



SPARK – a state-of-the-practice approach to the Common Criteria implementation requirements

Rod Chapman

Publication notes

Presented at the 2nd International Common Criteria Conference, Brighton, UK, July 2001.

SPARK—A state-of-the-practice approach to the Common Criteria implementation requirements

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

1. ABSTRACT

The Common Criteria (CC) require the use of programming languages whose statements have an "unambiguous meaning." This presentation considers SPARK[1]: a widely-used language that is perhaps unique in actually meeting this requirement. While SPARK has its roots in security research, it is currently most widely used in the aerospace and rail industries, and has a well-established track record in meeting the most demanding standards in these domains, such as UK Def. Stan. 00-55 (for military systems) and DO-178B (for civil aviation). SPARK has recently proven, though, to be ideally suited to the development of secure systems.

The design principles of SPARK will be described, highlighting the language's suitability for meeting the requirements of secure systems development. SPARK's static analysis tool (the Examiner) will also be considered, concentrating on the types of analysis, such as information flow analysis and proof of exception freedom, that can be achieved. The strengths of the language will be illustrated with reference to the MULTOS Global Key Centre (MGKC) - the system at the heart of the MULTOS CA - which was developed by Praxis Critical Systems using SPARK to meet the requirements of ITSEC E6.

1.1 Keywords

SPARK, Ada, Static analysis, Proof, ITSEC, Common Criteria, Industrial case studies.

2. Implementation Challenges in the Common Criteria

The Common Criteria set some bold challenges for the implementation of secure systems. In particular, at higher evaluation levels, the degree of formality required in design and construction becomes significant. The use of formal methods in the specification of systems is well known, but the use of formal implementation languages remains less common. Experience shows that "implementation slips" remain a serious source of security breaches—the ubiquitous "buffer overflow" for instance.

The CC requirements most obviously relevant to this problem are ALC_TAT, ADV_IMP, ADV_INT, and ADV_RCR. For instance ALC_TAT.1.2c states

"The documentation of the development tools shall unambiguously define the meaning of all statements used in the implementation."

What does this mean in practice? One view of this requirement is that the programming language used must have an unambiguous definition, yet can *any* current widely-used programming language actually claim to meet this requirement?

3. Requirements for High Integrity Languages

Here, we introduce the term “High Integrity” to cover not just highly-secure systems, but system in general where the cost of loss or failure is significant. This definition covers safety-critical systems (such as those controlling aircraft, nuclear power plants and so on), and systems where failure may result in significant financial loss or embarrassment—the recent high-profile failures of Formula 1 “launch control” systems spring to mind. This term is useful, since the problems facing high-integrity software development have much in common with those found in the development of secure systems.

In the safety-critical domain, the use of *static analysis* is wide-spread. In its most general definition, this simply refers to any form of analysis that does not involve actually running a particular program. Static analysis techniques applicable to software therefore range from simple code reviews through to full mathematical proof of program properties.

A key realization here is that to enable static analysis to produce useful results, the notation being analyzed must be as *precise* (i.e. free from ambiguity) as possible. In the presence of ambiguity, static analysis techniques have to make assumptions (e.g. “we’ll assume the compiler evaluates expressions left-to-right”) which can render the results somewhat dubious. Alternatively, a static analysis tool may attempt to cover all the possible outcomes of any ambiguity—this typically leads to an explosion in analysis time that makes the tool unusable.

Unfortunately, no current standard programming language meets this need. To explain why, we must briefly consider recent trends in programming language design.

Firstly, the definition of languages have always contained deliberately ambiguous or “implementation dependent” characteristics (see ALC_TAT.1.3c for the CC’s response to this problem.) For instance, Ada is specified to allow floating-point mathematics on a range of target architectures, which may or may not implement the IEEE 754 standard for such operations. This is a perfectly sensible design choice on the part of the language designers, but means that the exact meaning of a floating-point program does indeed depend on the compiler and target CPU in question.¹ Other implementation dependencies that affect almost all common languages are evaluation order of expressions, order of association of parameters, and subprogram parameter passing mechanism.

Another issue is the recent trend in programming language design towards more and more dynamic language features. C++ brought Object-Oriented Programming to the world, where *run-time* decisions dramatically affect program semantics, such as dynamic dispatch and polymorphism. Java brings garbage collection to the party and so on. Such language have also pushed for performance over safety—the legacy of C’s unchecked arithmetic operators and array indexing still haunts us today.

These language features are dramatically at odds with the needs for static analysis, and so are *not appropriate* for the construction of high-integrity systems. This has been broadly accepted in the safety-critical community, where the use of *high-integrity subset languages* is the norm.

