The SPARK Programming Language

Johan Olofsson

May 22, 2011

Abstract

SPARK is a refinement and addition to the Ada programming language designed for critical systems. By removing ambiguous parts of the Ada language and adding annotations automated tools can be used to verify the soundness of software. SPARK thus closes the semantic gap between formal specifications and executable code. The result is fewer software faults making it through to dynamic testing and certification, during which time repairs are most costly.

1 Critical Systems

Some systems used in aircraft, nuclear power plants or medical equipment could cause injury or even death to humans if they failed. It is not sufficient for the developers themselves to be confident in their product, they must also convince regulators that the product is adequate. As such verification and validation(V&V) tends to constitute more than half of the total expenditure for critical systems. Even where human lives are not at risk, the cost of a system failure can be so great as to justify costly verification.[1]

2 Ada

In the late 1970's the United States Department of Defense identified their use of 450 programming languages for various projects as a source of excessive costs and as an obstacle for improvement[2]. A process was started to find a high level programming language for their own systems development. What later came to be known as the Ada programming language[3] had among their main requirements software reliability, simplicity and maintainability.[4]

Another requirement was for tools to help programmers find, understand and correct errors. Ada is very stringent about code being written correctly and issues warnings if for instance a variable is declared but never used. As the language was to be used by the entire Department of Defense there had to be a certain level of generality as well. Standard Ada contains libraries for network communication, file access and other common activities.

3 Formal Methods

Software development using formal methods involves the creation of a mathematically based model of the desired system. The model can then be analyzed to verify

that it is consistent and correct, and turn it into a formal specification. Such specifications tend to use languages like Z or VDM. Sometimes this is the extent to which formal methods are used in a software development process. The benefits to this is that is reduces the scope for misunderstandings, of which there tends to be many in software development.

It is also possible to continue and refine the formal specification into executable software, making sure that every step in the refinement process is logically consistent with the previous step. Using formal methods tends to be expensive during the early stages but the costs for verification and validation may well be lower. Formal methods are rarely used even for critical systems but the United Kingdom's Ministry of Defense requires formal methods to be used for safety-critical software.[5][1]

4 SPARK

4.1 Rationale

A research project at the University of Southampton set out to create a programming language for situations where "[t]he integrity of the software is vital: it must be verifiable." The language should preferably be familiar to some developers already and existing compilers that could be trusted would also be beneficial. Ada had been used successfully in the domain of critical systems, both for military and civilian purposes[6], but was not in its entirety suitable for the project because of excessive complexity and loose definitions.[7]

4.2 An Ada subset

A subset of Ada was determined to be a good starting point. Several subsets of Ada meant for critical systems had already been defined.[8] SPARK eliminates generics, derived types, recursion and a number of other features or patterns. Some of the exclusions have been made because they complicate static analysis and not necessarily because they are unsafe. While not all Ada programs are valid SPARK programs the reverse is true, all SPARK programs are valid Ada programs.[7]

4.3 Annotations

The most obvious part of a SPARK program is not the features of Ada that have been removed but the annotations that have been added. They are included as specially formatted Ada-style comments and so have no effect on compilation. A set of tools read the annotations and analyze whether the program does what the developer has described and if the descriptions are coherent. If one procedure assumes that it is never fed a negative number and a calling piece of code has not been annotated to rule out the possibility of a non-negative number being passed to the first, the Examiner will highlight this inconsistency.

Ada already issues a warning if a variable is declared but never used. SPARK goes a step further and alerts the developer if a variable *might* go unused and how this may affect other variables. While a developer working on a critical system should aspire to a higher degree of expertise, someone poorly versed in Ada(i.e. me) can learn how to make crude use of SPARK annotations within a few hours. An example:

```
procedure Power_Up(power : in Integer);
   --# global out power_level;
   --#        out is_active;
   --# derives power_level, is_active from power;
```

The procedure is allowed to modify two global variables, `power_level` and `is_active`(because they are labeled `out`), and may modify them based on the value of the parameter `power`. Designating `power` as an `in`-parameter is an Ada construct, not a SPARK annotation, and means that `power` may not be modified within the procedure.

5 Tools

The main tool for SPARK is the Examiner, which can perform a variety of functions as listed below.

- Checking of SPARK language syntactic and static semantic rules
- Data flow analysis
- Formal program verification
- Proof of absence of run-time errors

There are supporting tools such as the SPADE Automatic Simplifier(SAS) and the Proof Checker. Both deal with Verification Conditions(VCs) from the Examiner with the SAS trying to prove the assertions made about the program, handing over unprovable conditions to the Proof Checker which queries the developer for information.

