SPARK toolkit - an overview

[Correctness by Construction] *

Kim Rostgaard Christensen
Technical University of Denmark
s084283@student.dtu.dk

ABSTRACT

It is desirable to be able to build software without run-time errors in safety-critical applications. Using the SPARK static analysis tools, this can be asserted at compile-time.
SPARK turns conventional verification and validation approaches upside down and seeks to find a proof of correct software instead of trying to find defects run-time by debugging.

1. INTRODUCTION

Being able to provide guarantees for the correctness of software is not goal easily achieved. Software is dynamic in nature and proving a static behaviour can be near impossible, using conventional development. Software is not subject to conventional MTBF[1] models - such as those used to model hardware failures, as it fails very different. Providing a guarantee for the behaviour and correctness of software is usually done by manual code reviews and extensive testing, verifying the implementation according to specification and in effect increasing reliability and availability of the entire system.

This report gives an overview of the Correctness by Construction approach to software engineering. This approach requires a full up-front formal specification done prior to implementation and that it is written in formal unambiguous language. The process of verification is then passed to the an automated tool that gives proof of correctness - no longer subject to human error. A requirement of significant automation of proof can also be a by-product of a large code base.

This report also gives an explanation of Design-by-contract principals, an introduction to the SPARK language and the SPARK toolset. The report is concluded by a brief introduction to the Ravenscar profile with RavenSPARK and a discussion.

Throughout the report the terms ”contract” and ”specification” will be used interchangeable.
This report focuses on the SPARK 95 language specification.
A defect is defined as an faulty software implementation, leading to an either propagated or direct error in the system it is implemented in.

SPARK is developed by Altran Praxis.

2. CORRECTNESS BY CONSTRUCTION

It is argued in [1] that verification and validation account for as much as 80 percent of the development of safety-critical applications. By employing a technique called Correctness by Construction, it is possible to reduce the time spent in the most expensive part of the development phase; the testing/-validation phase, thus meeting project deadlines and lowering production cost.

Correctness by Construction is a philosophy manifested in the SPARK programming language and toolset. It seeks to expand the the use of tools in order to create automated proof that the application is indeed the application intended. The two primary goals is specified as following[5].

- Deliver low-defect rate software in a cost-effective manner
- Deliver software resilient to change throughout its lifecycle

This sounds like a good deal for every party involved in the development process. But how is it realized?
Correctness by Construction argues that verification should be done as early as possible. The application should be broken down to as many sub-components as possible, verifying every part individually, before moving on to the next component.
By the use of a formal specification language to capture the requirements, it is possible to define the behaviour of the application - even prior to implementation. This enables tool-supported analysis of the implementation. This can reveal potential erroneous behaviour in the implementation, and in some cases also flaws in the specification itself. The

*This report should also be available online at `http://retrospekt.dk/files/spark_overview.pdf`
[1]Mean Time Between Failures

approach of elaborating the specification is called contract programming - or Design-by-contract

2.1 Design-by-contract

Design-by-contract is a method of elaborating the behaviour of an application via a formal language. It works by adding additional information to the components of the application. This design information serves as the guideline for the implementation, and is similar to a legal contact - hence the name. All contract information is considered redundant from the compilers point of view.
Contracts work on method/procedure/function or module level, and typically supply the following information:

- Context
- Side-effects
- Pre- and postconditions

A contract could also hold information about other guarantees, for example about time or space used. Or about errors and exception conditions that could occur (not applicable in SPARK, though)
Design-by-contract specifies a set of assertions that must hold. These are typically preconditions for what must be true prior to calling a procedure, and correspondingly postconditions that specifies what must hold after the call.
A very nice property of Design-by-contract, is that it gives the development team a chance to reflect on the application components and interfaces prior to implementation. This alone can identify many potential defects.
When the implementation is done, it is possible to verify that it is in accordance with the contract by the use of automated tools.

3. SPARK PROGRAMMING LANGUAGE

The SPARK programming language consists of a set of tools and a programming language. It can be discussed whether the SPARK language is in fact a language for itself, as is is a subset of the Ada3.1 programming language with the addition of contract information in the form of Ada comments.

