safety critical was also subject to proof of partial correctness against its specification, which was written in the Z notation [13]. Proof that the system’s top-level safety properties were maintained by the code was also carried out.

The depth and breadth of static analysis used on SHOLIS is probably the most novel aspect of the project. Static analysis was used to show separation of critical and non-critical functions. Information flow analysis and proof of the absence of predefined exceptions were used to show functional separation, while static analysis of I/O usage, memory and timing were used to show the separation of non-functional properties.

The code proof discharged some 9000 verification conditions—the largest such effort we were aware of at the time. When SHOLIS was constructed (1996 and 1997), proof was traditionally seen as both too difficult ("don't you need a Ph.D. in Maths") and having no effective tool support. Both of these points were found to be untrue. While it is undoubtedly true that skilled staff are required, the qualification and experience required are only those that would be expected for a qualified engineer working in a safety-critical environment anyway. The tool support available at the time was certainly sufficient, although we did find many areas where the SPARK Examiner could be improved—many of these discoveries have since been incorporated in subsequent Examiner releases.

Finally it is worth noting that machine resources have increased by some two orders of magnitude since SHOLIS was developed. SHOLIS was developed by a team of approximately 8 engineers using a **single** UNIX server—it is now readily possible to supply at least ten times that computing power to **every** engineer at reasonable cost. A standard $1000 PC is now capable of supporting significant proof work, most of which is automated by theorem-proving tools. Proofs which took days to simplify and replay in the past could be reproducible in minutes today. The entire proof of SHOLIS could be reproduced overnight with reasonable ease. This realization brings "regression proof" (as opposed to "regression testing") within reach for new projects—a technique we hope to field in future. More information on SHOLIS, and in particular the proof activities, can be found in [4]. The most important finding was that proof (of both Z and code) was significantly more cost-effective at finding faults than traditional testing activities.

4. The MULTOS CA

The Multi-Application Operating System (MULTOS) is a smart-card OS that allows several applications to reside on a single "MULTOS Carrier Device" (MCD)—more commonly known as a "smart-card". MULTOS enforces separation of applications, and applications can be loaded and deleted dynamically. A key security concern is the prevention of forging MCDs and applications. To this end, the data that is used to enable MCDs and applications includes digital certificates, which are signed by the MULTOS Certification Authority (CA).

A computer system at the core of the CA issues these certificates, and is subject to the most stringent security constraints. The software for this system was designed and developed by Praxis Critical Systems to meet the standards of the UK ITSEC scheme [5] at the most demanding "E6" level.

The system is distributed: a single standard PC acts as the user-interface, but performs no security critical functions. A second group of industrial PCs are located in a tamper-proof environment—these perform all security critical functions, such as the signing of certificates, encryption of output files, and the generation of cryptographic keys.

The software developed for the CA has a slightly novel architecture. The following requirements were considered:

Availability. The software is designed to run uninterrupted, and cannot be upgraded or even restarted without significant effort. Avoiding memory-leaks and unexpected behaviour (e.g. exceptions) was therefore a major goal. The system is also designed to withstand the total failure of one or more machines in the tamper-proof environment.

COTS. The developers decided to use as little off-the-shelf software as possible, since the security and failure properties of such components could not be depended upon. For instance, we chose to design and implement our own inter-process communications and remote procedure call mechanisms, rather than relying on some COTS solution.

Lifetime. The system has an expected lifespan of decades. "Fast-moving" development techniques or technologies (i.e. those that weren't likely to be "in fashion" next year) were rejected.

Separation of Security Concerns. Each part of the system was classified as security-enforcing, security-related, or not secure. In particular, the entire user-interface and the software outside of the tamper-proof environment are considered insecure. The GUI is "dumb" in that it knows nothing of the application. All data coming from the GUI is considered insecure, and is rigorously validated by the system. Data displayed on the GUI was carefully analysed as having no threat to security.

Throughput. The CA is required to generate certificates at a significant rate. This entailed the provision of specialized cryptographic hardware and required concurrency to be employed in some particularly time-consuming operations.