Borrowing from Anderson’s well-known analogy, it is clear that if you’re programming Satan’s computer, you should not use Satan’s programming language!

4. SPARK—An Angelic Programming Language

The SPADE Ada Kernel (SPARK) is a programming language designed to be appropriate for the development of high-integrity systems.

SPARK is a high-level programming language, designed for writing software for high integrity systems. The executable part of the language is a subset of Ada[6], but the language requires additional annotations that make it

¹ Somewhat oddly, the original definition of Java™ did call for IEEE floating point. Pedants would argue that this actually forbids the implementation of Java on particular architectures. Whether this was luck or judgement on the part of Java’s designers remains a moot point.

possible to carry out data and information flow analysis[3], and to prove partial code correctness, using the commercial toolset associated with the language: the SPARK Examiner, Simplifier and Proof Checker.

4.1 A brief tour of SPARK

There were several design goals behind the choices as to what parts of Ada should be removed to form the SPARK programming language:

Logical soundness: there should be no ambiguities in the language;

Simplicity of formal description: it should be possible to describe the whole language in a relatively simple way;

Expressive power: notwithstanding the previous two factors, the language should be rich enough to describe real systems;

Security: it should be possible to determine statically whether a program conforms to the language rules;

Verifiability: program verification should be not only theoretically possible, but also tractable for industrial-sized systems;

Bounded time and space requirements: in order to avoid the possibility of run-time errors caused by exhausting finite resources such as time and space, the resource requirements of a program should be determinable statically.

Minimal runtime library: SPARK is designed to be compiled with no supporting runtime library. This possibility has been demonstrated by various compiler vendors, originally in response to the needs of commercial avionics, where the certification of such libraries remains a significant hurdle. Similarly, this facility is useful in the development of security-related products, where evaluation of such COTS components remains troublesome or expensive.

Together, these considerations led to decisions to omit several features of Ada: the goto statement, aliasing, default parameters for subprograms (i.e. procedures and functions), side-effects in functions, recursion, tasks, user-defined exceptions, exception handlers and generics. In addition, several other features, such as the type model, are simplified: there are no access types (pointers), type aliasing, derived types or anonymous types. Apart from these exclusions and restrictions, the normal Ada package structure is used for programming, with its distinction between package interfaces (or specifications) and package bodies. Within a package, further structuring is possible using procedures and functions and it is at this level that we can see the first of the new annotations.

Annotations are user-supplied comments that are ignored by an Ada compiler, but processed by the SPARK tools. The first group of annotations is concerned with data and information flow analysis—these are the *global*, *derives*, *own*, and *inherit* annotations.

The *global* and *derives* annotations between them specify the information needed for data and information flow analysis of individual subprograms. Data flow analysis involves checking that global variables and parameters are used in the expected way: imported variables can only be read from, exported variables can be written to, and variables that are both imported and exported can be read from and written to. There are also checks that variables are not being read before being initialized with a value, that values are not overwritten before being read, that all imported variables are actually used somewhere, and so on.

Data flow analysis does not examine dependencies between variables. However, information flow analysis uses the *derives* annotation (which for each exported variable lists the imported variables on which its final value is expected to depend) to check that the actual dependencies in the code match what is intended. Both data and information flow analyses are decidable, and entirely automated by the SPARK Examiner.

The *own* and *inherit* annotations are used for scoping and structuring. The *own* annotation is used to declare the existence of state variables inside a package: the values of these variables are preserved between calls of subprograms in the package. The *inherit* annotation makes visible the items from another package scope, e.g. it allows the annotations in a package to refer to the own variables of the inherited package.

The second group of annotations is used for code verification.

The *pre* and *post* annotations are found in the specification of a procedure, and are used for the traditional precondition and postcondition of the procedure—*pre* gives a predicate on the input parameters and initial state (imported) variables, while *post* relates input and output parameters and initial and final state (exported) variables. On non-looping programs, the SPARK Examiner produces proof obligations by 'hoisting' the postcondition through the procedure body and checking that the supplied precondition implies this transformed postcondition. For looping programs, the *assert* annotation is used to specify the loop invariant. The verification conditions (VCs) generated by the Examiner for looping programs check for partial correctness: separate arguments are needed to consider loop termination, if total correctness is required. Finally, the *return* annotation is used to define (explicitly or implicitly) the result of a function, thus allowing checking of functions to be carried out at a more abstract level.

