A (Very) Short Introduction to SPARK: Language, Toolset, Projects, Formal Methods & Certification

Eduardo Brito

CCTC / Departamento de Informática
Universidade do Minho
Campus de Gualtar, 4710-057 Braga, Portugal
`edbrito@di.uminho.pt`

Abstract. Guidelines for the development of software in safety-critical systems usually restrict programming languages, removing features that are unsafe and/or hard to thoroughly test and certify. There are also recommendations and demands in newer guidelines for the use of formal methods, as a way to achieve high assurance software. SPARK is a strict subset of Ada that was designed to have unambiguous semantics and that aimed at formal verification from the start. In this paper we present the SPARK language, its toolset, examples of projects where it has been used and argue why SPARK is relevant for academia and industry, especially for people interested in formal verification and safety critical systems. We also point directions for an improved use of SPARK. Concurrency will not be addressed in this paper.

Resumo: Os parâmetros para o desenvolvimento de *software* em sistemas *safety-critical* normalmente restringem as linguagens de programação, removendo características inseguras e/ou difíceis de testar e certificar de forma rigorosa. Existem também recomendações e exigências em novos parâmetros para o uso de métodos formais como forma de obter software com mais garantias. Neste artigo apresentamos a linguagem SPARK, o seu conjunto de ferramentas, exemplos de projectos onde foi usada e argumentamos o porquê do SPARK ser relevante para a academia e a indústria, especialmente para pessoas interessadas em verificação formal e sistemas *safety-critical*. Apontamos também direcções para um uso melhorado do SPARK. Neste artigo não abordaremos concorrência.

1 Introduction

Safety-critical systems are some of the most demanding systems when it comes to certification. This certification is not only required from the software but from the system as a whole, although in this paper we will only address features that are directly related to software.

When developing software for these systems, one of the most used programming languages is Ada[34]. Ada is a well known and respected language used in several domains of safety-critical software development. Furthermore, Ada is also a general purpose programming language, which has been kept up-to-date with several major revisions (83, 95, 2005 and now the future 2012) which have been standardized and thoroughly documented in the Ada Reference Manual[1].

When developing safety-critical systems, the language features that are allowed to be used are rather restricted. This is enforced by several guidelines (which vary depending on the application domain) with the aim of having software that is easier or, in some cases, possible, to certify mainly through thorough testing within a reasonable amount of time/cost/effort. These restrictions also depend on the integrity level (e.g. SIL, ASIL, DO-178B level) aimed by the software component that is being developed. This integrity level has to be given not only by the importance it has on the system but how it can affect other systems.

Some guidelines now incorporate recommendations and/or demand the use of formal methods so that it can be mathematically justifiable that the software behaves as intended, thus producing software that provides a higher assurance.

SPARK[6] is on the convergence of these aims. It is a strict subset of Ada that was designed to have unambiguous semantics and that aimed at formal verification from the start. It removes several features of the Ada language while still retaining enough expressive power so that it can be used in realistic scenarios.

The rest of this paper is organized as follows: Sec. 2 overviews design-by-contract and behavioral interface specification languages, two approaches that are at the basis of SPARK; in Sec. 3 we briefly present the SPARK language and its toolset; Sec. 4 surveys projects, both in industry and academia, where SPARK is/was used; in Sec. 5 we point out various aspects that could enhance SPARK and further justify the use of SPARK as a framework for rigorous software development; Sec. 6 concludes the paper.

2 Contracts & Specification

In this section we present what is Design-by-Contract (DbC) and in what ways it resembles and differs from Behavioral Interface Specification Languages (BISL). It is important to describe this since the language SPARK which we will be focusing on (as well as several others that aim at source code verification) has its approach rooted in this.

2.1 Design by Contract™

The term "Design by Contract" was first coined by Bertrand Meyer[26] and is largely associated with the Eiffel[27] programming language, an Object-Oriented programming (OOP) language, as part of the language's philosophy and design process. The term is also a registered trademark and some authors prefer to use the expression Programming by Contract.

