Useful features and tools of Behavioral Interface Specification Languages for SPARK

Eduardo Brito
Universidade do Minho

SPARK is a (true) subset of the Ada programming language, which is augmented with source code annotations. These annotations enable the static analysis and formal verification of source code, for which SPARK is known.

In this survey, we study other behavioral interface specification languages, such as JML and ACSL, and try to extract useful features that could be added to SPARK. We also consider the difficulties and/or impossibility behind such additions, how would they relate to the SPARK philosophy and what implications it would have on SPARK.

Categories and Subject Descriptors: A.1 [**Introductory and Survey**]: Survey; D.2.4 [**Software Engineering**]: Software/Program Verification – Class invariants; Correctness proofs; Formal methods; Programming by contract; D.3.3 [**Programming Languages**]: Language Constructs and Features – Abstract data types; Procedures, functions, and subroutines; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs – Assertions; Invariants; Mechanical verification; Pre- and post-conditions; Specification techniques

General Terms: Design, Language, Proof, Verification

Additional Key Words and Phrases: ACSL, Automated theorem provers, Behavioral Interface Specification Languages, Design by Contract, Dynamic Logic, High Assurance Software, High Integrity Software, Hoare Logic, Interactive provers, JML, Larch, Programming by Contract, Proof assistants, Safety-critical systems, Separation Logic, SMT provers, SPARK, Spec#

1. INTRODUCTION

Design by Contract [Meyer 1992; 2000] (**DbC**) and Behavioral[1] Interface Specification Languages [Hatcliff et al. 2009] (**BISL**) have received much attention in the software engineering community for the past two decades and it is becoming a standard practice to use these methodologies, especially in Object-Oriented (**OO**) software.

While OO programming (**OOP**) is a well established practice in the development of non safety-critical software, it has found very little acceptance in the safety-critical arena [M.Brosgol 2008]. This lack of acceptance of OO technology (**OOT**) stems from the increased difficulty of testing and certifying OO software (**OOS**) but also from the lack of accepted guidelines that address OOP for the development

[1]Behavioral is American english while behavioural is British english. It is used *behavioral* in this survey because BISL has an American origin.

This work is being partially funded by a research grant of the FCT-funded RESCUE project (PTDC/EIA/65862/ 2006). Author's address: CCTC / Departamento de Informática, Universidade do Minho, Campus de Gualtar, 4710-057 Braga, Portugal. Author's e-mail: edbrito@di.uminho.pt

Universidade do Minho Master Course on informatics - State of the Art Reports 2010

of such systems, such as DO-178B [Taylor et al. 2002]. These guidelines are mainly focused on the existing difficulties of certifying Ada 83 and C software [Cromar 2009].

Although it is more difficult to test OOS, there is interest in using OOP because of its several advantages, such as reuse and faster development cycle. There has been at least one study from the FAA[2] on what has to be done to ease the testing and certification of OOS [Federal Aviation Administration 2004].

The upgrade of DO-178B to DO-178C, which is expected to be released during 2010 [McHale 2009], will contain an OOT suplement as well as a formal methods (**FM**) suplement, giving guidelines on both aspects [Cromar 2009]. Also, Common Criteria EAL7[3] already demands the use of FM [Common Criteria Recognition Agreement 2009].

This survey will focus on features and tools of several BISL that we deem to be useful additions to a specific programming language, SPARK [Barnes 2003], which is also a BISL, that is used in safety-critical systems software.

The remainder of this survey is organized as follows: Section 2 will explain the main ideas behind BISL, SPARK and other related areas, Section 3 will focus on features and related technologies we deem useful for SPARK, Section 4 will give a glimpse at other useful features that were not approached on this survey and Section 5 will conclude the survey and reflect upon our findings.

