Correctness by Construction: Better Can Also Be Cheaper

Peter Amey
Praxis Critical Systems

For safety and mission critical systems, verification and validation activities frequently dominate development costs, accounting for as much as 80 percent in some cases [1]. *There is now compelling evidence that development methods that focus on bug prevention rather than bug detection can both raise quality and save time and money. A recent, large avionics project reported a four-fold productivity and 10-fold quality improvement by adopting such methods* [2]. *A key ingredient of correctness by construction is the use of unambiguous programming languages that allow rigorous analysis very early in the development process.*

In December 1999 CROSSTALK [3], David Cook provided a well-reasoned historical analysis of programming language development and considered the role languages play in the software development process. The article was valuable because it showed that programming language developments are not *sufficient* to ensure success; however, it would be dangerous to conclude from this that they are not *necessary* for success. Cook rightly identifies other issues such as requirements capture, specifications, and verification and validation (V&V) that need to be addressed.

Perhaps we need to look at programming languages not just in terms of their ability to code some particular design but in the influence the language has on some of these other vital aspects of the development process. The key notion is that of the benefit of a *precise* language or language subset. If the term *subset* has set anyone thinking "oh no, not another coding standard," then read on, the topic is much more interesting and useful than that!

Language Issues

Programming languages have evolved in three main ways. First came improvements in *structure*; then attempts at improving compile-time error detection through such things as *strong typing*; and, most significantly, facilities to improve our ability to express *abstractions*. All of these have shaped the way we think about problem solving.

However, programming languages have not evolved in their precision of expression. In fact, they may have actually gotten worse since the meaning of a sample of machine code is exact and unequivocal, whereas the meaning of the constructs of typical modern high-order languages are substantially less certain. The evolution of C into C++ certainly improved its ability to express design abstractions but, if anything, the predictability of the compiled code decreased. These ambiguities arise either from deficiencies in the original language definition or from implementation freedoms given to the compiler writer for ease of implementation or efficiency reasons.

> *"Programming languages have not evolved in their 'precision' of expression. In fact, they may have actually gotten worse ..."*

None of this may look like a very serious problem. We can still do code walkthroughs and reviews and, after all, we still have to do dynamic testing that should flush out any remaining ambiguities. In fact the evidence is quite strong that it *does* matter because it creates an environment where we are encouraged to make little attempt to reason about the software we are producing at each stage of its development. Since we typically do not have formal mathematical specifications and we use imprecise programming languages, the first artifact we have with formal semantics is object code (object code is formal in the sense that the execution machine provides operational semantics for it). So our first opportunity to explore the behavior of the software in any rigorous fashion occurs very late in the development cycle with malign consequences.

Where this trend is most harmful is in high-integrity systems where reliability is the pre-eminent property required. The most common form of high-integrity system is the safety-critical system. Such systems are characterized by the proportionally very high overall effort that goes into showing that the system is fit for service: the V&V effort. We are seeking to demonstrate *before there is any service experience* that a system *will be* fit for use.

Claimed failure rates of 10^{-9} per flying hour are not uncommon in aviation: 10^{9} hours is more than 114,000 years! Leaving aside for now the question of whether we can ever hope to demonstrate such levels of integrity by dynamic testing alone [4, 5, 6], what is certain is that any such attempt will be expensive. For high-integrity systems where typically more than half – and sometimes as much as 80 percent – of the time is spent in the integration and validation phases, we are locked into a vicious circle: We are spending most of our time at the most expensive point in the life cycle. Worse, it is the point at which any delay will inevitably affect the overall program schedule.

The consequences of these difficulties are well recognized. It is often stated that developing software to a standard such as DO-178B[1] [7] at level A raises the cost by a factor of five over non-critical developments. Much of this extra cost comes from meeting the specific test coverage criterion of the standard.

