Developing a Program Logic for the SPARK Programming Language

Eduardo Brito
HASLab / INESC TEC
Universidade do Minho
Campus de Gualtar, 4710-057 Braga, Portugal
edbrito@di.uminho.pt

Abstract. Ada is one of the most used programming languages for the development of software in the critical systems arena. SPARK is a well known subset of Ada, with its own toolset for software verification, that is being increasingly adopted by the industry. Although SPARK has been enjoying industrial (and academic) success, we have found some flaws in its (semi-)formal approach to verification. In this paper we identify some of these flaws and we provide an example of the formal semantics and program logic of SPARK. The example is presented, along with the soundness proof, including the soundness theorem that was adapted to deal with the safety conditions inherent to SPARK.

Keywords: Ada, Formal Methods, Program Logics, Program Verification, Semantics of Programming Languages

1 Introduction

Ada is a highly respected programming language in the critical systems arena and one of the most popular programming languages used for the development of high integrity software (HIS)[2,14,3,15]. It is a full-fledged programming language with concurrency and real-time support built into the core language.

SPARK[4] is a (strict) subset of the Ada programming language with its own toolset for enforcing programming practices and providing program verification capabilities. SPARK achieves this using a semi-formal approach to program verification.

In this introduction we overview the programming languages and their aims and we focus on some of the deficiencies that SPARK has as a formal approach to verification.

The Ada programming language Ada has always been developed with the aim of providing a reliable, readable and safe programming language for use in critical systems, as it was initially commissioned by the DoD1.

1 Department of Defence of the United States of America.
Through the years, the programming language has been revised and expanded several times (Ada 83[11$^2$, 95[24,25], 2005[26], 2012[21,1]), bringing new features to the language while removing (some) ambiguities in the programming language and in its Reference Manual.

Although Ada has been evolving as a general purpose programming language, for the critical systems arena, especially for HIS, strict restrictions are placed upon the features of the programming languages that can be used. These restrictions are designed as ways to avoid common programming errors and to simplify the verification and validation process, which is of the utmost importance in these systems.

The SPARK Programming Language and Toolset SPARK$^3$ takes into account the usual restrictions that are placed upon HIS and implements tools to analyse and enforce (some of) these practices.

It is important to note that SPARK is a strict subset. This means that all (valid) code written in the SPARK programming language corresponds to, not only a valid Ada program, but to a Ada program with the exact same semantics (and independent of compiler implementations). SPARK also supports a subset of the Ravenscar profile for real-time concurrent applications called RavenSPARK.

SPARK advocates the early adoption of the language and tools in the software development process. Late integration in the software development process leads to programmers trying to “sparkify” already existing code, which has been shown to be hard and costly[6].

SPARK promotes a Correctness by Construction approach$^4$ to software development. Correctness by Construction tries to guarantee that programs behave as specified by guaranteeing that each individual component (in the broad sense of the word; also applied to subprograms) is implemented as a refinement of an abstract specification. In SPARK this is implemented by using Programming by Contract in the packages’ body and specification, generating Verification Conditions (VC) for refinement.

The SPARK toolset consists of various tools. In this paper we shall focus on the main tools of the toolset$^5$: Examiner, Simplifier and POGS. The Proof Checker, Zombiescope and SPARKBridge will not be considered in this discussion. Although they are flagship tools they are outside the scope of this article. In Fig. 1 we present the usual verification cycle used with the SPARK tools.

The Examiner[22] is responsible for analysing the syntax and static semantics of SPARK source code. It is also the VC Generator (VCGen) of the toolset[23].

---

$^2$ Hoare criticised the Ada programming language in his Turing Award speech[9]. The version he addressed is what is known as ”preliminary” /”pre”-Ada/Ada 80, the first draft of the language that was revised and extensively changed into the first official standard, Ada 83.


$^4$ Correctness by Construction is also advocated by other formal methods such as the B method.

$^5$ The SPARK version that is considered in this article is SPARK 9 GPL.
The Examiner outputs the VCs into the FDL language which are then supplied to the Simplifier.

The Simplifier is an automated theorem prover tailored for SPARK. It attempts to discharge the VCs written in FDL using its own set of rules, that take into consideration the specific features of SPARK, while additional rules may be added by the user to aid in the proof process.

Problems related to the SPARK approach

Although SPARK has shown itself able to certify critical software up to Common Criteria EAL-5[20] (and EAL-6 in some cases), because it is semi-formal, it can not reach the highest level of certification from Common Criteria, EAL-7[7]. The same can be said for the future DO-178C[16].

To be a fully formal approach, not only should the language be described in a mathematical formalism (both static and dynamic semantics) but also the tools that are used in the toolset should also provide correct and verifiable formal specifications. The implementation of the tools would then have to be proven correct regarding the specifications.

