SPARK 2014: A Language for Safety *and* Security

F L F Schanda, **S R Matthews**†*

*Altran UK Limited, florian.schanda@altran.com, †Altran UK Limited, stuart.matthews@altran.com

Keywords: SPARK, safety, security, static analysis, formal methods

Abstract

Usually systems are engineered with either safety properties in mind, or with security considerations. However in today's increasingly connected world, safety-critical systems often also have security requirements. We present SPARK 2014, a language which addresses the verification of safety and security requirements simultaneously and cost-effectively.

1 Introduction

Historically, safety-related systems have tended to be developed only to safety-related standards but in the future will increasingly be required – either explicitly or implicitly – to also exhibit high levels of security assurance.

A good example can be found in modern cars, which have an increasing number of safety-critical control systems. However, these systems can be vulnerable to attacks via both internal and external communication channels, and thus traditional security requirements of networked systems also apply to support its safe operation.

SPARK 2014 is a programming language that can be used to address both concerns simultaneously. In this paper we:

- briefly introduce SPARK 2014 and how it has evolved from SPARK 2005, and
- show how proving safety properties also automatically demonstrates key security properties.

2 SPARK 2014

In order to introduce SPARK, it is useful to first introduce contract-based programming – or Design by Contract ™ [1]. Many languages already contain assertions — optional statements that a programmer includes to check a property about the program state. Contract-based programming extends this idea and introduces more detailed specification (or contracts) on subprogram boundaries. The two most widely used contracts are pre- and post-conditions, but other common contracts can describe frame-conditions[1], loop variants and invariants, and type and object invariants. Many languages designed for formal verification follow this principle and provide support for contracts, such as Eiffel [2] (where the concept originates from), JML [3], and many others.

SPARK has always been both a language and a static analysis tool-set. The language itself is a verifiable subset of Ada, that has been extended with a contract language. In previous versions of SPARK this contract language was embedded in special comments. Ada 2012 introduced a contract language inspired by SPARK, and SPARK 2014 uses this language instead. Figure 1 shows an example procedure specification that has been annotated with a key safety requirement (the lift doors must not be open if the lift moves). The postcondition contract specifies that once the procedure returns we will have moved to the given floor. Note that contracts are often partial specifications - they must be fulfilled, but they do not necessarily describe everything, only the properties of interest.

```
procedure Move_Lift (Target : Floor_Id)
with
   Pre  => not Is_Door_Open,
   Post => Current_Floor = Target;
```

Fig. 1: Example SPARK 2014 procedure specification with a contract.

In 2009 the SPARK tools were released under a FOSS[2] license (GPL3), which encouraged academic collaboration resulting in two tools that made their way into SPARK: Victor [4] a new proof tool and Riposte [5] a counter-example generator. It has also encouraged collaboration from industry, in particular the Isabelle/HOL integration [6].

In April 2014, the SPARK 2014 language and tools [7] were released, with the following key differences:

- the language is a much bigger subset of Ada,
- contracts are no longer special comments but part of the language and can be executed if desired,
- the static analysis tools have been re-written to take advantage of recent advances in static analysis and automatic theorem proving.

[1] which variables a subprogram may change and may not change

[2] Free and Open-Source Software

The contract language of SPARK 2014 can be used to express either safety properties (an example can seen in Figure 1) or security properties (for instance requiring the parameter of a function to be in a certain range in order to avoid a buffer overflow). SPARK contracts can be split into two groups: data- and information flow contracts, and functional correctness contracts.

2.1 Contract language

Two flow contracts exist currently: a global contract that can be used to announce all global variables a subprogram may use and a depends contract that can be used to verify that one set of data does not leak directly or indirectly into another. Below both contracts are shown in Figure 2. The global contract here

```
procedure Swap (A, B : in out Data_T)
with
   Global  => null,
   Depends => (A => B,
               B => A);
```

Fig. 2: Dataflow contracts in SPARK 2014

guarantees that this swap procedure does not read or write any global variables. The depends contract specifies that the final value of A contains only information derived from B (and in particular no information from the original A has any effect on its final value), and vice versa.

In Figure 1 we have already seen the two most important contracts for functional correctness, pre- and post-conditions. SPARK also supports loop invariants (used to reason about the behaviour of loops), loop variants (used to prove termination of loops) and general asserts (used to check any property, both safety and security, at any point in the program).