The following table shows the programming languages used, the rough proportion of the total software written in each language, and the main functions performed. Coding the entire system in SPARK was judged to be simply impractical. Several functions, such as the database interface, the interface to the Win32 API, top-level concurrency, and the GUI were clearly beyond the scope of SPARK—the mixed-language approach reflected a simple "right tools for the job" approach to the construction of the various subsystems.

SPARK	30%	"Security kernel" of the tamper-proof software.
Ada95	30%	Infrastructure (concurrency, inter-task and inter-process communications, database interfaces etc.), bindings to ODBC and Win32.
C++	30%	All GUI components (Microsoft MFC)
C	5%	Device drivers and some standard cryptographic algorithms.
SQL	5%	Database stored procedures.

It is worth noting that SPARK is almost certainly the only industrial strength language that meets the requirements of ITSEC E6, which not only calls for the use of standardized languages, but goes on to require that "The definition of the programming languages shall define unambiguously the meaning of all statements used in the source code" [6].

The use of Ada95 largely followed a "Ravenscar-like" profile [15]. All partitions consist of a fixed number of library level tasks communicating via protected objects and rendezvous. Dynamic allocation (of tasks, memory etc.) was avoided. We also avoided language features whose implementation was still unproven, such as user-defined storage pools, controlled types, asynchronous transfer of control, requeue and so on. Communication between processes is achieved using Win32 named pipes, rather than using the facilities of the Ada95 distributed systems annex. This was largely a practical choice—the DSA was not implemented by the project's compiler when the project started.

The security-enforcing core of the system is implemented in SPARK. The static analysis offered by SPARK proved useful here. Dataflow errors can cause subtle security problems—for example, an uninitialised variable might just get an initial value which happens to be a piece of cryptographic key material "left over" on the stack from the execution of another subprogram. The absence of such problems in SPARK is a useful property.

Information-flow analysis also proved useful. The separation of some data sections (i.e. "Information stored in variable X cannot end up leaking into variable Y") gave confidence that certain security properties were being maintained by the code. This use of the Examiner is an interesting aside: the research conducted in the late 1970's that led to the development of SPARK was originally aimed at the needs of high-security computing [7][8]. Clearly, this research can be judged successful—it just took 20 years to find a practical, commercial application!

The MULTOS CA demonstrates the use of SPARK in a large, mixed-language development. A crucial part of the design is the split between Ada95 and SPARK in the security-related software. This was considered at great length in the early days of the project. It is a common misunderstanding that SPARK is an "all-or-nothing" language, but this is never the case in practice—even SHOLIS included some small units coded in assembler. "Drawing the line" between SPARK and non-SPARK is a crucial design activity, embodied in our "INFORMED" design approach [9], which is now delivered to all SPARK users.

5. LOCKHEED C130J

The Lockheed C130J is the latest in a long line of military and commercial transport aircraft, commonly known as the "Hercules". The C130J shares the same airframe design as previous models, but features significantly improved avionics, engines, and (most noticeably) 6-bladed composite propellers.

The core of the new avionics system is the Mission Computer (MC), which performs the majority of mission critical functions on the aircraft.

The MC is specified using the CoRE technique [10], using so-called "Parnas Tables." These offer a formal tabular notation for relating inputs (or "monitored variables") to required outputs (or "controlled variables"). The core of the MC software (approximately 80%) is implemented in SPARK, and was subject to information flow analysis prior to final integration and testing. SPARK was found to be a natural fit with the larger context of Lockheed's mature development processes, which aim for "Right first time" software.

Unusually, the aircraft was to be dual-certified for both civil and military use. This required it to meet several standards, including DO178B[14] for civil applications. The UK Royal Air Force, as lead military customer, required certain additional verification activities, most notably static analysis.

In this area, SPARK proved to be a major success. MC/DC test coverage analysis is known to be extremely expensive to carry out, is hard on staff morale and time, and often forms a serious bottleneck in project owing to its dependence on the availability of real target hardware. There is also some question over its effectiveness as a verification activity, since the effort required is often at the expense of other (possibly more useful) activities.