The SPARK Examiner has a mode of operation where, in addition to the VCs generated by the flow analysis and proof annotations, it also generates VCs that, if discharged, would guarantee that the SPARK program could not raise any run-time exceptions. The design of the SPARK language itself ensures that the Ada exceptions *Tasking_Error* and *Program_Error* can never arise in a SPARK program. In addition, since SPARK is designed so that the space requirements can be computed statically, it is possible to guarantee that *Storage_Error* cannot be raised. The only remaining possible exception is *Constraint_Error*, and the restrictions on the SPARK language mean that this can only be caused by a division check, an index check, a range check or an overflow check. When invoked with the run-time check (RTC) option, the SPARK Examiner generates VCs for the first three of these checks (e.g. that all divisors are not zero) and the VCs for the overflow check (e.g. that $A+B$ does not overflow in the expression $(A+B)/2$) can be generated by the RTC plus Overflow option.

There are two possible routes for discharging the VCs produced by the Examiner: the Simplifier and the Proof Checker. The Simplifier is an automatic tool that carries out routine simplification using a collection of rules. If a VC cannot be discharged by the Simplifier, then a developer can invoke the Proof Checker, which is an interactive assistant allowing exploration of the problem and (it is hoped) the construction of a proof.

4.2 SPARK and the development of secure systems

SPARK actually has its roots in the security community. Research in the late 1970's (e.g. [7][4]) into information flow in programs led to research at DERA Malvern and Southampton University that resulted in SPADE Pascal and, eventually, SPARK. SPARK has been widely used in safety-critical systems, for example most risk-class 1 systems on the European Fighter Aircraft are constructed in SPARK, but remains less well-known in the security community.

Several features of SPARK, though, make it ideal for the development of secure systems. These include:

- Program-wide, complete data- and information-flow analysis. These analyses make it impossible for a SPARK program to contain a dataflow error (e.g. the use of an uninitialized variable)—a common implementation error that can be the cause of subtle (and possibly covert) security flaws.
- Proof of correctness of SPARK programs is achievable, and so allows a program to be shown to correspond with some suitable formal specification. This allows for formality in the design and specification of a system to be extended through its implementation.
- Proof of the absence of predefined exceptions (for such things as buffer overflows, divide by zero) offer strong *static* protection from a large class of common security flaw. Such things are an anathema to the

safety-critical community, yet remain possibly the most common form of attack against networked web-servers and so on.

- SPARK can be compiled with absolutely no supporting run-time library. This implies that a SPARK application can be delivered with no COTS component. At the highest assurance levels, this may be of significant benefit, where evaluation of such components remains problematic.

5. The MULTOS CA

The Multi-Application Operating System (MULTOS[8]) 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 principal security concern is the prevention of forged 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 [2] 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, such as CORBA or COM.

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 analyzed 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 SHA1 algorithm.
SQL	5%	Database stored procedures.

The use of Ada95 largely followed a "Ravenscar-like" profile [5]—a subset of Ada's tasking facilities designed for use in high-integrity applications. All partitions consist of a fixed number of tasks communicating via protected objects (a form of controlled shared state) and rendezvous (a form of synchronous inter-task message pass). Dynamic allocation (of tasks, memory etc.) was avoided. We also avoided language features whose implementation was still unproven in Ada95. Communication between processes is achieved using Win32 named pipes, rather than using the facilities of the Ada95 distributed systems annex (DSA). 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 uninitialized 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. Finally, SPARK rewards an information-flow centric design, which promotes high cohesion and loose coupling of components. This property proved useful during development as requirements changes (and the subsequent rework of testing) were introduced.

6. CONCLUSIONS

Programming Satan's computer might be just about possible. SPARK is certainly not a magic bullet, but has a significant track record of success in the implementation of high-integrity systems. SPARK, we believe, is unique in actually meeting the implementation requirements of the ITSEC and CC schemes. SPARK's support for strong static analysis and proof of program properties (e.g. partial correctness or exception freedom) means that the CC requirements for formal development processes can be met. The simplicity of the language subset itself and the data- and information-flow analysis offered by the Examiner make a large class of common errors simply impossible to express in SPARK. The development of the MULTOS CA has shown SPARK to be a successful contributor in the development of secure systems meeting the most stringent standards.

7. ACKNOWLEDGEMENTS

The author would like to thank John Beric of Mondex International for his comments on an early draft of this paper.

8. REFERENCES

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

- [2] Information Technology Security Evaluation Criteria (ITSEC), Provisional Harmonised Criteria, Version 1.2, June 1991.
- [3] 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.
- [4] 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.
- [5] Burns, A., Tasking Profiles, in Proceedings of the 8th International Real-Time Ada Workshop. ACM Ada Letters, September 1997.
- [6] Reference Manual for the Ada Programming Language. International Standards Organization. ISO/IEC 8652:1995.
- [7] Denning, D. E., and Denning, P. J. Certification of Programs for Secure Information Flow. CACM Vol. 20, No. 7. July 1977.
- [8] MULTOS. See www.multos.com