As indicated above a key difference between previous versions of SPARK and SPARK 2014 is that the contracts can be executed. This is helpful as it allows proof and test to be combined: you prove what is easy to prove and test the rest; the contracts marry-together the two usually separate worlds of static and dynamic verification [8].

2.2 Expressiveness

An expressive contract language is important, as it allows the programmer to express specifications easily and naturally. It also means properties from different domains can be expressed in the same language.

As the SPARK 2014 contract language is executable Ada (or, more precisely, SPARK's subset of Ada), it is already quite flexible; with the additional benefit that the programmer only has to learn one set of syntax and semantics. It is further extended through constructs that are not available in normal program text: ghost functions and variables, and data-flow contracts.

To express complex conditions Ada 2012 already features quantifiers, conditional expressions and case expressions - although all of these can be used in both contracts and ordinary program text they are intended for use in contracts. SPARK 2014 also improves expressiveness in relation to record and array updates. For example an array updated at positions 2 and 4 (with all other fields untouched) used to be difficult to specify in contracts as it is more natural to express this in a sequence. So you had to either involve a quantifier or explicitly mention all other fields. To avoid this SPARK 2014 introduces update expressions, an example can be seen in Figure 3.

```
Post => A = B'Update (2 => 42,
                      3 => F (X))
```

Fig. 3: Update expression in SPARK 2014

As this feature can be useful for ordinary program text, it will be considered for inclusion in a future version of Ada.

2.3 Architecture

The SPARK static analysis tools first translate source code to the intermediate language WhyML [9], from which verification conditions (VCs) are generated for a number of SMT solvers such as Alt-Ergo [10] and CVC4 [11]. Discharging all VCs guarantees absence of all run-time exceptions (no arithmetic over- or underflow, buffer overflows, etc.), and that user-specified contracts (such as pre- or post-conditions and loop variants) hold. The results are then reported back to the user in an IDE (such as GPS[3]), or on the command-line - but in all cases by relating the results directly to the source code.

3 Safety *and* security

Traditionally software has been written with either safety *or* security in mind, and different approaches have been used to achieve either verification objective. Although the objectives of safety and security are different, there is significant overlap in some areas, and the same methods can be used to specify either.

3.1 Similar properties

Safety properties and security properties are often surprisingly similar.

For example a safety requirement might be "the robot must never move faster than 1 foot per second". This property can be verified by using an appropriate type, and statically proving that the software can never produce the conditions required to exceed it.

Now consider a common high-level security requirement "information must not leak from one domain into another". A

[3] GNAT Programming Studio

key component in meeting this requirement is that arrays are never accessed outside their given bounds: otherwise unrelated memory might be read - or even worse - modified. However this property is identical to the safety property above, and it requires the same solution.

3.2 Automatically mitigating top CWE issues

All security and safety properties related to type safety and run-time exception freedom are inserted automatically by the SPARK tools so that the user can focus on the remaining interesting properties. This automation also ensures complete coverage, and is the main reason why many common programming errors are identified by the tools.

Using SPARK can help deal with many of the top CWE[4] issues. They fall in broadly two categories - language issues (i.e. buffer overflows), and API issues (i.e. strcpy or SQL injections). In this section we survey the "extended" top-25 issues: these are the top-25 issues plus 16 serious issues that are not as common but have serious impact. Note that interesting real-world examples for each of these can be found on the CWE project's website [12].

3.2.1 Issues mitigated by the SPARK language

There are a number of CWE issues that cannot happen in SPARK programs as the language definition forbids these constructs in the first place. Since as there are no pointers in SPARK (instead we have references and formalized containers), the following issues do not apply to SPARK:

- CWE-822 (Untrusted Pointer Dereference)
- CWE-476 (NULL Pointer Dereference)
- CWE-825 (Expired Pointer Dereference)

3.2.2 Issues mitigated by flow analysis

Flow analysis in the SPARK tools can eliminate the following issues:

- CWE-456 (Missing Initialization) - all variables must be initialized at some point before their first use in SPARK
- CWE-807 (Reliance on Untrusted Inputs in a Security Decision) - flow analysis can be used to ensure information from one variable does not leak directly (i.e. assignments) or indirectly (i.e. control flow) into another[5].