2. BEHAVIORAL INTERFACE SPECIFICATION LANGUAGES, SPARK AND RELATED AREAS

In this section we show and explain the information that is needed throughout the rest of the paper. We define what is Design by Contract and how it differs from Behavioral Interface Specification Languages and we present the SPARK language, which is the language that will focus on the survey.

2.1 Design by Contract^{TM}

The term "Design by Contract" was first coined by Bertrand Meyer [Meyer 1992; 2000] and is largely associated with the Eiffel programming language, an 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 to: a) document the interface of modules[4] and its expected behaviour, b) to help with testing and debugging and c) to 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[5], written in the same language as the host programming language and are intended to be checked at runtime (*runtime assertion checking (**RAC**)*). This is detailed in such books as [Meyer 2000; Mitchell and McKim 2001].

[2]Federal Aviation Administration
[3]Evaluation Assurance Level 7
[4]The term modules is equivalent to package or class
[5]Subprograms is being used as a more generic way to denote methods, functions and procedures

Writing assertions in this way is friendlier to developers but it makes static verification impossible in most cases.

An article was also published [Jazequel and Meyer 1997] where it was illustrated how Eiffel could have helped to prevent the bug in the software of Arianne V, thus avoiding one of the most expensive software errors ever recorded. Although the use of DbC for testing and debugging is regarded as a benefit for non safety-critical software and, specifically in the case of Eiffel, the *rescue* assertion can help to recover from pre- and post-conditions that are breached, Meyer shows that this was not feasible in this case and that the Ada exception mechanism could have dealed with this also.

What is stated is that the error that crashed Arianne V could have been avoided if the pre-conditions for the subprogram that failed had been clearly stated in the code; if it was explicitly declared in the code, it is assumed that such condition would have been checked at the testing stage by the validation & verification (**V&V**) team.

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, as far as we know, was a term introduced with Larch [Guttag et al. 1993]. The Larch family of languages would, unlike other more traditional specification languages such as Z [Spivey 1989] and VDM [Jones 1990], have the specification written alongside the source code (Larch supported C, C++, Smalltalk and Modula-3). This is in the same line as DbC but it differs greatly from traditional specification languages 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 datatypes[6] invariants. Although some specifications[7] were executable, executability of specifications was not an objective of the Larch family of languages; this is the exact opposite of DbC. These specifications would then be checked along with the source code and proof obligations would be generated.

The way that specifications are written in BISL and specification languages are similar. In both cases, the specifications are written using a well-defined formal and mathematical logic. These formal definitions do not use any kind of expressions from a host language although they may look similar in some cases. This is another difference regarding DbC, where the annotations are written as expressions of the host language.

It is possible to animate/execute specifications written in some specification languages (depends on the available tools and methodologies) but this is still different from DbC and BISL because we are animating a specification and not the code of an implementation. Even so, it is possible to refine a specification into an implementation but this is not the focus of the survey.

While writing the annotations in a mathematical notation is much more expres-

[6]Datatypes can also refer to classes. The term is used because in LCL (Larch for the C language) it is used to refer to structures.

[7]When talking about BISL, the terms "annotation" and "specification" are mostly interchangeable

sive and particularly good for program verification, Larch has showed us that excessive mathematical notation can lead to the poor adoption of a BISL. JML [Leavens et al. 1999] 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 circles, especially with the success of Java Card [Sun microsystems 2000] certification [Antoine and Requet 2002; Engel et al. 2008; Marche et al. 2004].

JML, as noted in [Leavens et al. 2005; Burdy et al. 2003], 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 what is called RAC, it also allows for the formal verification of Java programs, given the right tools.

ACSL[8] [Baudin et al. 2009] is another interesting and modern BISL. While it does have a large influence from JML, it has greater expressive power regarding the definition of mathematical structures. It also adds the notion of *labels*, allowing for the definition of frame conditions[9]. Labels are a very powerful framing mechanism for languages with dynamic memory allocation, such as C. ACSL is the BISL used by the Jessie [Baudin et al. 2009] plug-in of the Frama-C tool.