The main ideas behind DbC are: a) to document the interface of modules[2] and its expected behaviour, b) to help with testing and debugging and c) to

[1] Usually referred to as ARM.
[2] The term modules is equivalent to package or class.

assign blame when a contract is breached. This is achieved by having structured assertions such as invariants and pre- and post-conditions.

In DbC, following the tradition of Eiffel, assertions are boolean expressions, often using subprograms[3] written in the same language as the host programming language and are intended to be checked at runtime (*runtime assertion checking (RAC)*), by executing them. Writing assertions in this way is friendlier to developers but it makes formal verification impossible because contracts are also code and not mathematical specifications describing the properties to be ensured.

In article[19] it was illustrated how Eiffel could have helped prevent the bug in the software of Arianne V, thus avoiding one of the most expensive software errors ever documented. What is stated is that the error that made Arianne V go wrong could have been avoided if the pre-conditions for the subprogram that failed had been clearly stated in the code. If the dependencies were clearly documented in the code then the verification & validation team would have been aware of what could (and did) generate a runtime error.

To sum it up, DbC is used to document source code and to have the program checked while it is executing, using structured annotations that are written as boolean expressions of the host programming language.

2.2 Behavioral Interface Specification Languages

Behavioral Interface Specification Languages was a term introduced with Larch [15]. The Larch family of languages had the specification, in a mathematical notation, written alongside the source code (Larch supported C, C++, Smalltalk and Modula-3). This resembles DbC but it differs greatly from traditional specification languages (SL), such as Z[32] and VDM[20], where the specification is written as a separate entity with no relation to the implementation.

Larch was focused mainly on specifying, with brevity and clarity, the interface of subprograms and datatype[4] invariants. Although some specifications[5] were executable, executability of specifications was not an objective of the Larch family of languages; this is the exact opposite of DbC. These specifications, along with the source code, would give rise to proof obligations.

The way that specifications are written in BISLs and SLs are similar. In both approaches, the specifications are written using a well-defined formal notation that is related to a well-defined formal logic. These formal definitions do not use expressions from the host language although they may look similar in some cases. This is another difference regarding DbC.

It is possible to animate/execute specifications written in some SLs (depending on the available tools) but this is still different from DbC and BISL because

[3] Subprograms is being used as a more generic way to refer to methods, functions, procedures and subroutines.

[4] Datatypes can also refer to classes. The term in LCL (Larch for the C language) was used to refer to structures.

[5] When talking about BISL, the terms “annotation” and “specification” are mostly interchangeable

we are animating a specification and not an implementation. Even so, we could argue that it is possible to refine a specification into an implementation.

While writing annotations in a mathematical notation is very expressive and particularly helpful for program verification, Larch has showed us that excessive mathematical notation can lead to the poor adoption of a BISL. JML[22] is a modern example of a BISL, rooted on the principles of Larch, which has taken this into account. JML avoids excessive mathematical notation, while having a mathematical background, and has gained several supporters in the academic and industrial arena, especially with the success of JavaCard[33] verification tools[3,24].

JML, as noted in[23,11], is associated with a set of tools that makes possible to overcome the typical non-executable nature of BISLs. Also, besides being able to do RAC, it also allows for the formal verification of Java programs, given the right tools/frameworks.

ACSL[6][9] is another interesting and modern BISL. While it has a large influence from JML, it has greater expressive power regarding the definition of mathematical entities.

It is also worth mentioning that BISLs, contrary to the classical notion of DbC, are not only useful for OOP but they can also be applied to other programming paradigms. Larch, for example, had support for C and ACSL was designed specifically for C. It is only because it is more natural to use this type of specifications in the OOP context that it has been largely associated with it.

3 SPARK

In this section we describe the features of the SPARK programming language and how it is supposed to work with the help of its toolset. We will focus mainly on program verification and proof support for SPARK, with the tools that are available from the official toolset distributed by AdaCore and Altran Praxis. This section assumes SPARK GPL 2009 (version 8.1.x), unless it is stated otherwise[7].

3.1 The Programming Language

SPARK stands for SPADE Ada Kernel[8]. SPADE was a previous project from Program Verification Limited, with the same aims as SPARK, but using the Pascal programming language. The company later became Praxis High Integrity Systems and it is now called Altran Praxis.

