Bringing SPARK to C developers

Authors: Johannes Kanig (AdaCore), Quentin Ochem (AdaCore), Cyrille Comar (AdaCore)

Abstract

Selecting a language in a safety critical application is often a choice dictated by constraints beyond bare technical merits. Availability of tools or internal resources at the time of decision is often critical. However, once such a choice is made, it is extremely difficult to revert. This is very visible in domain such as avionics or automotive where code bases are sometimes created and maintained over decades. Rewriting software is just not an option.

As such, many software teams live with technical choices that can't be questioned, or marginally. This is notably the case in the world of the C programming language. Its defects are well documented and have been known for many years. An entire sector of the tool industry is focused on developing workarounds in the form of code analyzers, coding standards or auto-testing tools. Other languages and environments are known to provide results at lower cost. However, the barrier of entry, software re-writing, is often beyond what is industrially acceptable.

In this paper, we will discuss one of these alternatives, the SPARK language. We will describe a framework that allows to gain direct benefits from early investment phases and we will discuss supporting tools currently under development.

1. Introduction

Programming languages are usually designed in terms of how well they can express executable semantics and software architecture. They are seldom designed to optimize behavior regarding specification and verification. This explains how difficult it is for a static analysis tool to analyze a piece of C code without either reporting a lot of false alarms or missing a lot of problems. More modern languages such as Ada tend to exhibit better results – although still not entirely satisfactory.

The SPARK language [1] was designed the other way around – with static verification in mind from the beginning. In order to avoid reinventing an entire development environment, it was decided to use the syntax and the executable semantics of an existing language not too far off the overall goal. In this case, the Ada 2012 language. However, all functionalities that were not congruent with static analysis objectives have been removed. This includes in

particular so called access types (pointers) and exception handlers. To this foundation, static semantics have been superposed to some of the existing language notations, notably contracts and assertions. A number of additional annotations (pragmas and aspects) came in to complete the picture.
With the above in mind, a SPARK program can take advantage of state-of-the-art proving technologies (The Why3 program verification platform [2], SMT-solvers such as CVC4 [3], Altergo [4] and Z3 [5], manual provers such as Isabelle [6] and Coq [7]) and be formally verified regarding various aspects such as data coupling, absence of run-time errors or functional correctness. When formal methods are not usable, dynamic verification can still be used to verify most of the SPARK annotations, through regular testing and assertion checking.
Taking full advantage of SPARK however requires making the choice of designing the software with that language very early on. There are various examples in the recent history of teams that have successfully done so, but such opportunities of rewrite are merely an exception. In practice, most developments are done in full Ada or C, and this cannot be changed after years of development, because of the high cost of rewriting existing code. So many projects are stuck with a less-than-ideal choice made at the very beginning of the project, which can't be changed.
Another problem is that, in projects where a custom processor is used, C is often the only language for which a compiler is provided.
There is a solution. In this paper, we describe two ways to make a project adopt SPARK in small steps, getting benefits for all investments. We also propose a solution for the case where only a C-compiler is available.

2. Current Situation of Many Projects in the Industry

The default programming language used in the high integrity embedded industry today is and remains the C language. There are many reasons for that - C is a relatively simple language, the resulting code is fairly efficient, there is usually no abstraction layer between the code generated from the compiler and the execution hardware, many libraries and off-the-shelf components are available, many trained developers can be found on the market place, tools are fairly comprehensive and supported on almost every single hardware platform. In many situations, it's also the default choice, that is a choice that does not get questioned by management.
Unfortunately, C is also a language with a large list of very well documented flaws and vulnerabilities [8]. Various tools and techniques have been developed as an attempt to workaround and get this flaws under control, from language subsets (such as MISRA) to a wealth of static analysis and testing tools. The need to master these tools and techniques however reduce some of the benefits mentioned before (in particular the availability of trained developers).
The predominance of C in the current industry is such that it is not reasonable to expect a shift in practice. The size and longevity of existing components is such that rewriting them is not economically feasible. C is and will remain a strong actor no matter what. Interestingly, the same is true for Ada. Many systems developed in the 80' and 90' were done in Ada, the

default language for military applications at the time, since it was mandated by the DOD. Although the attractiveness of the technology temporarily decreased at the turn of the millennium, the user base in the A&D domain stayed strong and consistent. This is probably thanks to the same mechanics and trends that it is possible today to find not only a strong Ada toolset and environment, but means to take advantage of it even in the context of a strongly C-based culture.