It is also worth mentioning that BISL, contrary to the classical notion of DbC, is not only useful for OOP and OOT but it can also be applied to other paradigms of programming languages. Larch, for example, had support 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.

2.3 SPARK

SPARK stands for SPADE Ada (?)[10] Kernel. SPADE was a previous project from Program Verification Limited, with the same purposes as SPARK, but using the Pascal programming language. The company is now called Praxis High Integrity Systems.

SPARK is both a (true) subset of the Ada language (Ada 95 at the present) and also a toolset that supports its methodology.

It is a 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 code. SPARK was also cleverly engineered so that the meaning 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, will not make a difference for the meaning of the program). Although outdated, SPARK has a formal semantic that is defined using operational semantics and Z notation [O'Neil 1994].

SPARK removes some features of the Ada language such as recursion, dynamic memory allocation, pointers, dynamic dispatching and generics. It also imposes certain restrictions on the constructs of the language (i.e. "exit when/until" inside loops must be at the beginning or end of a loop; returns may only be used at the

[8]ANSI/ISO C Specification Language.
[9]Conditions related to program state at a given moment.
[10]The R in SPARK only exists because "SPARK" is prettier than "SPAK"!

end of a function; arrays are always defined by a type or subtype[11]).

On top of the language restrictions, it adds annotations that allow for the specification of data-flow, abstract functions, abstract datatypes and structured assertions (loop invariants are available but package[12] invariants are not). In SPARK, annotations are never executable. Also, there is a clear distinction between procedures (which may have side-effects) and functions (which are pure, in the mathematical sense, and can be used in annotations, although annotations are not executable).

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 OO 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 this book, related to Ada 2005 [Barnes 2008].

Package specifications can have abstract datatypes and abstract functions, which can then be used to define an abstract state machine. The abstract functions and datatypes should be used in pre- and post-conditions and when implementing the package body, the abstract datatypes have to be refined into a specific datatype, using the own annotation. This allows for a great separation between a package's specification and its implementation. Abstract functions are defined only in *proof rules* files. These files contain

The SPARK toolset is what makes SPARK possible. By not having a compiler, it must have a tool that checks that the restrictions being imposed by SPARK are being met. The tool that is responsible for this is the Examiner. The Examiner is also responsible for generating the verification conditions (**VC**) that are to be discharged by the proof tools of the toolset.

SPARK's VCs can only be discharged using the SPARK toolset. The available tools are the Simplifier and the Proof Checker. The Simplifier tries to discharge the VCs by using term inference and rewriting. While it is very successful discharging VCs related to safety [Jackson et al. 2007], it lacks the expressive power that can be found in other BISLs such as ACSL.

We also find that the Proof Checker, the interactive prover/proof assistant of the toolset, to be quite hard to use and less powerful than other proof assistants, namely Coq [The Coq development team 2004].

Finally, it should be noted that SPARK has support for a subset of Ravenscar[13] dubbed RavenSPARK [SPARK Team 2008]. We will give a glimpse of this in Section 4.

3. FEATURES AND TOOLS OF BEHAVIORAL INTERFACE SPECIFICATION LANGUAGES

The work presented in this section reflects the several surveys, articles and books related to SPARK, BISLs and program verification that were studied [Leavens et al. 2007; Filliâtre and Marché 2007; 2004; Hatcliff et al. 2009; Kiniry et al. 2008;

[11]In Ada, the type mechanism allows for the definition of *ranges*. These ranges are limited by a lower and upper bound, know as T'First and T'Last where T is the type. It is also possible to get the range using T'Range.
[12]As previously stated, packages are equivalent to modules and classes
[13]Ravenscar is a limited subset of concurrency and real-time for the Ada language

Leavens et al. 2005; Cok and Kiniry 2004; Leavens et al. 2006; Burdy et al. 2003; Engel et al. 2008; Baudin et al. 2009; Jackson et al. 2007], several discussions with people directly involved with Praxis, AdaCore, Frama-C and Université de Paris 12 and the use of SPARK in research activities.