SPARK is both a strict subset of the Ada language, augmented with source code annotations, and also a toolset that supports its methodology. The SPARK annotations can be thought of as a BISL.

[6] ANSI/ISO C Specification Language.
[7] There are plans to launch a SPARK GPL 2010 version with features from SPARK Pro 9 during this summer[25].
[8] The R in SPARK is only there for aesthetic purposes.

It is a strict/true subset of the Ada language because every valid SPARK program is a valid Ada program; in fact, SPARK does not have a compiler and depends on Ada compilers to generate binary code. SPARK was also cleverly engineered so that the semantics of SPARK programs do not depend on decisions made by a specific compiler implementation (e.g. whether a compiler chooses to pass parameters of subprograms by value or by reference). Although outdated, SPARK also has a formal semantics[28] that is defined using operational semantics and Z notation.

SPARK removes some features of the Ada language such as recursion, dynamic memory allocation, access types[9], dynamic dispatching and generics. It also imposes certain restrictions on the constructs of the language (e.g. array dimensions are always defined by previously declared type(s) or subtype(s)[10]).

On top of the language restrictions, it adds annotations that allow for the specification of data-flow, creation of abstract functions and abstract datatypes and to the use of structured assertions (loop invariants are available but package[11] invariants are not). There is also a clear distinction between procedures (which may have side-effects) and functions (which are pure/free of side-effects and also have to return a value). SPARK provides no way to execute annotations.

Ada has the notions of package (specification) and package body (implementation); these are the building blocks for OOP in SPARK and Ada, although we may choose not to use the OOP features of the language. SPARK restricts OOP features that make the code hard to verify, such as dynamic dispatching. This issue is also addressed in[7], relatively to Ada 2005.

Package specifications can have abstract datatypes and abstract functions, which can then be used to define an abstract state machine. When coding the implementation, the abstract datatypes have to be refined into concrete datatypes, using the own annotation, and the definitions for abstract functions are written into *proof rule* files. These files are used when trying to discharge verification conditions.

Although it is not our aim to talk about the information flow capabilities of SPARK, it is interesting to note that in SPARK Pro 9, which has the SPARK 2005 version of the language, is also possible to validate secure and safe flow of information between different integrity levels and information security levels (i.e. unclassified, top secret).

All these features enable the possibility of developing software using the Correctness by Construction approach[12] where a software component is designed and developed with the aim of being formally verified.

[9] Users not familiar with Ada should note that these are equivalent to pointers.

[10] In Ada, the type mechanism allows for the definition of *ranges*. These ranges are limited by a lower and upper bound, known as T'First and T'Last where T is the type. It is also possible to get the range using T'Range.

[11] As previously stated, packages are equivalent to modules and classes

[12] This is different from the refinement approach taken by the B Method.

Finally, it should be noted that SPARK has support for a subset of Ravenscar[13] dubbed RavenSPARK[30].

3.2 Toolset & Program Verification

The SPARK toolset is what enables the use of SPARK. By not having a compiler, it must have a tool that checks the restrictions of the SPARK language; this tool is called the Examiner. The Examiner is also the verification condition (VC) generator (VCGen).

With the SPARK toolset we can use its proof tools to discharge the VCs. The Simplifier is the automated theorem prover and tries to discharge the VCs by using predicate inference and rewriting. While the tool is very successful in discharging VCs related to safety[17], it is not as capable of discharging VCs related to functional correctness as other modern provers[18]. The Proof Checker is the interactive prover/proof assistant of the toolset. It is a difficult tool to use and there is not much documentation provided and/or available.

An auxiliary tool, that ties all these tools together, is called POGS. POGS stands for Proof ObliGation Summarizer. It generates a report with much information, including warnings and errors related to the code but, most importantly, it constructs the report regarding the VCs that have been proven, that remain to be proven and those that have been shown to be false.

The SPARK Pro 9 toolset also has another tool called the ZombieScope. This tool provides analysis related to dead paths. A dead path is a piece of code that will never be reached; this is most likely caused by programming errors. This tool generates dead path conditions (similar to VCs) and tries to disprove that something is not reachable; if it fails, then it is a dead path. This information also appears on the report generated by POGS, as expected.