3.1 Ada programming language

Ada is a strongly typed, ISO-standardized[2] general purpose programming language. It is originally designed for usage in embedded applications and real-time systems. It has runtime checks and supports exceptions. The language is also designed to be as human readable as possible, as code readability usually contributes a great deal to code maintainability.
A simple hello world application is shown below.

```
1 with Ada.Text_IO;
2
3 procedure Hello_World is
4 begin
5     Ada.Text_IO.Put_Line("Hello␣World");
6 end Hello_World;
```

[2]ISO 8652

The Ada language has the potential of splitting up the specification and implementation into seperate compilation units enabling abstraction from concrete implementation when doing initial design.
Ada is used typically in avionics, missile systems, rail systems and financial applications.[3]
A very nice feature in Ada is the ability to define constrained types. Observe the following example:

```
1 type Traffic_Colors is (Red, Yellow, Green);
```

This is very useful tool in translating informal requirements into formal specifications.

3.2 Elimination of dynamic behaviour

In order to determine whether an application behaves correctly, every time, we need to cut away some of the undesirable functionality of the Ada language. This is all the dynamic features of the language, and constructs that makes proving the software more difficult.
The following feature are no longer available:

- Recursion
- Generics (Templates)
- Concurrency
- Access types
- Functions must be pure

Recursion is not surprisingly prohibited, as it can cause stack overflow.

Generics is similar to templates in C++. It is a dynamic allocation of a generic unit, such as a bounded buffer without the content specified. Exclusion of this puts some constraints on the recyclablility of the program units.

Multi tasking is not allowed in core SPARK, but is available via the Ravenscar profile and the additional annotations specified in RavenSPARK (see section 7.2).
Access types is the Ada equivalent to C pointers with added type safety. But as these pose problems with provability, the support is left out. The use of memory mapped I/O can be achieved by specifying the address of a variable manually instead of using a pointer type.

Exceptions are not supported as they also pose problems in provability. But although SPARK does support them, it does effectively eliminate them if the program is proved correct. This is described in detail in [3].

A pure function is a function that behaves the same way as a mathematical function - for example $cos(x)$. The rule here is that the function must return the same value at arbitrary times, using the same parameters. Thus, it cannot depend on an internal or global state. Pure functions also cannot have any side effects, such as the modification of a global variable or I/O operations.

[3]http://www.adacore.com/home/ada_answers/lookwho/

By specification a pure function is allowed to call another pure function, but not an impure one, as it makes the the function impure indirectly.
In full Ada, functions are not allowed to have parameter of mode out, as the only variable it may output is its return value. It can have side effects though, such as modification of a variable outside its own scope. If there is a need for modifying an internal package state or doing I/O from within a function, a procedure with a parameter of mode out can be used instead.

3.3 SPARK annotation language

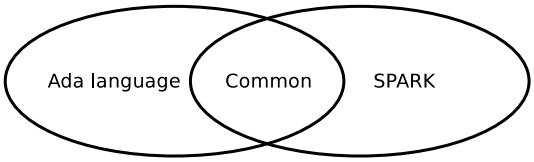

Figure 1: SPARK and Ada relationship[3]

SPARK annotations is embedded in Ada comments, making every SPARK program a valid Ada program without introduction the need for a new compiler. In effect; SPARK cannot introduce new features to Ada, so any restrictions that apply to Ada, also apply to SPARK
As previously noted, to enable static and semantic analysis of the code, more information about the intended operation of the application is needed.
The SPARK annotation language provides this. With the help of some extra keywords, it supplies the developer with the tools needed for describing the code behaviour in contractual terms, describing what the intended operation is, and what the implementation must live up to.

3.4 Examples

A simple example is the one given in [3]. It is a simple add operation, that adds an integer to a global variable Total. In normal Ada - and most other langauges, the procedure can access any global variable from within the body, but is not required to do so.
When no extra information about what the intended operation of Add is, the implementation could be anything. It does not even have too add the X to Total, it can do what ever it wants.
By constraining the procedure and telling it what variables it should access in the specification, it enforces the implementation to follow it. The Add procedure also has a postcondition statement. This enables us the specify what the exit state of the procedure should be. In this case, `Total` should be the initial value (`Total~`) plus the parameter `X`.

```
1 procedure Add(X : in Integer);
2 --# global in out Total;
3 --# post Total = Total~ + X;
```

Similar to postconditions, preconditions can also be specified. This is a very useful way of specifying logical preconditions at entry points. An example is as follows:

```
1 procedure Turn_Traffic_Light_Green
2 --# Pre Intersection_All_Red;
```

The logic is sound in human language, that no traffic lights should turn green until all lights are red. This constraint is not so easily captured in conventional programming languages, and only deep into the implementation by using acceptance tests or language asserts, making traceability difficult and complex.