Although there is previous work on the formal semantics[19] (both static and dynamic) of SPARK, it is outdated and does not reflect the actual status of the language. Furthermore, this only addresses the language and not the program logic and VCGen, which are not formalized anywhere in the SPARK literature. Without the availability of such specification, results on the soundness and completeness of the logic can not be given (the same for the VCGen).

Development of a Program Logic for SPARK

In this paper we present our work on the formal specification of a subset of the SPARK programming language, which we call mSPARK\(^6\). Although we have formalized mSPARK’s semantics and program logic and proved the program logic’s soundness (although not for the full mSPARK) using our own adaptation of the soundness theorem, which accounts for safety conditions on range/constraint types, we present here only a small but illustrative part of our work. For the detailed semantics, program logic and proof, please refer to the author’s thesis[5].

\(^6\) The m stands for mini and Minho.
Fig. 1. Use of the SPARK toolset for program verification.
2  mSPARK - A Subset of SPARK

In this section we present our subset of the SPARK programming language, mSPARK. We start by presenting the features that were included in the subset, followed by a small discussion on why these features were chosen. We omit most of the operational semantics because of space constraints, choosing only a representative example. For further reference, this work has been previously presented in[5].

The mSPARK Programming Language  As previously stated, mSPARK is a subset of SPARK. Given this, we present in Table 1 the list of features that can be found in the SPARK programming language and their status in our subset. Items marked with ✓ are fully implemented, items marked with X are not implemented, and items with ± are partially implemented. We also added a column to the table, justifying some of our options.

Most features that are not implemented yet are, for our immediate purposes, not as important as the features we chose. The features that were chosen were based on our experience in developing programs in SPARK and other programming languages and also based on what we deemed to be the most important and interesting features for our first approach at SPARK formalization.

We also present a minimal example of a program written in mSPARK, which has a different syntax from SPARK (albeit very similar) to provide a small “flavour” of the programming language. This program finds the maximum of two numbers.

```plaintext
types

    type Short is range -65536 .. 65536;

variables

    x, y, res : Short;

subprograms

    function Max( a : Short; b : Short ) return Short is
        aux : Short;
        begin
            if a >= b then
                aux := a;
            else
                aux := b;
            end if;
            return aux;
```
end Max;

execute

x := 9001;
y := 0;
res := Max( x, y );

end;

Operational Semantics example The inference rule from Fig. 2 provides a representative example of the inference rules in our operational semantics of mSPARK, portraying the different safety problems that might occur during the evaluation of a command of mSPARK.

The Assign Array inference rule is applied when all arguments of the array assignment are safe, including the indexing and range constraints of mSPARK types. The other two rules, AssignArr\_index\_error and AssignArr\_value\_error, deal with type errors on the ranges of the expressions being evaluated, regarding the declared type for indexes and arrays, respectively.

Fig. 2: Inference rules for array assignment with safety conditions.
Table 1. Features of the mSPARK programming language.

<table>
<thead>
<tr>
<th>Feature</th>
<th>mSPARK</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic commands</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Basic control structures</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Extended control structures</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Basic data types</td>
<td>±</td>
<td>Some types, such as fixed and floating-point and modular arithmetic are not included, for now, in the language.</td>
</tr>
<tr>
<td>Enumeration types</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Range types/subtypes</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Type casting</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Record types</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Subprograms</td>
<td>±</td>
<td>We allow parameterless procedures with frame conditions and pure functions (with and without parameters).</td>
</tr>
<tr>
<td>Named Parameters</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Assertion annotations</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Data-flow annotations</td>
<td>±</td>
<td>We only use data-flow annotations for frame conditions.</td>
</tr>
<tr>
<td>Packages</td>
<td>±</td>
<td>We use a different syntax from SPARK. Also, we do not support some of SPARK’s features.</td>
</tr>
<tr>
<td>Object Orientation</td>
<td>X</td>
<td>An important addition to future iterations of the language. SPARK has only partial support for it.</td>
</tr>
<tr>
<td>Refinement</td>
<td>X</td>
<td>One of the most important features we have on our “wishlist”. The most important part will be to have the axiomatic semantics and VCGen for this feature.</td>
</tr>
<tr>
<td>RavenSPARK</td>
<td>X</td>
<td>Important feature for the future. We intend to have the operational and axiomatic semantics for this subset of concurrency.</td>
</tr>
<tr>
<td>Generics</td>
<td>X</td>
<td>SPARK’s team has been struggling with this feature for some time. We would like to formally specify this in our language.</td>
</tr>
</tbody>
</table>
3 Developing a Program Logic for mSPARK

In this section we present an example of the program logic of mSPARK, the soundness theorem adapted for safety and we show how to prove the soundness of the program logic, by induction on the inference rules of the operational semantics. We start by specifying in Fig. 3 what is to safely evaluate an expression (for a restrict subset of expressions), presenting the definition of the safe predicate for any given environment (an environment contains the context information of the program such as variable declarations, types, subprograms and so on and so forth).