The GPS (GNAT Programming Studio), which is provided by AdaCore, has support for SPARK and it has been constantly updated so it is easier to use the SPARK toolset inside an IDE[14] that is familiar to some Ada users. GPS is not integrated in the toolset but it is tightly coupled to it.

4 Projects & Methodologies Related to SPARK

In this section we illustrate the capabilities of SPARK by considering both projects where the language has been used and also some methodologies and tools that use or generate SPARK. As a preview, we will talk about Tokeneer, Echo, SCADE and the C130J helicopter in this section.

4.1 Industrial & Academic Projects

Tokeneer[5] was an industrial project developed under a contract with the NSA, to show that it was possible to achieve Common Criteria EAL5 in a cost effective

[13] Ravenscar is a limited subset of concurrency and real-time of the Ada language.

[14] Integrated Development Environment.

manner. The purpose of the project was to develop the “core” part of the Tokeneer ID Station (TIS). The TIS uses biometric information to restrict/allow physical access to secured regions.

The project was successful, exceeding EAL5 requirements in several areas, and was allowed to be publicly released during 2009, along with all deliverables and a tutorial called “Tokeneer Discovery”. In 9939 lines of code (LOC), there were 2 defects; one was found by Rod Chapman, using formal analysis, and another during independent testing, thus achieving 0,2 errors per kLOC. Tokeneer has been proposed as a Grand Challenge of the Verified Software Initiative[36].

Praxis also published a short paper[12] where they described their industrial experience with SPARK (up until 2000). In that paper it is presented SHOLIS (a ship-borne computer system), MULTOS CA (an operating system for smartcards) and the Lockheed C130J (Hercules) helicopter, for which SPARK was used to develop a large part (about 80%) of the software for the Mission Computer.

These case studies illustrate several features of the SPARK language, including how the language eases MC/DC verification by having simpler code, how proofs are maintained and can be used as “regression proofs” instead of “regression testing” and how SPARK can also inhabit multi-language systems. It also shows that SPARK is not appropriate to being adopted late in the development, especially when trying to re-engineer Ada code into SPARK.

Recently[1] SPARK has been chosen as the programming language for a new NASA project, a lunar lander mission. SPARK will be used to develop the software of the CubeSat project. SPARK was also used as a target for a study on verified component-based software, based on a case study developed in SPARK for a missile guidance system[21]. This goes hand-to-hand with the SPARK philosophy of Correctness by Construction.

The Open-DO initiative[15] is also developing a project, called Hi-Lite[16], which uses SPARK in several ways. One of the aims of the project is to create a SPARK profile for the Ada language[17]. While SPARK itself can be viewed as a profile for Ada, the aim of Hi-Lite is to provide a less restrictive SPARK, with the benefits that SPARK has shown over time. Another aim of the project is to write a translator from SPARK (or sparkify) to Why, so that it is possible to use Why’s multi-prover VCGen. Why’s VCGen has support for interactive provers as well as automated provers. This would provide an alternative toolchain for SPARK besides the one that is maintained by Altran Praxis.

There is also ongoing research work by Paul B. Jackson[18] in translating the VCs generated by the Examiner to SMT-Lib input. This work, that originated from the academia, is scheduled to be included in the SPARK Pro toolset as an experimental feature, in a future release.

[15] http://www.open-do.org
[16] It should be noted that this project is very large and this is just a small portion of the aims of that project.
[17] This is being called sparkify.

4.2 Industrial Tools & Methodologies

The SCADE Suite is a well known software design tool that has been used, for example, by Airbus in the development of the Airbus A340 and is being used in several of the Airbus A380 projects. The SCADE Suite has a certified code generator for SPARK[2]. Perfect Developer[14] is another tool for modelling software for safety-critical systems that has a code generator for SPARK. It is out of the scope of this paper to try and thoroughly explain these tools but, in a very simple way, these tools are driven by model engineering in a formal setting and have been used in large scale projects, especially SCADE.

SPARK is also being used as an essential component of the Echo approach. This approach[37] is able to tightly couple a specification (in PVS[29]) with an implementation. It generates SPARK annotations from the PVS specification; this generated code is then completed to form the implementation. Afterwards, the tool extracts properties from the implementation to check if the implementation conforms to the specification. This approach has been used to formally verify an optimized implementation of the Advanced Encryption Standard.