In meeting the needs of DO-178B effectively, some simple observations can be made:

Don't debug erroneous programs on the target. Programs which contain a data-flow error are said to be "erroneous" in Ada terminology. Such defects are notoriously difficult to find, especially on a target system that might have limited debugging and I/O support. In one such case, some 30 person days (and some nights) of rig-based testing were spent failing to find a simple data-flow error in a function. The Examiner detected this problem trivially, needing approximately 1 person-*hour* of effort to locate and analyse the offending package.

The target forms a bottleneck. Most projects employ a large team of engineers, but typically only have one or two realistic target systems, which quickly become a bottleneck in integration, testing, and verification if too much reliance is placed upon them.

Simplify code structure. The complexity of MC/DC analysis is directly related to the structural complexity of the source and object code. A key goal, therefore is to eliminate unnecessary code, such as predefined checks, from the object code.

These observations lead directly to some key features of SPARK, and how they fit within this context:

Static analysis. Semantic checking, data- and information-flow analysis, and program proof are all forms of **static** analysis—they are performed without running the program, and can be performed on incomplete programs during development. Moreover, such analysis can be performed by all project engineers, without access to the final target hardware. SPARK eliminates erroneous behaviour (such as the above-mentioned data-flow errors), so these problems cannot even reach the integration and test phase.

This approach was used on the C130J MC to great effect. Lockheed have reported an 80% saving in the expected budget allocated to MC/DC testing, yet coding proceeded at near normal Ada rates. On a system the size of the MC, this represents a significant sum of money! SPARK cannot take all the credit for this: other significant factors included the maturity of Lockheed's processes, the use of formal requirements, and the ability to generate structural test cases directly from the CoRE specification of units. SPARK *did* contribute directly, in that the code reaching MC/DC testing exhibited an unusually low fault density, reported to be less than one tenth of the expected industry norm for safety critical software—a direct testament to the usefulness of static analysis within the process. Furthermore, this saving was achieved by using only the most basic level of SPARK flow analysis—proof work was not carried out prior to MC/DC testing (although this was later performed in the UK).

In the context of DO-178B, SPARK offers further useful facilities:

Proof of exception freedom. It is common practice when Ada is used in real-time embedded systems to compile "with checks off" to reduce code size and improve performance. This approach carries some risk, though, since a program may actually contain instances of predefined exceptions. Confidence in the code is usually built through informal analysis such as code reviews, or incomplete techniques such as testing.

SPARK offers an alternative: the proof of the absence of predefined exceptions. These proofs are a static analysis (and so, again, can be conducted earlier than testing, and without reliance on target hardware), and are valid **for all input data**, offering a qualitative improvement over testing. When such proofs are conducted, run-time checks can be disabled with confidence and evidence can be produced to show that this is justified.

Simple object code. We have previously demonstrated that SPARK can be compiled with little or no support from a run-time library [11]. Moreover, if run-time checks are disabled as described above, then the resulting object code is simplified. This implies a significant reduction in the effort required to conduct MC/DC analysis.

More detail on the C130J, and its use of SPARK, can be found in [12].

6. A LESS SUCCESSFUL PROJECT

This project aimed to build a SIL4, real-time embedded control system. The project chose a CASE-driven object-oriented design style, based on the Shlaer/Mellor notation with the expectation that this would provide rapid development. SPARK was selected to meet the regulatory requirements, but the code was not constructed in SPARK. Instead, the project aimed to convert the code into SPARK **after** testing.

Initial signs were good—the project reported rapid progress in design and coding. Software integration and the first attempts to "SPARKify"[1] the code showed dark clouds gathering on the horizon. The code-generator used by the CASE tool "flattened" the structure of the code so that all state was at the same (i.e. global) level. In SPARK, this goes against good practice, which encourages the use of abstraction, refinement, and hierarchy of state. The code-generator also generated code which violated some of the static semantic rules of SPARK, so actual code changes had to be implemented late in the project—these were unfortunately seen as "distortions" of the original design.

Progress slowed significantly at this stage, and the integrated system did not meet its requirements. At present, the scope and requirements of the system are being reconsidered.

7. CONCLUSIONS

The four projects described in this paper have illustrated a number of points.