3.1 Features and tools for static verification

Loop variants would be a useful feature for SPARK. While it could be argued that most loop variants in safety-critical systems are trivial, it would benefit SPARK to have the possibility of specifying loop variance in its annotations, for proving (possibly automatically) the termination of non-trivial loops. ACSL provides this feature and complements it with a *terminates \true* construct that skips loop variance checking, although this is not recommended for most loops. Loop variants and several approaches to attack this verification problem are given in [Hatcliff et al. 2009].

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 it is not able to assert that a given state is always valid because it does not have this notion of package invariant. Package invariants have some implementation difficulties that need to be addressed when implementing, such as temporary invalid states in a sequence of assignments. ACSL solves this problem by defining the notion of weak and strong global invariants. Inheritance also represents a problem with this feature but the same approach that is used for pre- and post-conditions can be adapted to this case.

The SPARK toolset is limited when it comes to proving the behavioural correctness of programs. The Simplifier is very capable when it comes to dealing with safety checks[14] and discharging the related VCs but it fails short when we want to prove that the program has the intended behaviour and not only that it does not *crash*.

Thinking about the BISL of SPARK, this would mean that we would need to add a better way to write abstract functions, logical predicates and lemmas. ACSL is one of the most powerful BISLs when it comes to the expressiveness of these annotations. From a practical point-of-view, this would need a different prover toolchain for SPARK. This leads to the next useful feature that we believe to be needed in SPARK.

The SPARK tools are regarded as safe tools that can be used in program verification for critical systems and they are verified and supported by a software company. Even so, the proof tools that come with SPARK have some serious limitations and not only that, they are the only tools capable of dealing with the VCs that the Examiner generates. There are several automated theorem provers and Satisfiability Modulo Theories (**SMT**) provers that are used and studied in the academia. One possible target for a hypothetical VC generator (**VCGen**) could be SMT-LIB [Barrett et al. 2008] but this would only address automatic proof. An even more interesting alternative would be to use Why [Filliâtre and Marché 2007] as the target for the hypothetical VCGen; this would allow interfacing with several SMT

[14]In SPARK this is usually called runtime checks (rtc for short). Although it is called *runtime* it only generates, statically, VCs for program verification.

provers but also with proof assistants such as Coq [The Coq development team 2004].

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. CASL [Astesiano et al. 2002] is an algebraic specification language that could serve as the basis for this. This feature was a late addition and there was not much investigation into it although it promises to be an interesting alternative.

Another limitation that SPARK has is in dealing with the verification of floating-point arithmetic (the documentation of SPARK [SPARK Team 2009] says that enabling this option may enable the detection of 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 that many approaches do not even try to deal with and assume real arithmetic instead of floating-point arithmetic.

A recent article [Monniaux 2008] stressed the importance of floating-point computations in modern safety-critical systems and addressed several of its problems. There is a recent article [Boldo et al. 2009] with an interesting approach to this problem. What they propose is to have a dedicated tool (in this case, Coq) 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. Gappa has also been integrated as a backend prover for the Why platform.

3.2 Features and tools for dynamic certification

The certification process for critical systems relies heavily on testing, even in the presence of FM. This is true even for DO-178C, where it is stated that some tests may be removed by the use of FM but not all. For JML there is a tool [Burdy et al. 2003] called *jmlunit* which is capable of generating test inputs by looking at invariants, pre- and post-conditions but it forces the user to supply predefined data, as examples for the tests. QuickCheck [Claessen and Hughes 2000] is more powerful than *jmlunit*, generating random tests and even though we have to specify generators for our custom datatypes (default datatypes have predefined generators), the definition of the generators allows for more variation. QuickCheck seems like it could be adapted to SPARK and any implementation that would be made would benefit from the strong-typed system of the language. There are several other approaches to the automatic generation of test inputs and several other tools [Cheon et al. 2005; Marinov and Khurshid 2001; Boyapati et al. 2002] and albeit most of them have been discontinued, there is several publications describing their techniques.