3.4.1 Acceptance test

The purpose of an acceptance test is to determine if a measurement or calculation is within a reasonable range or domain. For example, an integer addition with two positive operands should never return a negative integer. A good example is from [7] of an acceptance test is visual inspection of a thermometer that reads -10° on a smelting hot summer day.

3.4.2 Run-time asserts

A traditional way of going about software testing is to use run-time asserts. In regular Ada, it is enabled by a compiler pragma.

```
1 pragma Assert (K > 3, "Bad value for K");
```

When asserts are enabled, the compiler translates the above piece of code into the following:

```
1 if not K > 3 then
2     System.Assertions.Raise_Assert_Failure
3                 ("Bad value for K");
4 end if;
```

Notice the linebreak is ignored by the compiler. When asserts are disabled in the compiler, it removes the pragma line, effectively clearing the application of any run-time asserts. Thus they should only be used while testing the application for ensuring values. This is sometimes also referred to as a debug-build.
The purpose of this build is to throw a controlled exception instead of having a null pointer exception or a division by zero or an entirely seemingly unrelated propagated error. But this check is still done at run-time, and for the `Raise_Assert_Failure` to be raised, extensive testing is still needed to make sure that this situation never happens.
When using SPARK, assertions is statically analysed before run-time, and proved not to happen during execution. Preconditions, postconditions and SPARK asserts are all static asserts that has no functional effect on the compiled code, but is only used for analyses.

4. PROVING CORRECTNESS

SPARK examiner performs a static analysis in the order shown in figure 2.

It starts with the subset and static semantics phase where the basic semantics of the language is verified. It also checks if no aliasing occurs.

The next phase is the information flow analysis. This phase determines the flow of information is done within the body of the unit under test. It finds data-flow errors and constructs

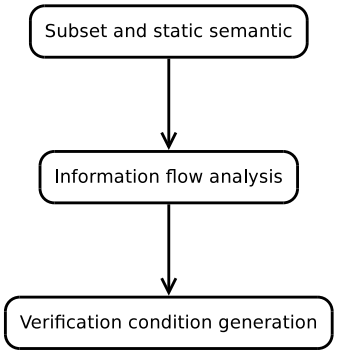

Figure 2: Examiner phases

that does not affect any outputs, and are thus ineffective.

The final phase of the examiner is the verification condition generation phase. This phase generates sets of hypothesis and conclusions for use in automated or manual proof. By moving the programmers testing up the abstraction ladder, letting him/her do proof of a module contrary to doing code reviews provokes a reflection upon the code from a critical stance. Provability is also closer to the object oriented approach adopted in modern development, as is searches to find a proof for the model as a whole.

The examiner produce messages prepended with a three character tag. Table 1 shows the meaning of the different symbols. It should come as no surprise that syntactic or

Symbol	Description
***	Syntactic or semantic error
!!!	Flow analysis error
???	Flow analysis warnings and notes
---	Poor implementation practice
+++	No errors found

Table 1: Spark examiner message symbol matrix

semantic errors leads to the flow analysis not being run.
A flow error can be an ineffective statement in the form of a variable being assigned, but never referenced. A warning can be triggered from a potential code path, that might never be taken, that can lead to a contract disagreement. The examiner also warns if it assumes for example left-to-right evaluation of expressions.

4.1 Information flow analysis

This phase identifies ineffective statements and verifies the flow of information between program modules. It identifies all data-flow errors by making sure that the implementation does indeed implement the contract. A disagreement is considered an error, and will result in the verification conditions not being generated.

4.2 Verification conditions

Upon completion of the static semantics and flow-analysis verification conditions are generated.
Verification conditions consists of a set of hypothesis and a set of conclusions.

Verification conditions is written in a language called a Functional Description Language (FDL) that specifies real and integer values, internal requirements, deducible expressions ans equivalent expression.

There are two levels of verification condition run-time checks. The basic level generates verification conditions for range, division and index check. The next level also generates verification conditions for overflows.
This is basically equivalent to writing the precondition:

```
1  procedure Increment (X : in out Integer);
2  --# pre X < Integer'Last;
3  --# post X = X~ + 1;
```

And the higher level of verification condition generation can be considered a convenience level, freeing the programmer of writing tedious preconditions - and also eliminates a potential human error. This level of verification condition is defined by a command line switch to the examiner.

4.2.1 Code paths

The examiner runs though the different code paths and verifies that all of them is within the contract (specification). This means that if a procedure has one global keyword attached to one variable of mode in, the procedure must read the variable at at least one path. it cannot access (read of write) any other variables, other than the ones specified in the contract.
It is concluded in [6], not surprisingly, that the number of verification conditions is directly related to the number paths though the code. Thus it should be considered good practise to reduce the number of these paths by for example adding extra procedures and functions.