- SPARK's **early** adoption in a project and its influence on **design** of systems are perhaps the most important factors in successful projects. SPARK is sometimes criticized as being "just a programming language", but

[1] SPARKify. *v.tr.colloq.*/spa:kɪfΛɪ/ to turn Ada into SPARK after it's been written. An unfortunate verb we wish had never been invented.

our experience in this area shows quite the opposite—the judicious use of SPARK can have a profound (and we hope positive) influence on systems' architecture, design, verification, and cost.

- Retrospective "SPARKification" of code is ill-advised, and often leads to significant difficulty.
- CASE tools and their associated code-generators do not (currently) know enough about SPARK. Their favoured design and code-generation strategies may lead to code which is perfectly acceptable Ada, but which does not follow our recommended guidelines for SPARK.
- SPARK has been both a commercial and a technical success in meeting three of the most stringent software standards—Def. Stan. 00-55, DO-178B level A, and ITSEC E6. In particular, on the C130J, SPARK contributed to a significant commercial as well as a technical success.
- The SHOLIS project has shown that program proof is now a deployable and reasonable verification technology. The proof of exception freedom has been shown to of major benefit in meeting the needs of DO-178B level A, where the difficulties and cost of target-based testing and MC/DC coverage analysis are so pronounced.

8. ACKNOWLEDGEMENTS

The author would like to thank Jim Sutton of Lockheed Martin, Nick Williams, Jonathan Hammond and Peter Amey of Praxis Critical Systems, and Dave Roberts and Dave Meadon of Mondex International for their comments on an early draft of this paper.

9. REFERENCES

[1] Barnes, J., The SPARK way to Correctness is Via Abstraction. Proceedings of ACM SigAda 2000.

[2] Barnes, J., High Integrity Ada - The SPARK Approach. Addison Wesley, 1997.

[3] MoD. The procurement of safety critical software in defence equipment. UK Ministry of Defence, April 1991. Interim Defence Standard 00-55 Parts 1 (Requirements) and 2 (Guidance).

[4] King, S., Hammond, J., Chapman, R.C., and Pryor, A. Is Proof More Cost Effective than Testing? IEEE Transactions on Software Engineering. August 2000.

[5] Information Technology Security Evaluation Criteria (ITSEC), Provisional Harmonised Criteria, Version 1.2, June 1991.

[6] UK ITSEC Developers' Guide, UK Scheme Publication No. 4, Part II: Reference for Developers, Issue 1.0, July 1996.

[7] Bergeretti, J-F., and Carré, B. A., Information-Flow and Data-Flow Analysis of While Programs. ACM Transactions on Programming Languages and Systems, Vol. 7, No. 1, January 1985. p.p. 37–61.

[8] Cohen, E., Information Transmission in Sequential Programs. In Foundations of Secure Computing, R. A. DeMillo et al., Ed. Academic Press, New York, 1978, pp. 297-335.

[9] Praxis Critical Systems. INFORMED Design Method for SPARK. S.P0468.42.4, Issue 1.0, January 1999.

[10] Software Productivity Consortium. Consortium Requirements Engineering Guidebook, SPC-92019-CMC Version 02.00.03, Hendon, VA, USA. December 1993.

[11] Chapman, R.C., and Dewar, R.B.K., Re-engineering a Safety Critical System using SPARK95 and GNORT. in Reliable Software Technologies - Ada Europe 1999. Harbour, M., and J. De la Puente Eds., Springer Verlag Lecture Notes in Computer Science Vol. 1622, 1999. pp. 39-51.

[12] Croxford, M., and Sutton, J., Breaking Through the V and V Bottleneck. in Ada Europe 1995, Springer Verlag Lecture Notes in Computer Science Vol. 1031, 1996.

[13] Spivey, J.M., The Z Notation - A Reference Manual. 2nd Edition. Prentice Hall. 1992.

[14] RTCA. Software Considerations in Airborne Systems and Equipment Certification. RTCA/DO-178B, 1994.

[15] Burns, A., Tasking Profiles, in Proceedings of the 8th International Real-Time Ada Workshop. ACM Ada Letters, September 1997.