These approaches to testing 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.

This approach is used by the Spec Explorer [Veanes et al. 2008] tool, using programs written in Spec# [Barnett et al. 2005] 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.

Even if MBT and model checking are not being used, it would be helpful to have the protocol of a program clearly specified in annotations, to serve as documentation on how to use package, and to have the protocol checked during runtime, giving a trace of the execution when the protocol was breached. These specifications would follow the automata reasoning of Spec Explorer and it would be checked during runtime. This is a compromise between having an algebraic specification and having a full-blown software model checker. It would be easier to implement, it would deliver a protocol as part of the specification and it could be executed and tested by the V&V team. To represent this as part of the specification, in text, it can be written using regular expressions. This approach was taken in this paper [Cheon and Perumandla 2007] by extending the JML language and compiler.

Another feature that could ease the verification process and help increase the productivity of the V&V team, would be the use of tools to help write loop invariants. Automatic detection and generation of loop invariants is an active research field. Recently, Daikon [Ernst et al. 2007] has managed to provide very interesting results, not just for loop invariants but also for class invariants. One problem with Daikon is that it relies on traces of program execution. This is a problem because the correctness of the invariant depends on the correctness of the program. Even so, this is a feature that may be feasible and may be worth to study it further, if it proves itself to be sound.

3.3 The underlining logic foundations

Up to this point in the survey, we have omitted the logical foundations of the tools and of the different approaches. This is because (almost) all of the deductive tools and features that were shown here are based on Hoare Logic [Floyd 1967; Hoare 1969] (**HL**). SPARK's Examiner uses HL with Dijkstra's weakest pre-condition [Dijkstra 1975] calculus to generate the verification conditions [SPARK Team 2009]. HL is the most influential and used logic for axiomatic reasoning of programs and their verification.

We believe that SPARK would benefit greatly in having a complete Hoare Logic and VCGen that would represent the language as it is nowadays. The formalization that was done [O'Neil 1994] is already outdated and for such a language, it should be important to have a well-defined, complete and sound logic backing up the tools. Université de Paris 12 has already a VCGen with Dijkstra's weakest pre-condition mathematically defined and implemented in Standard ML97. Their definition of SPARK is not yet complete; it is possible for someone to use that work, that is partially done, and to extend it to cover the rest of the SPARK language. It would also be interesting to have the HL and VCGen completely proved using a proof assistant and furthermore, to prove that the implementation is correct regarding the formal and verified VCGen.

For the sake of completeness, we will now address other logics that are present in relevant tools and languages that were mentioned.

In the KeY approach, for dealing with Java Card programs with JML annotations, Dynamic Logic[15] (*DL*) is used. Using DL, for example, it is possible to avoid writing loop invariants. This is done by making the user prove that the loop does what is supposed to do, by performing an inductive proof. DL has a different syntax from HL, although it is an extension to HL. DL is also the logic used by the KIV [Balser et al. 2000] formal method.

The Z notation [Spivey 1989] does not use Hoare Logic. It instead uses the set theory of ZermeloFraenkel to reason and specify programs. Praxis, the maintainers of SPARK, along with other British companies in the area of FM, continue to use Z. The Tokeneer [Praxis High Integrity Systems 2008] project, that was developed by Praxis, also uses this specification language.

4. A GLIMPSE AT FUTURE DIRECTIONS

In this survey we did not approach some language features that are quite interesting and relevant. These features are: a) generics, b) concurrency and c) real-time.

Ada 2005 has several libraries that use generics. The problem in using generics is that it is very hard to test all possible combinations and it is also hard to prove the correctness of subprograms that use them. It would be interesting to see if there are any developments in this area and to try and implement this in SPARK. This would enable the use of Ada's libraries.