A number of coding standards mitigate the initialization problem by requiring all variables to be set at the point of declaration. Although this is easy to check with a tool, it is not actually that helpful: there might still be a path in the code that does not set the variable and its programmer-chosen default value is used. For some types with strong invariants or complex constructors it may not even be possible to construct a default or null value. SPARK helps here because the coding standard does not have to mandate initialization-at-declaration, and hence you can initialize variables at the appropriate points.

3.2.3 Issues mitigated by proof

Proof of absence of run-time exceptions will deal with a large number of CWE issues. In particular the following issues from the extended top-25 list can be entirely eliminated by SPARK:

- CWE-120 (Buffer Copy without Checking Size of Input)
- CWE-131 (Incorrect Calculation of Buffer Size)
- CWE-190 (Integer Overflow or Wraparound)
- CWE-129 (Improper Validation of Array Index)
- CWE-805 (Buffer Access with Incorrect Length Value)
- CWE-681 (Incorrect Conversion between Numeric Types)
- CWE-754 (Improper Check for Unusual or Exceptional Conditions)

The Ada and SPARK reference manuals clearly identify each point where run-time exception might be raised, including (but not limited to) all of the above list. The SPARK tools automatically generate verification conditions, which, when proven, guarantees that run-time exceptions never occur regardless of input.

As an aside, it should be noted that completing proof of absence of run-time exceptions can be used as justification for *safely* turning off all run-time checks; in which case performance can be comparable to a C implementation [13].

3.2.4 Issues mitigated by strong typing

A common defect, which is difficult to detect in many languages, is the confusion of different data items that just happen to share a common underlying program data type. A good real-world example of this is confusion of thrust expressed in lb-sec and not Newton-sec, which ultimately led to the failure of the Mars Climate Orbiter [14]. Although both types are floating-point types, they should obviously not be confused.

Although it is not immediately obvious, this same problem arises in a more general form in information security: the number one top issue in the CWE project is CWE-89 (SQL Injection), and there are many related issues (such as CWE-78 "OS Command Injection" and CWE-79 "Cross-site Scripting"). These generally relate to a common data type (strings in this case) being passed from the outside world into a query

[4] Common Weakness Enumeration

[5] Also see Section 5 for a planned improvement in this area

or instruction using the same data type. For example, a user-supplied string might be used naïvely in the query string fragment `WHERE Name=%s`. If the user enters a valid name such as `ada_lovelace` then the resulting query is OK:

```
WHERE Name=ada_lovelace
```

However, if a malicious user enters the user name as `ada_lovelace OR 1=1` then this will cause a potential security breach as it changes the meaning of the query and now *all* entries are returned:

```
WHERE Name=ada_lovelace OR 1=1
```

Strong typing and a well-designed API can be used to *mitigate* issues like this, by helping the tools to detect incorrect use. For example, you might create a "Raw_Data" type and "Sanitized_Data" type:

```
type Raw_Data       is new String;
type Sanitized_Data is new String;
```

Although they are both strings in our example, they are incompatible in Ada; so it would be more difficult to accidentally assemble a problematic query without calling some kind of escaping function:

```
Query    : Sanitized_Data;
Username : Raw_Data := Get_Username;

Query := "WHERE User=" & Username;
-- illegal: type mismatch
```

For example the above program fragment is illegal and any Ada compiler is required to raise an error, as the string concatenation operator may not be used to concatenate two different types of string, reminding the user to use an escaping function to sanitize the user name.

4 Related work and applications

SPARK 2014, the programming language, draws inspiration from a number of comparable languages intended for formal verification. Most obviously from the previous generations of SPARK and from Ada 2012, but also from languages such as Eiffel [2].

Existing publications such as [15, 16] have compared the safety and security inherent in these earlier versions of SPARK with other commonly-used languages (i.e. C, C++, Java). Although no comparison has yet been published for SPARK 2014 it has been designed to support the same verification objectives as SPARK but for a larger subset of the Ada language.

The advantages of the contract-based approach to software verification embodied in SPARK 2014 and its predecessors are demonstrated by a series of successful applications [17, 18]. Some of the key reference applications and their implications are discussed below.

4.1 SHOLIS

SHOLIS, the *Ship/Helicopter Operational Limits Instrumentation System*, is a ship-borne embedded system which aids the safe operation of helicopters on naval vessels. SHOLIS achieves this by advising on current conditions — the wind vector and the ship's roll and pitch — relative to a defined safety envelope for particular operations such as landing or in-air refuelling.