Reliable Programming in Standard Languages

If we want to avoid the vicious circle of late error detection and costly repair we must start to reason logically about our software at an earlier stage in its development. We can do this by using a programming language whose source code has a precise meaning; this makes it possible to provide tool support in the form of static analyzers[2] that can be applied very early in the coding process, *before dynamic testing begins*.

This kind of early analysis, at the engineer's terminal, is an amplification of the static analysis (such as type checking) performed by compilers. The aim is to prevent errors ever making it to test. These gains are only possible if the language rules are precise and the semantics are well defined.

Given the imprecision of programming languages in general we have three ways of gaining the properties we seek:

- Work with particular, compiler-defined dialects of programming languages.
- Design new languages with the properties we require.
- Use subsets of mainstream languages designed to have the required properties.

Using dialects is quite a respectable approach but not without its drawbacks. First, vendors may be reluctant to reveal the dialect. Also, there is no guarantee that the compiler's behavior won't change from version to version and without your knowledge. Finally, there is no guarantee that the compiler will even be consistent in its behavior.

Purpose-designed languages with formally defined semantics are a fascinating research topic but are unlikely ever to achieve the kind of critical mass required to make them suitable for widespread industrial use. My favorite example is Newspeak [8]. It is named after the language in George Orwell's novel *1984*. Newspeak is elegant, precise and has all the required properties. It also has several major flaws: you can neither buy a compiler for it, nor a textbook, nor get training, nor hire staff! This is the fundamental drawback of the custom-language approach.

That leaves using coherent subsets of mainstream languages. Done properly this can provide the best of both worlds: precise semantics together with access to tools, training, and staff. However, typically these subsets are informally defined. Some language features may be omitted. There is no attempt to construct a logically coherent sublanguage that makes the qualitative shift to having no ambiguous or insecure behavior. For example, MISRA-C is a C-language subset defined by the United Kingdom (UK) automotive industry. MISRA-C [9] prohibits some of the more problematic constructs of C, such as unrestricted pointer arithmetic, which is frequently the cause of coding problems [10]. However, MISRA-C is not, and does not claim to be, a logically sound sublanguage.

Other subsets such as SPARK[3] [11, 12, 13] are more ambitious since they seek to define a language whose source code wholly defines the eventual compiled behavior. The language is intended to be completely free from ambiguity, compiler-dependent behavior, and other barriers to precise reasoning. Before going on to describe SPARK in more detail it is worth looking further at the practical advantages of correctness-by-construction approach.

The Lockheed C130J

The Lockheed C130J or Hercules II Airlifter was a major updating of one of the world's most long-lived and successful aircraft. The work was done at Lockheed's own risk. Much of the planned aircraft improvement was to come from the completely new avionics fit and the new software that lay at its heart. The project is particularly instructive because it has some unusual properties that provide some interesting comparisons:

"The exact semantics of SPARK require software writers to think carefully and express themselves clearly; any lack of precision is ruthlessly exposed by ... its support tool, the SPARK Examiner."

- Software subsystems developed by a variety of subcontractors using a variety of methods and languages.
- Civil certification to DO-178B.
- Military certification to UK Def-Stan 00-55 involving an extensive, retrospective independent V&V (IV&V) activity.

For the main mission computer software, Lockheed adopted a well-documented correctness-by-construction approach [14, 15, 16]. The approach was based on:

- Semi-formal specifications using Consortium Requirements Engineering (CoRE) [17] and Parnas tables [18].
- "Thin-slice" prototyping of high-risk areas.
- Template-driven approach to the production of similar and repetitive code portions.
- Coding in SPARK with tool-supported static analysis carried out as part of the *coding* process and certainly prior to formal certification testing; this combination was sufficient to eliminate large numbers of errors at the coding stage – before any formal review or testing began.