Concurrency is still one of the hardest problems in program verification. Concurrency also "gave birth" to one of the most widely used formal methods in the industry, model checking [Clarke 2008]. Even though model checking has been somewhat successful in verifying concurrent software, the challenge of being able to do deductive proofs over concurrent software and to automate this is still open. There are some articles on the subject [Owicki 1975; Owicki and Gries 1976; Brookes 2007] but it seems that there is still much work to do on it. SPARK may provide a good basis for the development of this, especially considering the limitations that RavenSPARK imposes on concurrency.

Real-Time programming, besides having the same problems that can be found in concurrency, has an extra set of concerns. From modelling to implementing, to being able to describe the real-time behaviour of the software, to certify that the code is able to meet its deadlines in the worst-case scenario, to optimize the use of resources, to ensure that the system will not deadlock because of the priorities and scheduling and several more interesting challenges are being faced everyday in the development of safety-critical systems. SPARK and RavenSPARK may also provide a good basis for the development of formal methods associated with this problems.

All of these problems are real problems that are affecting the critical systems industry. It is of the utmost importance for the FM community to face and solve

[15]DL is an extension of HL

these problems.

5. CONCLUSION

SPARK is an industrial-level programming language for safety-critical systems, with tools that enable meaningful static analysis and formal program verification. In spite of this, we presented several features on this survey that we believe that can improve and provide a better framework for formal reasoning about SPARK programs.

We also presented other tools and techniques that are available on other BISLs and that could further develop the usage and adoption of SPARK and provide a faster turnaround for software companies that choose to use SPARK as their language for safety-critical systems.

The features, tools and techniques that were chosen, were chosen regarding their usefulness for safety-critical applications. Features and tools that were not sound, were not considered (such as Extended Static Checking [Detlefs et al. 1998]).

Besides formal verification, we presented testing and model based techniques that may help with the certification process of software systems that have to comply with strict guidelines such as DO-178B (and DO-178C in the future).

Finally, we hinted at some other directions where we think that research using SPARK and safety-critical systems should focus on.

REFERENCES

Antoine, L. B. and Requet, A. 2002. JACK: Java Applet Correctness Kit. In *In Proceedings, 4th Gemplus Developer Conference.*

Astesiano, E., Bidoit, M., Kirchner, H., Krieg-Brückner, B., Mosses, P. D., Sannella, D., and Tarlecki, A. 2002. CASL: the common algebraic specification language. *Theor. Comput. Sci. 286,* 2, 153–196.

Balser, M., Reif, W., Schellhorn, G., Stenzel, K., and Thums, A. 2000. Formal system development with KIV. In *FASE '00: Proceedings of the Third Internationsl Conference on Fundamental Approaches to Software Engineering.* Springer-Verlag, London, UK, 363–366.

Barnes, J. 2003. *High Integrity Software: The SPARK Approach to Safety and Security*, First ed. Addison Wesley.

Barnes, J. 2008. *Safe and Secure Software: An invitation to Ada 2005.* AdaCore.

Barnett, M., Rustan, and Schulte, W. 2005. The Spec# programming system: An overview.

Barrett, C., Ranise, S., Stump, A., and Tinelli, C. 2008. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`.

Baudin, P., Filliâtre, J.-C., Marché, C., Monate, B., Moy, Y., and Prevosto, V. 2009. *ACSL: ANSI/ISO C Specification Language, version 1.4.* `http://frama-c.cea.fr/acsl.html`.

Boldo, S., Filliâtre, J.-C., and Melquiond, G. 2009. Combining Coq and Gappa for certifying floating-point programs. In *Calculemus '09/MKM '09: Proceedings of the 16th Symposium, 8th International Conference. Held as Part of CICM '09 on Intelligent Computer Mathematics.* Springer-Verlag, Berlin, Heidelberg, 59–74.