5 SPARK as a Workbench for Rigorous Software Development

After showing some of the capabilities of SPARK and the support it has from the industry and academia, we now provide some insights on what we believe we can do to improve SPARK by enhancing or adding new features.

5.1 Theoretical Foundations of SPARK & Further Improvements

As it has been stated previously, SPARK has already a formal semantics that is specified with operational semantics and Z notation, but it is already outdated.

We believe that it is important to update the formal semantics of the language, for several reasons. First and foremost, it can be argued that since the domains where the language is used are mostly related to safety-critical and that the language aims at formal verification, it should be formally specified and be kept up-to-date, so that there is no distinction between the formal specification and the implementation.

Also, the semantics is separated into two documents describing the static and dynamic semantics of the language. An unification of the semantics could have benefits (which is also stated in the document but was not carried out). Another question is that the specification, to our knowledge and to what is shown in the documents, was not thoroughly verified (although the use of Z guarantees that at least everything is properly typed). By this we mean that there are no proofs showing relevant properties of the language that are stated, such as expressions being pure (i.e. free of side effects).

Yet importantly, there is only the operational semantics. There is no axiomatic semantics nor weakest pre-condition (or strongest post-condition) calculus nor a specification for a VCGen. While this is implemented in the toolset, an implementation is not a specification (although it could be used as a reference).

For greater formalisation and assurance, we believe that the language would benefit from having a specification for the axiomatic semantics and for the VCGen. The axiomatic semantics could be shown to be sound, regarding the operational semantics and the VCGen could be shown to be sound and correct, regarding the axiomatic semantics. Ideally this would involve mechanical verification using interactive proof assistants. This type of formal verification has already been done by Homeier[16] for a standard While language.

SPARK is ideal for this because it is a real programming language, used in large scale projects, but it is also one of the smallest real languages that do not compromise expressive power, at least for its application domain. Even though it is smaller than most languages, it is much bigger than any toy language.

Thus SPARK provides several challenges for researchers, related to its type system, to the separation between specification and implementation, to the refinement of datatypes and so on and so forth.

There is ongoing work on this area from the author of the paper.

5.2 Adding Features and Tools for Program Verification

One of the most obvious things that is lacking in SPARK is loop variants. While it could be argued that most loop variants in safety-critical systems are trivial, and that is why they still have not been implemented in SPARK, SPARK would benefit from having the possibility of specifying loop variance in its annotations, for proving the termination of non-trivial loops.

Package invariants would also benefit SPARK. SPARK is able to restrict the use of global variables in subprograms through the use of *global* and *derives* annotations, but SPARK is not able to assert that the program state has always specific properties because it lacks the notion of package invariant. Package invariants raise some implementation difficulties, such as temporary invalid states in a sequence of assignments and problems related to inheritance in OOP. These problems have been addressed by JML in the past so we are certain that they can be dealt with, especially given the restrictive nature of OOP in SPARK.

Regarding the BISL of SPARK, we could add a better way to write abstract functions, logical predicates, axioms and lemmas. ACSL is one of the most powerful BISLs when it comes to the expressiveness of these specifications. To maintain compatibility, these annotations could use a different notation so that the SPARK tools could ignore them (they would be dealt with by specific tools).

It would also be interesting to have the possibility of writing algebraic definitions in SPARK. This would enable to write things such as $top(push(stack, x)) = x$. Being able to write these types of specifications would allow for a higher level of reasoning and it would also allow to define properties about the sequence of execution (or protocol) of a given package or set of packages. The Common Algebraic Specification Language (CASL)[4] has a well-defined semantics and could serve as a basis for this. It was developed under the rationale of specifying requirements, design and architecture of conventional software.