4.3 Running the SPARK examiner

The examiner is, like the other tools, a command-line tool. It is invoked by typing `spark` in a command line window followed by a number of switches and, of course - the files that should be analysed. The examiner will produce a metafile containing the verification conditions. This file is suffixed by `.vcg` and is used as input to the simplifier and proof generator. It also produces a `.fdl` file that is the declarations, and a `.rls` file that are the rules for verification.

As a little note; The SPARK examiner is written in SPARK to ensure that the tool provides correct output.

5. THE SPARK SIMPLIFIER

The simplifier is a Prolog tools that seeks to reduce and discharge the verification conditions. It does this by, among others, the following strategies: It seeks out contradictions. If hypothesises contradict each other, our proof is incomplete. Then it tries to reduce the number of hypothesises by reducing them.
This procedure is the same as the one used in any other logic reduction algorithm.
The simplifier produces a file with the simplified verification conditions suffixed by `.siv`

5.1 The SPARK proof generator

This tool is the final one in the proof chain. It takes the declarations and rules from the examiner and the simplified verification conditions from the simplifier and tries to prove correctness by the use of some more advanced methods.

If proof is not complete at this point; the development team can either go back and try to further annotate the code, or do a proof by hand using the verification conditions and rules generated.

6. SPARK BENEFITS AND CHALLENGES

The following table shows an overview of some projects using SPARK with some metrics added. Code size is represented in lines of code (loc). Productivity is measured in loc per

Project	Year	Size	Productivity	Defects
CDIS	1992	197 kloc	12.7	0.75
SHOLIS	1997	27 kloc	7.0	0.22
MULTOS CA	1999	100 kloc	28.0	0.04
A	2001	39 kloc	11.0	0.05
NSA	2003	10 kloc	38.0	0

Table 2: Comparison of projects using SPARK[5]

day, and defects is measured per kloc. The table shows that a significant reduction in defects is to be gained by using SPARK. This is based around the classic assumption that there is one defect per kloc. It should be noted that all these projects are high-integrity projects. A brief presentation of each follows.
CDIS is a Real-time air traffic information system still in use today (according to Altran Praxis).
SHOLIS is a system developed to UK military of defence. It is an abbreviation of Ship/Helicopter Operating Limits Information System.
MULTOS CA is a certification authority (CA) for smart card operating systems.
A is a managements system for UK military stores.
NSA Is the Tokeneer biometrics system. This source code of the system is available at the AdaCore web page[4].

For a more in-depth discussion of the projects; please consult [2].

7. CONCURRENCY

The Ada language has language-support for tasks. The runtime environment has a built-in dispatcher that uses either fixed *a priory*[5] compile-time priorities, or dynamic run-time priorities. Ada tasks are corresponding to tasks from the scheduling analysis study field, and thus enables an application to be subjected to a schedulability analysis very intuitively.

When concurrency is enabled, there has to be a synchronization mechanism to provide exclusive access to shared resources. This is also supported by the Ada language by the use of compiler pragmas.
When dealing with hard real-time systems having multiple tasks, it is also important that resource access protocols is available. All the requirements that make up hard real-time system is packed into what is called an Ada compiler profile.

7.1 The Ravenscar profile

The Ravenscar profile aims to provide threading in a predictable fashion, such that threaded applications can be deployed in safety-critical systems. The profile is applied to the current Ada 2005 standard tasking subsystem, and dictates a subset of tasking features to be used in programs.
A profile is a common way of specifying the behaviour of a compiled program. The Ravenscar profile leads to the priority ceiling protocol and FIFO dispatching within priority groups to be enabled, among other limitations.[4]

7.2 RavenSPARK

RavenSPARK is an addition to the original SPARK language that introduces new keywords in order to extend the proofs produced by the SPARK tools to include multitasking. The RavenSPARK is a superset of the core SPARK language, and to be able to verify RavenSPARK applications, the examiner must be configured accordingly.

8. DISCUSSION

The SPARK language is a very robust and formal way of going about software development in general. However, it is not this authors personal impression that is widely adopted in safety-critical systems. This section takes a critical view on SPARK and seeks to clarify what the real gains and challenges of the language is.

8.1 Gains

There a number of gains by adopting into the development process of a safety-critical system. Some of the most essential are listed below.

8.1.1 Increase reliability and availability

In an application used in a safety-critical system, it usually important to either emphasize reliability or availability. By producing low defect rate software, both of the factors are enhanced.