Boyapati, C., Khurshid, S., and Marinov, D. 2002. Korat: automated testing based on Java predicates. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis.* ACM, New York, NY, USA, 123–133.

Brookes, S. 2007. A semantics for concurrent separation logic. *Theor. Comput. Sci. 375,* 1-3, 227–270.

Burdy, L., Cheon, Y., Cok, D., Ernst, M. D., Kiniry, J., Leavens, G. T., Rustan, K., Leino, M., and Poll, E. 2003. An overview of JML tools and applications.

Cheon, Y., Kim, M., and Perumandla, A. 2005. A complete automation of unit testing for Java programs. In *Software Engineering Research and Practice*. 290–295.

Cheon, Y. and Perumandla, A. 2007. Specifying and checking method call sequences of Java programs. *Software Quality Control 15,* 1, 7–25.

Claessen, K. and Hughes, J. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*. ACM, New York, NY, USA, 268–279.

Clarke, E. M. 2008. The birth of model checking. 1–26.

Cok, D. R. and Kiniry, J. R. 2004. ESC/Java2: Uniting ESC/Java and JML - progress and issues in building and using ESC/Java2. In *In Construction and Analysis of Safe, Secure and Interoperable Smart Devices: International Workshop, CASSIS 2004*. SpringerVerlag.

Common Criteria Recognition Agreement 2009. *Common Criteria for Information Technology Security Evaluation. Part 3: Security assurance components.* Common Criteria Recognition Agreement.

Cromar, C. 2009. DO-178C: upcoming guidance for OOS.

Detlefs, D. L., Leino, K. R. M., Rustan, K., Leino, M., Nelson, G., and Saxe, J. B. 1998. Extended static checking.

Dijkstra, E. W. 1975. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM 18,* 8 (August), 453–457.

Engel, C., Gladisch, C., Klebanov, V., and Rümmer, P. 2008. Integrating Verification and Testing of Object-Oriented Software. In *Tests and Proofs. Second International Conference, TAP 2008, Prato, Italy*, B. Beckert and R. Hähnle, Eds. LNCS 4966. Springer.

Ernst, M. D., Perkins, J. H., Guo, P. J., McCamant, S., Pacheco, C., Tschantz, M. S., and Xiao, C. 2007. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program. 69,* 1-3, 35–45.

Federal Aviation Administration 2004. *Object-Oriented Technology in Aviation.* Federal Aviation Administration.

Filliâtre, J.-C. and Marché, C. 2004. Multi-prover verification of C programs. 15–29.

Filliâtre, J.-C. and Marché, C. 2007. The Why/Krakatoa/Caduceus platform for deductive program verification. 173–177.

Floyd, R. W. 1967. Assigning meanings to programs. In *Proc. Sympos. Appl. Math., Vol. XIX.* Amer. Math. Soc., Providence, R.I., 19–32.

Guttag, J. V., Horning, J. J., Garl, W. J., Jones, K. D., Modet, A., and Wing, J. M. 1993. Larch: Languages and tools for formal specification. In *Texts and Monographs in Computer Science.* Springer-Verlag.

Hatcliff, J., Leavens, G. T., Leino, K. R. M., Müller, P., and Parkinson, M. 2009. Behavioral interface specification languages. Tech. Rep. CS-TR-09-01, University of Central Florida, School of EECS, Orlando, FL. Mar.

Hoare, C. A. R. 1969. An axiomatic basis for computer programming. *Communications of the ACM 12,* 10 (October), 576–580.

Jackson, P. B., Ellis, B. J., and Sharp, K. 2007. Using SMT solvers to verify high-integrity programs. In *AFM '07: Proceedings of the second workshop on Automated formal methods.* ACM, New York, NY, USA, 60–68.

Jazequel, J. M. and Meyer, B. 1997. Design by Contract: the lessons of Ariane. *Computer 30,* 1, 129–130.