This logical approach brought Lockheed significant dividends. Perhaps most striking was in the reduced cost of the formal testing required for DO-178B Level A certification: "Very few errors have been found in the software during even the most rigorous levels of FAA [Federal Aviation Administration] testing, which is being successfully conducted for less than a fifth of the normal cost in industry." At a later presentation [2] Lockheed was even more precise on the benefits claimed for their development approach:

- Code quality improved by a factor of 10 over industry norms for DO 178B Level A software.
- Productivity improved by a factor of four over previous comparable programs.
- Development costs were *half* that typical for non safety-critical code.
- With re-use and process maturity, there was a *further* productivity improvement of four on the C27J airlifter program.

These claims are impressive but they are justified by the results of the UK Ministry of Defense's own retrospective IV&V program that was carried out by Aerosystems International at Yeovil in the UK. It should be remembered that the code examined by Aerosystems had already been cleared to DO-178B Level A standards, which should indicate that it was suitable for safety-critical flight purposes. Key conclusions of this study follow:

- Significant, potentially safety-critical errors were found by static analysis in code developed to DO-178B Level A.
- Properties of the SPARK code (including proof of exception freedom) could readily be proved against Lockheed's semi-formal specification; this proof was shown to be cheaper than weaker forms of semantic analysis performed on non-SPARK code.
- SPARK code was found to have only 10 percent of the residual errors of full Ada; Ada was found to have only 10 percent of the residual errors of code written in C. This is an interesting counter to those who maintain that choice of programming language does not matter, and that critical code can be written correctly in any language: The claim may be true in principle but clearly is not commonly achieved in

practice.
- No statistically significant difference in residual error rate could be found between DO-178B Level A and Level B code, which raises interesting questions on the efficacy of the MC/DC test coverage criterion.

Lockheed succeeded because they had a strong process with an emphasis on requirements capture and accurate specifications; furthermore, they made their process repeatable by using templates. All these are wise things recommended by David Cook [3] in his article.

SPARK's role in this process was crucial. Its precision helped expose any lack of clarity in the specifications to be implemented. The exact semantics of SPARK require software writers to think carefully and express themselves clearly; any lack of precision is ruthlessly exposed by the rigorous analysis performed by its support tool, the SPARK Examiner[4].

One of the first effects of the adoption of SPARK on the project was to make the CoRE specification much more of a central and "live" document than before. Instead of "interpreting" unclear requirements on their own, developers were nagged by the Examiner into getting the required behavior agreed upon and the specification updated to match; everyone gained from this.

The second SPARK contribution came with the UK military certification process. Here the fact that SPARK facilitates formal verification or "proof" came into play. As Lockheed put it: "The technology for generating and discharging proof obligations, based on SPARK ... was crucial in binding the code to the initial requirements."

The Lockheed process might be termed "verification-driven development" since it is a process that recognizes that showing the system to be correct is usually harder than making it correct in the first place. Therefore the process is optimized to produce the evidence of correctness as a by-product of the method of development.

The SPARK Language

SPARK is an annotated subset of Ada with security obtained by two complementary approaches. First, the more problematic parts of Ada, e.g., unrestricted tasking, are eliminated. Second, remaining ambiguities are removed by providing additional information in the form of annotations. Annotations are special comments that make clear the programmer's intentions. Since they are comments, the compiler ignores them, but they are read and cross-checked against the code by the SPARK Examiner. Annotations also serve an important purpose by providing stronger descriptions of the abstractions used; this means that it is possible to analyze a part of the system without needing to access all the package bodies. Again this facilitates early analysis.

SPARK is introduced here to illustrate how a precise, unambiguous programming language is constructed and the benefits it brings. The goals set out for the original authors, more than 10 years ago, were as follows.

Logical Soundness

This covers the elimination of language ambiguities as mentioned earlier. The behavior of a SPARK program is wholly defined by and predictable from its source code.

Simplicity of Formal Language Definition

In order to demonstrate SPARK's logical soundness, it was felt desirable to write a formal definition of its static and dynamic semantics. This challenging task was completed in 1994, fortunately without producing any nasty surprises.

Expressive Power

The aim here was to produce a language that was rich and expressive enough for real industrial use. You can of course produce a safe subset of any language if you make it so small that it is impossible to write real programs in it: If you cannot write anything, you certainly cannot write dangerous code!