8.1.2 Proof is correctness

SPARK is sound - meaning that is produces no false positives. When the proof is complete, the software is correct. That is according to specification, naturally.

8.1.3 Formal and systematic approach to development

When developing safety-critical systems, a lot of requirements are produced by for example a HAZOP analysis. These requirements are unambiguous in nature and can serve as - or be written in the SPARK specification language to improve traceability upon implementation.

8.1.4 Emphasize design

The SPARK approach to software engineering forces you to spend a lot of time on your design. Being that if you do not have a specification of how your application should behave, then you have nothing to verify it against. This provokes a lot of reflection, both in the initial design stages, but also in the later verification stages when provability has to be done either by additional annotations, or by manual proof.

[4]http://www.adacore.com
[5]Know prior to execution

8.1.5 Encourages modularity

By letting smaller units be more easily proven (because of branches), modularity is encouraged. And by effect, this also improves improves maintainability.

8.2 Challenges

There are always two sides of a coin, here is listed some of the potential challenges I have identified.

8.2.1 Steep learning curve

A deep understanding of the software engineering field, the Ada language and the SPARK language is needed to be able to be able to construct a complete proof. If for example the simplifier/proof generator cannot reduce the verification conditions further, proof must be done by inserting additional asserts in the implementation or by hand - which also requires knowledge about constructing proof using FDL.
If the expertise is not present, the development team is in great risk of doing a lot of workaround solutions instead of figuring out what the SPARK-correct way of handling the problem should be. This is also the reason that Altran Praxis recommends that the a certified instructor is hired when SPARK is adopted.

8.2.2 Language constraints

The constraints enforced by the SPARK language itself is an Achilles heel. Leaving it best suited for applications where correctness is of extremely high importance. In some larger projects the tool can only be useful in proving the correctness of minor parts of an application that are static in nature.

8.2.3 Narrow adoption and paradigm shift

Another thing that can be considered a drawback is that it uses the Ada language as base - a language that is not very widely adopted. A change in a development department from for example C to SPARK would lead to a total change in tools, development process, management process and testing phase.
On top of all this, the process of moving to Ada is a paradigm shift and is subject to the pitfalls of this.

8.3 Is SPARK enough?

Clearly using SPARK is a good way of improving the reliability of your entire system, but it is still just a minor part of it. A holistic approach to a safety-critical system must be taken in order to ensure stability. The use of SPARK can be combined with for example SIHFT[6] or a dedicated error-checking hardware to ensure fault-tolerant behaviour.

9. CONCLUSION

Adoption of the Correctness by Construction concept into the development process of a safety-critical system can be a benefit on many levels as mentioned in section 8. The SPARK toolset actually guarantees that the reliability of your system increases, at the cost of additional formal work.

A plunge into the, fairly uncharted, SPARK ocean should also be based on the weighing of what the productivity penalty of a paradigm shift is contra the gains. If project deadlines is often missed, then perhaps Correctness by Construction and SPARK is a good way of bringing more structure into the development process.

SPARK can help the developers increase the integrity of their application and give a stronger sense of certainty that the product they deliver actually lives op to the requirements. The validation output can also serve as certification documentation for certification authorities, and middle-layer document for traceability purposes.

[6]Software Implemented Hardware Fault Tolerance

APPENDIX

This appendix holds a list of references.

A. REFERENCES

[1] P. Amey and C.B. Construction. Better Can Also Be Cheaper, CrossTalk Magazine. *The Journal of Defense Software Engineering*, 2002.

[2] P. Amey and A.J. Hilton. Practical experiences of safety-and security-critical technologies. *ADA USER*, 25(2):98, 2004.

[3] John Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison Wesley, New York, NY, 2003.

[4] John Barnes. Rationale for ada 2005, 2007.

[5] Roderick Chapman. Correctness by construction: a manifesto for high integrity software. In *Proceedings of the 10th Australian workshop on Safety critical systems and software - Volume 55*, SCS '05, pages 43–46, Darlinghurst, Australia, Australia, 2006. Australian Computer Society, Inc.

[6] Darren Foulger and Steve King. Using the spark toolset for showing the absence of run-time errors in safety-critical software. In *Proceedings of the 6th Ade-Europe International Conference Leuven on Reliable Software Technologies*, Ada Europe '01, pages 229–240, London, UK, 2001. Springer-Verlag.

[7] I. Koren and C. M. Krishna. *Fault-Tolerant Systems*. Morgan-Kaufman, San Francisco, CA, 2007.