The SHOLIS software, developed using SPARK was the first system to be developed to the requirements of the UK Ministry of Defense Interim Defense Standard 00-55 for safety-critical software. Part of the SHOLIS application was assessed to be at the highest safety integrity level, SIL 4, which implied rigorous requirements on the software development and verification process including use of a formal specification, functional proof of the source code against the specification, and information flow analysis.

The development of the SHOLIS application in SPARK demonstrated that the use of formal methods and program proof (both functional correctness and exception-freedom) was a practicable in an industrial context. Moreover, this project demonstrated that verification based on formal proof can by significantly more cost-effective at finding faults than a traditional test-centred approach to functional verification [19].

4.2 C130J

The C130J is the most recent generation of Lockheed-Martin's *Hercules* military transport aircraft. The Operational Flight Plan (OFP) software for the C130J was written in SPARK to meet the requirements of the avionics standard DO-178B and additional formal verification requirements imposed by the UK's Royal Air Force.

Both information flow dependencies and low-level functional requirements for the C130J OFP were captured using SPARK contracts. Conformance of the source code to the contracts was automatically checked using supporting static analysis tools.

The C130J project demonstrated one of the key advantages of using the contract-based programming approach of SPARK [20]. Because errors are detected early the source code passed from development to verification phases has an unusually low defect density. This low defect density results in greatly reduced time spent in the "debug cycle" where code is passed repeatedly between development and test, with consequent overall cost reductions reported for the project.

4.3 Tokeneer and secure applications

The Tokeneer ID Station (TIS) protects a secure enclave using a biometric access control. The software within TIS was required to meet evaluation assurance level (EAL) 5 of the Common Criteria.

To meet the stringent objectives of EAL 5 the software was implemented in SPARK [21]. Contracts were used to express

information flow dependencies and key security properties of the system.

Tokeneer was commissioned as a study by the US National Security Agency (NSA) to demonstrate the effectiveness of SPARK in the secure systems domain. Evidence of this effectiveness is re-enforced by a number of other security-related applications, including:

- IRONSIDES [22] — a secure DNS server written in SPARK, that is verifiably immune to remote code execution exploits and single-packet denial of service attacks.
- Muen [23] — a secure Open Source microkernel written in SPARK that has been formally proven to contain no run-time errors at the source code level. Muen provides an execution environment for components that exclusively communicate according to a given security policy and are otherwise strictly isolated from each other.
- SPARK-BigNum [6] — a SPARK implementation of a maths library for use in RSA encryption/decryption. This project demonstrates the effective application of an alternative proof system (using Isabelle/HOL) to the verification of SPARK programs.
- MULTOS CA [24] — a certification authority for the MULTOS operating system designed for use on smart cards. SPARK was used for the system at the heart of the MULTOS CA that issues digital certificates and meets the requirements of level E6 — the highest assurance level of the UK ITSEC scheme.

5 Conclusions and further work

We have seen that there is overlap between many common security and safety properties, and that the contract language of SPARK 2014 can be used to specify both. The automatic specification for absence of run-time exceptions, flow analysis and language restrictions mitigate many of the extended top-25 CWE issues.

The SPARK 2014 language is still evolving. Work is planned or underway in the following areas:

- Larger language subset - in particular OO and tasking. A useful subset of Ada's tagged types, interfaces and dynamic dispatch will be supported by the language and tools. The proposed approach will follow the Liskov substitution principle (LSP [25]) and will support both class-wide and regular contracts. For concurrency, Ada already specifies a suitable restricted profile for tasking (Ravenscar [26]), which would be an good first step. The motivation for this is to extend the safety and security verification to a wider class of programs and systems.
- Bit-precise reasoning for IEEE-754 floats - the current model used by SPARK 2014 is sound, but not complete. SMT-LIB has recently been extended with a theory of floating points [27] and experiments have been very promising. This is important as floating point arithmetic is being more widely used in critical control systems.
- Domain separation for information flow - this allows users to tag variables and abstract state with a domain and flow analysis will enforce some global policy. For example a security property might state that information from the "UNCLASSIFIED" domain might flow into "SECRET" domain, but not the other way. Subprograms (for example ciphers) could also be annotated with de-classification aspects, allowing them to locally break these policies.