SPARK retains all the important Ada features required for writing well-engineered object-based code. It has packages, private types, functions returning structured types, and all of Ada's control structures. This leaves a clean, easy-to-learn language that, while smaller than Ada, is still rich and expressive.

Security

SPARK has no unenforceable language rules. The static semantic rules of SPARK are 100 percent machine-checkable using efficient analysis algorithms. It is this feature that makes it feasible to consider static analysis to be part of the design and coding process rather than seeing it as a retrospective V&V activity.

Verifiability

This is a consequence of the exact semantics of SPARK. Since the source code has a precise meaning, it is possible to reason about in a rigorous mathematical manner. The SPARK Examiner can be used to facilitate proof that SPARK code conforms to some suitable specifications, or that it has certain properties. A very straightforward and useful facility is the ability to construct a proof that the code will be completely free from run-time errors (such as the predefined exceptions).

Bounded Space and Time Requirements

A very important property of many critical systems is that they should operate for long periods; this means, for example, that they should not suffer from such things as memory leaks. SPARK programs are inherently bounded in space – there is no recursion or heap allocation for example – and can be made bounded in time. The end result is that the required machine resources can be calculated statically.

Correspondence With Ada

This is how the "Newspeak" trap is avoided. All SPARK programs are legal Ada programs and can be compiled with any standard compiler. More usefully, the meaning of a SPARK program cannot be affected by the implementation freedoms that the Ada standard allows the compiler writer. For example, it does not matter whether the compiler passes parameters by reference or by copying, or in which order it evaluates expression; the compiled SPARK code will have the same behavior. In effect, to use the terminology of the Ada LRM, it is not possible to write *erroneous* Ada programs in SPARK.

Verifiability of Compiled Code

Since we are taking advantage of the precision of the language to reason about source code, we need to consider the accuracy with which the compiler will generate object code. Clearly SPARK cannot change the behavior of compilers but there is some evidence that the simplifications of Ada provided by SPARK tend to exercise the well-trodden paths of a compiler rather than its obscure back alleys. The resulting machine code seems to be easier to relate to the source than might otherwise be the case.

Minimal Run-Time System Requirements

This is an extremely important area. Complex languages that provide facilities for concurrency, exception handling, etc., require large run-time library (RTL) support. Since the RTL forms part of the overall system, we need to demonstrate its correctness just as we must the application

code itself. As the RTL is likely to be proprietary and opaque, this can be very difficult.

SPARK inherently requires very little run-time support; for example, the SHOLIS [19] system developed by Praxis Critical Systems made only *one* call to a small routine in a fixed-point math library. Many Ada compiler vendors supply small RTLs for certification purposes, and SPARK is compatible with all of these. The smallest of all is GNAT Pro High-Integrity Edition from ACT because this has no RTL at all. As a demonstration, the SHOLIS code was ported to this system [20].

As an aside, it is quite interesting to compare the language subsets supported by the compiler vendors with SPARK. The former are all produced by removing the complex and difficult-to-understand parts of their run-time systems and seeing how much of the language can still be supported. Reasoning about the semantics of the language itself and eliminating problematic areas makes the latter. Both approaches produce almost identical subsets.

Cost Saving Example

So how did SPARK help Lockheed reduce its formal FAA test costs by 80 percent? The savings arose from avoiding testing repetition by eliminating most errors before testing even began. During the early stages of the project, before SPARK was fully adopted, an illustrative incident occurred. This is described in [15], which states: "In one case, a code error in a standard Ada module had resisted diagnosis for one week using normal testing methods. The module was converted to SPARK ... and the error was then identified in 20 minutes through SPARK analysis. Efficiencies of this type were obtained repeatedly."

Later in the project, when SPARK was fully established, the savings were even greater because errors of this kind never got to the test stage at all. They were eliminated as part of the coding process.