For this case study AdaCore GNAT Programming Studio GPL Edition was used to investigate the language and tools. The provider AdaCore offers no guarantee that the GPL version is suitable for critical systems development. Instead they offer commercial products like SPARK Pro and GNAT Pro High-Integrity Edition, for which AdaCore does guarantee suitability for critical systems and that have been certified for use in safety and security critical systems.[9]

6 Benefits and Uses

6.1 Lockheed C130J

Based on an article from CrossTalk Magazine[10]

The American aerospace company Lockheed decided to update their C130 transport aircraft to a new version C130J. A large part of the update consisted of new avionics and by extension new software. Some code was developed by subcontractors while the primary mission software was developed by Lockheed themselves. A lot of effort was invested in capturing requirements and deriving from them semi-formal specifications.

SPARK was chosen as the programming language for the in-house development process which helped to clarify specifications that were not sufficiently precise to be translated into SPARK annotations. It was during V&V that Lockheed saw clear

cost reductions. According to Lockheed very few errors turned up during the US Federal Aviation Administration testing and the total cost was less than one fifth of costs normally incurred by aerospace companies during FAA testing. The cost of development for the main control software was claimed to be half of the normal cost for non-critical code!

SPARK can not be given total credit for Lockheed's success as they used other software engineering techniques to improve requirements capture and repeatability. None the less SPARK was a major contribution to the development effort, not least by forcing developers to identify unclear parts of a specification and improve them. SPARK also made it considerably easier to provide the formal proofs required by the United Kingdom's Ministry of Defence during their testing. Said test further more showed that code written in SPARK had only one tenth as many undetected errors as code written in standard Ada. That is an impressive find even though the MoD probably made no efforts to prove that SPARK and regular Ada code had been developed using the same process.

6.2 Security-Critical Systems

Interest has also been shown for using SPARK in security-critical systems. The NSA commissioned a research project in which Altran Praxis - the company developing SPARK - was to redevelop a core part of an existing system called the Tokeneer Identification System. By all accounts the project succeeded, producing a system meeting almost all of the requirements for Common Criteria EAL5.[11] Virtually all the artifacts produced during the project have been made publicly available but the NSA's own analysis is more of a secret. The only feedback is that two faults have been found in the system after delivery. With almost 10 000 lines of code this equates to 0.2 bugs per 1000 lines of code.[12] Admittedly, a comparison of development costs between the Tokeneer project and the original system's development would have been more useful.

7 Drawbacks

Although SPARK seems to have proven its worth in the field of critical systems, it is not a great fit for all purposes. The rationale for SPARK explicitly distances itself from languages like C that are aimed at "convenience of use and efficiency for low-level systems programming".[7] Because the language has been stripped of constructs that are known to be error-prone or that complicate static analysis it has lost some of its expressive power.

The programming community as a whole has shown increasing interest in loosely typed languages like Python and Ruby[14], whereas SPARK can be seen as one with extremely hard typing. Arguments can be made as to whether SPARK is flexible or not. With many more declarations and constraints in a program, SPARK increases the workload when changes need to be made. On the other hand the exact nature of SPARK and its annotations makes it easier to spot mistakes made during refactoring.

8 Conclusions

SPARK can be used to move the bulk of fault removal from the testing phase to specification and development. Testing is always necessary but catching faults early is considerably cheaper than finding them late. Proofs of software satisfying specifications can be obtained automatically, with few exceptions where humans must carry out the proof. This can considerably ease the effort required to acquire certification for critical operations. Indeed the success of SPARK in critical systems development has been shown by both less objective parties(like SPARK developers Altran Praxis) and more objective parties(like Lockheed). Some accounts - notably those of Lockheed - imply that the use of SPARK outside critical systems could be cost-effective, but that is yet to be tried.

References

[1] Software Engineering(9th Edition)
Ian Sommerville

[2] http://archive.adaic.com/pol-hist/history/holwg-93/holwg-93.htm

[3] Ada 95 Reference Manual

[4] Steelman requirements document
http://archive.adaic.com/docs/reports/steelman/steelman.htm

[5] Ministry of Defence - Interim Standard 00-55
http://www.dstan.mod.uk/standards/defstans/00/055/01000200.pdf

[6] http://www.seas.gwu.edu/~mfeldman/ada-project-summary.html

[7] SPARK95 - The SPADE Ada 95 Kernel
http://www.altran-praxis.com/downloads/SPARK/technicalReferences/SPARK95_RavenSPARK.pdf

[8] Ada: towards maturity
Lawrence Collingbourne

[9] http://libre.adacore.com/libre/comparisonchart/
http://www.adacore.com/home/products/sparkpro/

[10] Correctness By Construction: Better Can Also Be Cheaper
Peter Amey

[11] http://www.commoncriteriaportal.org/files/ccfiles/CCPART3V3.1R3.pdf

[12] http://www.adacore.com/home/products/sparkpro/tokeneer/

[13] http://en.wikipedia.org/wiki/SPARK_(programming_language)

[14] TIOBE programming community index - May 2011
http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html