Jones, C. B. 1990. *Systematic software development using VDM (2nd ed.).* Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Kiniry, J. R., Chalin, P., and Hurlin, C. 2008. Integrating static checking and interactive verification: Supporting multiple theories and provers in verification. 153–160.

Leavens, G. T., Abrial, J.-R., Batory, D., Butler, M., Coglio, A., Fisler, K., Hehner, E., Jones, C., Miller, D., Peyton-Jones, S., Sitaraman, M., Smith, D. R., and Stump, A. 2006. Roadmap for enhanced languages and methods to aid verification. In *GPCE '06:*

Proceedings of the 5th international conference on Generative programming and component engineering. ACM, New York, NY, USA, 221–236.

Leavens, G. T., Baker, A. L., and Ruby, C. 1999. JML: A notation for detailed design.

Leavens, G. T., Cheon, Y., Clifton, C., Ruby, C., and Cok, D. R. 2005. How the design of JML accommodates both runtime assertion checking and formal verification. *Sci. Comput. Program. 55,* 1-3, 185–208.

Leavens, G. T., Leino, K. R. M., and Müller, P. 2007. Specification and verification challenges for sequential object-oriented programs. *Formal Asp. Comput. 19,* 2, 159–189.

Marche, C., Mohring, P. C., and Urbain, X. 2004. The Krakatoa tool for certification of Java/JavaCard programs annotated in JML. *Journal of Logic and Algebraic Programming 58,* 1-2, 89–106.

Marinov, D. and Khurshid, S. 2001. TestEra: A novel framework for automated testing of Java programs. In *ASE '01: Proceedings of the 16th IEEE international conference on Automated software engineering.* IEEE Computer Society, Washington, DC, USA, 22.

The Coq development team. 2004. *The Coq proof assistant reference manual.* LogiCal Project. Version 8.0.

M.Brosgol, B. 2008. Safety and security: Certification issues and technologies. *Crosstalk: The Journal of Defense Software Engineering 21,* 10 (October).

McHale, J. 2009. Upgrade to DO-178B certification - DO-178C to address modern avionics software trends.

Meyer, B. 1992. Applying "Design by Contract". *Computer 25,* 10 (October), 40–51.

Meyer, B. 2000. *Object-Oriented Software Construction*, 2nd ed. Prentice Hall PTR.

Mitchell, R. and McKim, J. 2001. Design by Contract, by example. *Technology of Object-Oriented Languages, International Conference on 0*, 0430.

Monniaux, D. 2008. The pitfalls of verifying floating-point computations.

O'Neil, I. 1994. The formal semantics of SPARK83.

Owicki, S. and Gries, D. 1976. Verifying properties of parallel programs: an axiomatic approach. *Commun. ACM 19,* 5, 279–285.

Owicki, S. S. 1975. Axiomatic proof techniques for parallel programs. Ph.D. thesis, Ithaca, NY, USA.

Praxis High Integrity Systems 2008. *Tokeneer ID Station. EAL5 Demonstrator: Summary Report.* Praxis High Integrity Systems.

SPARK Team 2008. *SPARK Examiner: The SPARK Ravenscar Profile.* SPARK Team.

SPARK Team 2009. *Generation of VCs for SPARK Programs.* SPARK Team.

SPARK Team 2009. *Supplementary Release Note - The RealRTC Option.* SPARK Team.

Spivey, J. M. 1989. *The Z notation: a reference manual.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Sun microsystems 2000. *Java Card Techonolgy.* Sun microsystems.

Taylor, C., Alves-foss, J., and Rinker, B. 2002. Merging safety and assurance: The process of dual certification for software. In *Software, Proc. Systems and Software Technology Conf.*

Veanes, M., Campbell, C., Grieskamp, W., Schulte, W., Tillmann, N., and Nachmanson, L. 2008. Model-based testing of object-oriented reactive systems with spec explorer. 39–76.