Another limitation of SPARK is in dealing with the verification of floating-point arithmetic (the documentation of SPARK[31] states that the realrtc option may detect numerical errors in programs but not their absence, much like testing). This is not exclusive to SPARK since the verification of floating-point arithmetic is a difficult issue (and a topic of active research) that many approaches do not even try to deal with and assume real arithmetic instead of floating-point arithmetic. There is a recent article[10] with an interesting approach to this problem. It suggests a dedicated tool to deal with the verification of the non floating-point part of the program and to use Gappa (*Génération Automatique de Preuves de Propriétés Arithmétiques*) to deal with the verification of floating-point arithmetic.

5.3 Adding Features and Tools for Certification

The certification process for critical systems relies heavily on testing, even in the presence of formal methods; this is true even for DO-178C. For JML there is a tool[11] called *jmlunit* which is capable of generating test inputs by looking at invariants and pre- and post-conditions, but it forces the user to supply predefined data, as examples for the tests.

QuickCheck[13] is another approach for generating tests. Although we have to specify generators for our custom datatypes (default datatypes have predefined generators), this definition allows for greater variation. QuickCheck seems like it could be adapted to SPARK and any implementation that would be made would benefit greatly from the strongly-typed system of the language, probably reducing the number of necessary generators that would have to be created (although it should always be possible to create test generators, for our own specific purposes, if we wish so).

It should be noted that there are several other approaches to the automatic generation of test inputs and several other tools, as for example TestEra and Korat. Because of space constraints we chose to cite only these two, since they are extremely popular at the present date and also have a large community supporting them. It should also be said that any eventual test generation could also benefit from the algebraic specification mentioned earlier by having the protocol guide the test generation process.

The approaches to testing that are described on the previous paragraphs are very close to Model-Based Testing (MBT) based on source code annotations. With this type of MBT, the model is derived from the specification in the source code and then it generates abstract test data. A model checker then checks to see if the properties of the model are being respected. In some cases, it can also be possible to generate real test data and supply it to the program as unit test.

The MBT approach is used by the Spec Explorer[35] tool, using programs written in Spec#[8] as the basis for the model. In Spec Explorer, when a counter example is found, it is shown as a graph with all the actions that took place, following the order and values that lead to the error. To do this, Spec Explorer needs to have an automata of the program, which defines the protocol for it. This approach also allows to model check and test reactive systems. This feature may be helpful for model-checking and to do MBT on systems that may interact with sensors and/or actuators, which are an essential part of critical systems.

6 Conclusion

We have presented a short introduction to what is SPARK, the philosophy and toolset that support the language, and presented also projects, both industrial and academic, which use SPARK in their development. Furthermore, we have provided some directions to enhance SPARK that could interest people in formal verification and safety-critical software development.

We believe that the directions we suggest can bring further support to SPARK and its approach and they can be particularly useful in fostering a productive environment between the academia and industry, especially in further developing industrially usable and scalable formal methods for the (near) future.

References

1. Altran Praxis: New lunar lander project relies on SPARK programming language, http://www.altran.com/document/?f=Altran_20100610_CP_EN.pdf
2. Amey, P., Dion, B.: Combining model-driven design with diverse formal verification (January 2006)
3. Antoine, L.B., Requet, A.: JACK: Java Applet Correctness Kit. In: In Proceedings, 4th Gemplus Developer Conference (2002)
4. Astesiano, E., Bidoit, M., Kirchner, H., Krieg-Brückner, B., Mosses, P.D., Sannella, D., Tarlecki, A.: CASL: the common algebraic specification language. Theor. Comput. Sci. 286(2), 153–196 (2002)
5. Barnes, J., Chapman, R., Johnson, R., Widmaier, J., Cooper, D., Everett, B.: Engineering the Tokeneer enclave protection software. In: IEEE International Symposium on Secure Software Engineering (ISSSE). IEEE Press (2006)
6. Barnes, J.: High Integrity Software: The SPARK Approach to Safety and Security. Addison Wesley, first edn. (March 2003)
7. Barnes, J.: Safe and Secure Software: An invitation to Ada 2005. AdaCore (2008)
8. Barnett, M., Rustan, Schulte, W.: The Spec# programming system: An overview (2005)
9. Baudin, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language, version 1.4 (2009)
10. Boldo, S., Filliâtre, J.C., Melquiond, G.: Combining Coq and Gappa for certifying floating-point programs (2009)
11. Burdy, L., Cheon, Y., Cok, D., Ernst, M.D., Kiniry, J., Leavens, G.T., Rustan, K., Leino, M., Poll, E.: An overview of JML tools and applications (2003)
12. Chapman, R.: Industrial experience with spark. Ada Lett. XX(4), 64–68 (2000)
13. Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of Haskell programs (2000)
14. Crocker, D.: Perfect developer: A tool for object-oriented formal specification and refinement (2003)