\[
\begin{align*}
\text{safe}(k)(\text{env}) &= \top \\
\text{safe}(x)(\text{env}) &= \top \\
\text{safe}(\mathit{e})(\text{env}) &\iff \text{safe}(\mathit{e})(\text{env}) = \top \\
\text{safe}(\mathit{e}_1 \mathit{op} \mathit{e}_2)(\text{env}) &\iff \text{safe}(\mathit{e}_1)(\text{env}) \land \text{safe}(\mathit{e}_2)(\text{env}) \land \text{compatible types}(\text{type}(\mathit{e}_1), \text{type}(\mathit{e}_2)) \\
\text{where } \mathit{op} &\in \{+,-,\ast,=,\leq,\geq,\neq,\}
\end{align*}
\]

\[
\begin{align*}
\text{safe}(\mathit{e}_1 / \mathit{e}_2)(\text{env}) &\iff \text{safe}(\mathit{e}_1)(\text{env}) \land \text{safe}(\mathit{e}_2)(\text{env}) \land \text{compatible types}(\text{type}(\mathit{e}_1), \text{type}(\mathit{e}_2)) \\
\text{where } \mathit{e}_2(\text{env}) &\neq 0, \text{ where } \text{type}(\mathit{e}_2)(\text{env}) :: \text{integer}
\end{align*}
\]

\[
\begin{align*}
\text{safe}(\mathit{not} \ b)(\text{env}) &\iff \text{safe}(\mathit{b}) \\
\text{safe}(\mathit{b}_1 \mathit{and} \mathit{b}_2)(\text{env}) &\iff \text{safe}(\mathit{b}_1)(\text{env}) \land \text{safe}(\mathit{b}_2)(\text{env}) \\
\text{safe}(\mathit{b}_1 \mathit{or} \mathit{b}_2)(\text{env}) &\iff \text{safe}(\mathit{b}_1)(\text{env}) \land \text{safe}(\mathit{b}_2)(\text{env}) \\
\text{safe}(\mathit{a}(\mathit{e}_1, \ldots, \mathit{e}_n))(\text{env}) &\iff \bigwedge_{i=1}^{n} (\text{safe}(\mathit{e}_i)(\text{env}) \land \text{safe range ind}(\mathit{a}, i, \mathit{e}_i)(\text{env})) \\
\text{safe range}(\mathit{v}, \mathit{e})(\text{env}) &\iff \text{type}(\mathit{e})(\text{env}) :: \text{variable type}(\mathit{v}, \text{env})
\end{align*}
\]

Fig. 3: Safe predicate interpretation.

Then, using the safe predicate definition, we introduce the program logic for array assignment in Fig. 4 and we define in Fig. 5 the meaning of an Hoare triple and its validity.

Soundness is the property that what can be stated in the program logic agrees with the operational semantics. This is defined in Theorem 1.

Theorem 1 (Soundness of mSPARK’s program logic).

\[
\vdash \{ \phi \} S \{ \psi \} \Rightarrow \models \{ \phi \} S \{ \psi \}
\]
\[
\{\phi\} a(e_1, \ldots, e_n) := e \{\psi\}
\]

if \(\models \phi \rightarrow (\text{safe}(a(e_1, \ldots, e_n)) \land \text{safe}(e) \land \text{safe\_range}(a, e))\) and \(\models \phi \rightarrow \psi[a(e_1, \ldots, e_n \triangleright e)/a]\)

Fig. 4: Program Logic for array assignment.

We say a Hoare tripe \(\{\phi\} S \{\psi\}\) is valid iff

\[
\forall env \neq \text{error} \forall env'. [\phi](env) = T \land (S, env) \rightsquigarrow env' \Rightarrow
\]

\(env' \neq \text{error} \land [\psi](env') = T\)

We use the notation \(\models \{\phi\} S \{\psi\}\) to mean that \(\{\phi\} S \{\psi\}\) is valid.

Fig. 5: Validity of Hoare triples.

Finally, we show how to execute the soundness proof for the running example of array assignment.

\[
\{\phi\} a(e_1, \ldots, e_n) := e \{\psi\}
\]

\text{(Assign Array)}

\[
\text{Proof. if } \models \phi \rightarrow (\text{safe}(a(e_1, \ldots, e_n)) \land \text{safe}(e) \land \text{safe\_range}(a, e))\text{ and } \models \phi \rightarrow \psi[a(e_1, \ldots, e_n \triangleright e)/a]
\]

We want to prove \(\models \{\phi\} a(e_1, \ldots, e_n) := e \{\psi\}\), i.e.