The case in question took the form of a Boolean function with a significant number of possible execution paths. On one of these paths there was a *data flow error* resulting in a random value being returned for the function. Finding this by test was extremely difficult because it was not enough simply to traverse the faulty branch. It was also necessary to be lucky with the random value returned. Since most random bytes of memory are non-zero, and non-zero values are typically regarded as being Boolean "true" by the compiler, the function usually returned true when the incorrect branch was executed; unfortunately, this was the right answer for this test condition! So the function nearly always behaved correctly but, apparently inexplicably, returned the wrong answer under seemingly random conditions. Simplified, the example was as follows:

- **Specification Table.** Table 1 represents the required behavior of an aircrew alert system. The mon column shows the input value of a "monitored" variable, in this case showing the severity of an alert message. The con column shows the required output of a "controlled" variable, in this case a Boolean value saying whether an alarm bell should sound in the cockpit.
- **Flawed Implementation.** A flawed implementation of the function might be:

```
type Alert is (Warning, Caution, Advisory);
function RingBell (Event : Alert)
return Boolean
is
      Result : Boolean;
begin
      if Event = Warning then
            Result := True;
      elsif Event = Advisory then
            Result := False;
      end if;
      return Result;
end RingBell
```

- **SPARK Examination.** The analysis performed by the SPARK Examiner includes a mathematically rigorous data and information flow analysis [21] that uncovers *all* uses of undefined data values thus:

```
 13 function RingBell (Event : Alert)
    return Boolean
 14 is
 15   Result : Boolean;
 16 begin
 17   if Event = Warning then
 18         Result := True;
 19   elsif Event = Advisory then
 20         Result := False;
 21   end if;
 22   return Result;
                ^1
??? ( 1) Warning : Expression contains
    reference(s) to variable Result,
    which may be undefined.
 23 end RingBell;
??? (2) Warning : The undefined initial
    value of Result may be used in the
    derivation of the function value.
```

This clear indication is obtained *at the engineer's terminal* before compilation, before test, before the code is even checked back into configuration management. The error never enters the system so it never has to be found and eliminated.

mon_Event	con_Bell
Warning	True
Caution	True
Advisory	False

Table 1: *Specification Table*

Conclusion

Most high-integrity and safety-critical developments make use of language subsets. Unfortunately, these subsets are usually informally designed and consist, in practice, of simply leaving out parts of the language thought to be likely to cause problems. Although this shortens the length of rope with which the programmers may hang themselves, it does not bring about any qualitative shift in what is possible.

The use of coherent subsets free from ambiguities and insecurities does bring such a shift. Crucially it allows analysis to be performed on source code before the expensive test phase is entered. This analysis is both more effective and cheaper than manual methods such as inspections. Inspections should still take place but can focus on more profitable things like "does this code meet its specification" rather than "is there a possible data-flow error."

Eliminating all these "noise" errors at the engineer's terminal greatly improves the efficiency of the test process because the testing can focus on showing that requirements have been met rather than becoming a "bug hunt." In my 10-plus years of using SPARK, I have never needed to use a debugger. I have become so used to things working the first time that my debugging skills have almost completely atrophied. The only price I pay for this is the SPARK Examiner pointing at the source code on my terminal and displaying messages telling me I have been stupid again; I find I am grateful for those messages!

The SPARK language definition [11, 12] is both free and freely available (see <www.sparkada.com> or e-mail sparkinfo@praxis-cs.co.uk). Alternatively, John Barnes' textbook [13] provides an informal and approachable description of the language together with demonstration tools.◆

References

1. Private communication arising from a productivity study at a major aerospace company.
2. Sutton, James. "Cost-Effective Approaches to Satisfy Safety-critical Regulatory Requirements." Workshop