References

[1] Bertrand Meyer. Design by contract. Technical Report TR-EI-12/CO, Interactive Software Engineering Inc., 1986.

[2] Bertrand Meyer. Eiffel: A language for software engineering. Technical Report TR-CS-85-19, University of California, 1985.

[3] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: a Java Modeling Language. In *Formal Underpinnings of Java Workshop (at OOPSLA'98)*, October 1998.

[4] Paul B Jackson and Grant Olney Passmore. Proving SPARK verification conditions with smt solvers. Technical report, Technical Report, University of Edinburgh, 2009.

[5] Florian Schanda and Martin Brain. Using answer set programming in the development of verified software. In *ICLP (Technical Communications)*, pages 72–85, 2012.

[6] Stefan Berghofer. Verification of dependable software using SPARK and Isabelle. In *SSV*, pages 15–31, 2011.

[7] Altran and AdaCore. SPARK 2014, April 2014. www.spark-2014.org.

[8] Johannes Kanig, Roderick Chapman, Cyrille Comar, Jerôme Guitton, Yannick Moy, and Emyr Rees. Explicit assumptions - a prenup for marrying static and dynamic program verification. In *8th International Conference on Tests & Proofs*, July 2014.

[9] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3: Shepherd your herd of provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, pages 53–64, Wrocław, Poland, August 2011.

[10] Sylvain Conchon, Evelyne Contejean, and Johannes Kanig. Ergo: A theorem prover for polymorphic first-order logic modulo theories, 2006.

[11] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *Computer aided verification*, pages 171–177. Springer, 2011.

[12] MITRE. CWE/SANS Top 25 most dangerous software errors. http://cwe.mitre.org/top25, 2011.

[13] Roderick Chapman, Eric Botcazou, and Angela Wallenburg. SPARKSkein: a formal and fast reference implementation of skein. In *Formal Methods, Foundations and Applications*, pages 16–27. Springer, 2011.

[14] Arthur G Stephenson, Daniel R Mulville, Frank H Bauer, Greg A Dukeman, Peter Norvig, LS LaPiana, PJ Rutledge, D Folta, and R Sackheim. Mars climate orbiter mishap investigation board phase I report. *NASA, Washington, DC*, 1999.

[15] Andy German. Software static code analysis lessons learned. *Crosstalk*, 16(11), 2003.

[16] Paul E Black, Michael Kass, Michael Koo, and Elizabeth Fong. Source code security analysis tool functional specification version 1.1. *NIST Special Publication*, 2011.

[17] Ian O'Neill. Spark – a language and tool-set for high-integrity software development. In Jean-Louis Boulanger, editor, *Industrial Use of Formal Methods*, pages 1–27. Wiley, 2012.

[18] Florian Schanda and Roderick Chapman. Are we there yet? 20 years of industrial theorem proving with SPARK. In Gerwin Klein and Ruben Gamboa, editors, *Interactive Theorem Proving*. LNCS, July 2014.

[19] Steve King, Jonathan Hammond, Roderick Chapman, and Andy Pryor. Is proof more cost-effective than testing? *Software Engineering, IEEE Transactions on*, 26(8):675–686, 2000.

[20] Martin Croxford and James Sutton. Breaking through the v and v bottleneck. In *Ada Europe 1995*, volume 1031. LNCS, 1996.

[21] Janet Barnes, Roderick Chapman, Randy Johnson, James Widmaier, David Cooper, and B Everett. Engineering the Tokeneer enclave protection software. In *1st IEEE International Symposium on Secure Software Engineering*, March 2006.

[22] Martin C Carlisle and Barry S Fagin. IRONSIDES: DNS with no single-packet denial of service or remote code execution vulnerabilities. In *GLOBECOM*, pages 839–844, 2012.

[23] Reto Buerki and Adrian-Ken Rueegsegger. The Muen separation kernel. http://muen.codelabs.ch, 2013.

[24] Anthony Hall and Roderick Chapman. Correctness by construction: developing a commercial secure system. *IEEE Software*, pages 18–25, January–February 2002.

[25] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, November 1994.

[26] Alan Burns. The ravenscar profile. *ACM SIGAda Ada Letters*, 19(4):49–52, 1999.

[27] Martin Brain, Cesare Tinelli, Philipp Rümmer, and Thomas Wahl. An automatable formal semantics for IEEE-754 floating-point arithmetic. 2014.