Expanding:

\(\forall env \neq \text{error} \forall env'. [\phi](env) = T \land (a(e_1, \ldots, e_n) := e, env) \rightsquigarrow env' \Rightarrow env' \neq \text{error} \land [\psi](env') = T\)

Let \(env \neq \text{error}\) and \(env'\) be environments such that:

(i) \([\phi](env) = T\)

(ii) \((a(e_1, \ldots, e_n) := e, env) \rightsquigarrow env'\)

From (1) and the side condition \(\models \phi \rightarrow (\text{safe}(a(e_1, \ldots, e_n)) \land \text{safe}(e) \land \text{safe\_range}(a, e))\) we know that

(iii) \([\text{safe}(a(e_1, \ldots, e_n))](env) = T\)

(iv) \([\text{safe}(e)] = T\)

(v) \([\text{safe\_range}(a, e)] = T\)

Rule \text{AssignArr\_index\_error} could have not been applied because (iii) guarantees \([k_1, \ldots, k_n] :: \text{type\_array\_ind}(env, a)\).
Rule AssignArr\textsubscript{value\_error} could have not been applied because (iv) guarantees $[\text{safe}(e)] = \top$ and (v) guarantees $k :: type\_array\_val(env, a)$.

So $env' = env[a(k_1 \ldots k_n) \triangleright k]/a$. From (iii), (iv) and (v) it follows immediately that $env' \neq \text{error}$, since $env$ is assumed to be well formed and the substitution $[a(k_1 \ldots k_n) \triangleright k]/a$ is well defined.

From (i) and the side condition $\models \phi \rightarrow \psi[a(e_1, \ldots, e_n \triangleright e)/a]$, follows $[[\psi[a(k_1 \ldots k_n) \triangleright k]/a]](env) = \top$, but, by a typical substitution lemma (that one can shown by induction on $\psi$), this is equivalent to $[[\psi]](env[[a(k_1 \ldots k_n) \triangleright k]/a])(env/x]) = \top$, i.e. $[[\psi]](env') = \top$.

### 4 Related Work

The SPARK toolset now includes the SPARKBridge tool which takes the generated VCs of Examiner and translates them into SMT-Lib and also has native support for CVC3 and Yices. This is from the work of Paul B. Jackson on the study and expansion of Examiner and the FDL output of its VCGen\cite{12,13}, implemented in the Victor tool.

Although this work aims at proving VCs from SPARK, it does not implement its own VCGen. The aim of the project is to use the Victor tool on top of the SPARK tools, disregarding the correctness (or not) of Examiner, and translating Examiner’s output into other widely accepted formats so that other theorem provers can be used.

The Hi-Lite Project from the open-DO initiative has also presented work on deductive verification based on VCs for a SPARK-based subset of Ada\cite{17,18} named Alfa, which intends to be less restrictive than SPARK. Their approach to the VCGen is to translate Ada code (in the Alfa subset) into Why\cite{8} and then to use the VCGen from Why and its interface to the several proof tools it supports, including proof assistants.

Both these works focus on providing translators, either for FDL or for a subset of Ada, and then use existing tools to translate/generate the VCs. In both cases, there is no functional correctness proof of the translator nor for the VCGen implemented in Why.

The approach closest to the work we developed (and are developing) is the work done by Homeier\cite{10} where he proposes the formal verification of the verification tools themselves, although his work is aimed at a While\textsuperscript{7} language.

### 5 Conclusions

We have presented a brief outlook on the research work that we have been developing for a subset of the SPARK language. This work provides a firm

\textsuperscript{7} While languages are a staple of books on theories of programming languages.
theoretical background on the formal specification of a subset of a programming language aimed at the development of High Integrity Software and for the tools that may be developed for supporting this language and even full SPARK and Ada.

Although the work presented here is directed towards a very specific language, the results can be adapted for programming languages with similar features. Furthermore, the example presented in this paper hints on how semantics and program logics can be adapted to deal with safety restrictions and how to prove the soundness of a program logic in relation to its operational semantics counterpart.

6 Future Work

The program logic for the subset presented on Section 2, although completely specified, is still lacking the soundness proof for the whole subset (in [5] we provided the proof for a large set of constructs), with subprograms being the most important feature that has been left out. Work on specifying a VCGen has been carried out but it is also missing soundness and completeness proofs and has incomplete support for all features of mSPARK. Future work will tackle these theoretical aspects on the formal specification of mSPARK.

Embedding the mSPARK programming language and program logic in a proof assistant, thus providing a framework based on mechanical verification, is one of the most important directions that future work should address. Mechanical verification is less prone to errors in proofs and work on rigorous software development would benefit greatly from having a formally verified workbench for program verification.

References

22. SPARK Team: SPARK Examiner: The SPARK Ravenscar Profile (January 2008)
23. SPARK Team: Generation of VCs for SPARK Programs (February 2009)