Session, SIGAda 2000.
3. Cook, David. "Evolution of Programming Language and Why a Language is not Enough." CROSSTALK Dec. 1999: 7-12.
4. Littlewood, Bev, and Lorenzo Strigini, eds. "Validation of Ultrahigh Dependability for Software-Based Systems." CACM 1993, 36(11): 69-80.
5. Butler, Ricky W., and George B. Finelli, eds. "The Infeasibility of Quantifying the Reliability of Life-Critical Real-Time Software." IEEE Transactions on Software Engineering 19(1): 3-12.
6. Littlewood, B. "Limits to Evaluation of Software Dependability." In Software Reliability and Metrics. Proceedings of Seventh Annual CSR Conference, Garmisch-Partenkirchen.
7. RTCA-EUROCAE. Software Considerations in Airborne Systems and Equipment Certification. DO-178B/ED-12B. 1992.
8. Quoted in Andrew Hodges. Alan Turing: The Enigma. Walker & Co., ISBN 0-802-77580-2.
9. Motor Industry Research Association. Guidance for the Use of the C Language in Vehicle-Based Software. Apr. 1998, <www.misra.org.uk>.
10. Yu, Weider D. "A Software Fault Prevention Approach in Coding and Root Cause Analysis." Bell Labs Technical Journal Apr.-June 1998.
11. Finnie, Gavin, et al. "SPARK - The SPADE Ada Kernel." 3.3 ed. Praxis Critical Systems, 1997, <www.sparkada.com>.
12. Finnie, Gavin, et al. "SPARK 95 - The SPADE Ada 95 Kernel." Praxis Critical Systems, 1999, <www.sparkada. com>.
13. Barnes, John. High Integrity Ada - the SPARK Approach. Addison Wesley Longman, ISBN 0-201-17517-7.
14. Sutton, James, and Bernard Carré, eds. "Ada, the Cheapest Way to Build a Line of Business." 1994.
15. Sutton, James and Bernard Carré, eds. "Achieving High Integrity at Low Cost: A Constructive Approach." 1995.
16. Croxford, Martin, and James Sutton, eds. "Breaking through the V&V Bottleneck." Lecture Notes in Computer Science Volume 1031. Springer-Verlag 1996.
17. Software Productivity Consortium <www.software.org>.
18. Parnas, David L. "Inspection of Safety-Critical Software Using Program-Function Tables." IFIP Congress Vol. 3:270-277, 1994.
19. King, Hammond, Chapman, and Pryor, eds. "Is Proof More Cost-Effective than Testing?" IEEE Transaction on Software Engineering Vol. 26 No. 8:675-686.
20. Chapman and Dewar. "Reengineering a Safety-Critical Application Using SPARK 95 and GNORT." Lecture Notes in Computer Science Vol. 1622, Gonzales, Puente, eds. Springer-Verlag, Ada Europe 1999.
21. Bergeretti and Carré. "Information-Flow and Data-Flow Analysis of While-Programs." ACM Transactions on Programming Languages and Systems, 1985.

Notes

1. DO-178B (ED-12B in Europe) is the prevalent standard against which civil avionics software is certified. Level A software is defined as software whose failure would prevent continued safe flight or landing.
2. Static analyzers are tools that determine various properties of a program by inspecting its source code; unlike dynamic testing the program is not compiled and run.
3. The SPARK programming language is not sponsored by or affiliated with SPARC International Inc. and is not based on SPARC™ architecture.
4. SPARK is a programming language; the SPARK Examiner is the static analysis tool used to analyze SPARK programs.

About the Author

Peter Amey is a principal consultant with Praxis Critical Systems. An aeronautical engineer, he migrated to the software world when most of the interesting parts of aircraft started hiding in black boxes. Before joining Praxis, Amey served in the Royal Air Force and worked at DERA Boscombe Down on the certification of aircraft systems. He is a principal author of the SPARK Examiner tool and has wide experience applying SPARK to critical system development. Praxis Critical Systems will be exhibiting and presenting at the STC 2002.

Praxis Critical Systems
20, Manvers Street
Bath, BA1 1PX, UK
Phone: +44 (0) 1225 466991
Fax: +44 (0) 1225 469006
E-mail: peter.amey@praxis-cs.co.uk