15. Guttag, J.V., Horning, J.J., Garl, W.J., Jones, K.D., Modet, A., Wing, J.M.: Larch: Languages and tools for formal specification. In: Texts and Monographs in Computer Science. Springer-Verlag (1993)
16. Homeier, P.V., Martin, D.F.: Trustworthy tools for trustworthy programs: A verified verification condition generator. In: TPHOLs. pp. 269–284 (1994)
17. Jackson, P.B., Ellis, B.J., Sharp, K.: Using SMT solvers to verify high-integrity programs. In: AFM '07: Proceedings of the second workshop on Automated formal methods. pp. 60–68. ACM, New York, NY, USA (2007)
18. Jackson, P.B., Passmore, G.O.: Proving spark verification conditions with smt solvers (December 2009)
19. Jazequel, J.M., Meyer, B.: Design by Contract: the lessons of Ariane. Computer 30(1), 129–130 (1997)
20. Jones, C.B.: Systematic software development using VDM (2nd ed.). Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1990)
21. Lau, K.K., Wang, Z.: Verified component-based software in SPARK: Experimental results for a missile guidance system. In: Proc. 2007 ACM SIGAda Annual International Conference. pp. 51–57. ACM (2007)
22. Leavens, G.T., Baker, A.L., Ruby, C.: JML: A notation for detailed design (1999)
23. Leavens, G.T., Cheon, Y., Clifton, C., Ruby, C., Cok, D.R.: How the design of JML accommodates both runtime assertion checking and formal verification. Sci. Comput. Program. 55(1-3), 185–208 (2005)
24. Marche, C., Mohring, P.C., Urbain, X.: The Krakatoa tool for certification of Java/JavaCard programs annotated in JML (2004)
25. Messer, R.: Introducing SPARK Pro 9.0 webinar (April 2010), http://www.adacore.com/home/products/gnatpro/webinars/
26. Meyer, B.: Applying "Design by Contract". Computer 25(10), 40–51 (October 1992)
27. Meyer, B.: Object-Oriented Software Construction. Prentice Hall PTR, 2nd edn. (March 2000)
28. O'Neil, I.: The formal semantics of SPARK83 (1994)
29. Owre, S., Rushby, J.M., , Shankar, N.: PVS: A prototype verification system. In: Kapur, D. (ed.) 11th International Conference on Automated Deduction (CADE). Lecture Notes in Artificial Intelligence, vol. 607, pp. 748–752. Springer-Verlag, Saratoga, NY (jun 1992), http://www.csl.sri.com/papers/cade92-pvs/
30. SPARK Team: SPARK Examiner: The SPARK Ravenscar Profile (January 2008)
31. SPARK Team: Supplementary Release Note - The RealRTC Option (February 2009)
32. Spivey, J.M.: The Z notation: a reference manual. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1989)
33. Sun microsystems: Java Card Techonolgy (2000)
34. Taft, S.T., Duff, R.A., Brukardt, R.L., Ploedereder, E., Leroy, P.: Ada 2005 Reference Manual. Language and Standard Libraries: International Standard ISO/IEC 8652/1995(E) with Technical Corrigendum 1 and Amendment 1 (Lecture Notes in Computer Science). Springer-Verlag New York, Inc., Secaucus, NJ, USA (2007)
35. Veanes, M., Campbell, C., Grieskamp, W., Schulte, W., Tillmann, N., Nachmanson, L.: Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer (2008)
36. Woodcock, J.: Tokeneer experiments, http://research.microsoft.com/en-us/events/vsworkshop2009/jim_woodcock.pdf
37. Yin, X., Knight, J.C., Nguyen, E.A., Weimer, W.: Formal verification by reverse synthesis. In: SAFECOMP. pp. 305–319 (2008)