Another trend should also be emphasised: multi-language programming. It's now extremely common to see projects developed using a multitude of different languages. In a single executable, it's not rare to find C linked with C++ or Ada, sometimes interacting with virtual machines running Java, Python or C#. To a software architect, the preexistence of an appropriate component often matters more than the language in which said component is initially developped. Not to mention migration paths, where developers want to migrate from X to Y technology, while keeping legacy components developed in X. In effect, it's common to only migrate new components while still taking advantage of years of previous development.

To summarize, if a different language such SPARK should be adopted, it must be able to interoperate with existing languages, and in particular with the C language.

3. Short Presentation of SPARK

SPARK is more than a simple programming language - it is also a specification language and a verification system. We will explain these aspects here, as they are essential to the understanding of the remainder of the paper.

3.1. SPARK - the Programming Language

SPARK is a programming language that has been designed from the beginning with safety, maintainability and verifiability in mind. The new starting point was the Ada programming language, from which a number of features have been removed which are considered dangerous in safety-critical programming. The most important features that have been removed are pointers and exception handling. However, a recent major redesign of the language has been based on Ada 2012 [9], the latest version of the Ada language, which contains many features that are particularly useful for safety-critiical programming, such as contracts.
Ada is a very rich language and SPARK inherits many of the powerful features of Ada. The rich type system allows to specify specify ranges for integer and floating-point variables, which is much more precise than simply using the predefined word sizes. Array types are much safer and more powerful. In particular, they do not require the use of pointers, and the length of an array can be queried from it. Common array operations such as initialization, copying, slicing and concatenation are built-in.

3.2. SPARK - the Specification Language

However, SPARK is more than a programming language, it also includes a powerful and expressive specification language, which allows to specify the behavior of the program. The most common specifications are attached to functions and are able to describe the behavior of the function in great detail:

- The *Global* specification describes which global variables are accessed by this function, and whether they are read, written, or both. The same property for parameters is already covered by regular Ada syntax.
- *Pre- and Postcondition* (which already exist in Ada 2012) are boolean expressions that must be true at the beginning and the end of the function, respectively. Together they express what the function *requires* to work (the precondition) and what it *guarantees* when returning.

At a larger scale than a single function, other specification features exist, such as abstract state, which allows to group together the state of some package into a single logical variable, and elaboration/initialization properties, which allow to state the properties of data that is set-up only once at the beginning of the program.

It should be noted that all specification features are completely optional.

3.3. SPARK - The Verification System

Ada (and SPARK) allows very easy checking of safety properties such as division by zero, arithmetic overflow and buffer overflow, simply by providing run-time checks. For example, for division by zero, before carrying out the division, it is checked that the divisor is different from zero. If not, the program stops and an error is reported. This feature can be enabled or disabled using a compiler switch, and it is already very useful to find silly errors during testing.

In a similar manner, most of the specification features of SPARK can be checked when the program is run, e.g., during testing. For example, pre- and postconditions are handled like assertions; with a simple compiler switch, they can be compiled into the binary and will be checked during the run of the program.

In addition to this, the SPARK language comes with verification tools that take a SPARK program with specifications and check it for errors statically (that is, without running the program). There are a number of different individual verifications done by the tool. It checks that:

- All variables are properly initialized before they are used;
- Every function only reads and writes the specified global variables and parameters;
- No so-called run-time errors such as division by zero, arithmetic overflow or buffer overflow can occur;

- When an function is called, its precondition is true, so that the function is called in a valid state according to its specification;
- When a function returns, its postcondition is true so that the function returns in a valid state according to its specification.

When applied full-scale, the verification of SPARK can guarantee that no errors of the above kind appear in the program.

The SPARK toolset is based internally on the Why3 verification platform, and various SMT solvers such as CVC4, Alt-Ergo and Z3.

The remainder of this paper will argue that SPARK is useful even when not used everywhere with full verification, but only one or two aspects of SPARK are integrated into the development process.

4. A mixed SPARK-and-C environment

The SPARK language is designed to perform unit verification. In other words, it proves correctness of a subprogram (function) according to its own specification and the one of its dependencies (callees). One can prove a subprogram assuming its precondition, verify that the precondition of its dependencies, assuming that the postcondition of its dependencies holds, and finally proving the postcondition. There are two strong assumptions that need to be verified – the subprogram precondition and the postconditions of the dependencies. In a perfect world, all of those are also verified through formal methods, but this is not a requirement of the language. Traditional testing can also be used as an alternate means of verification.
From a practical point of view, SPARK declarations (ideally with contracts, but this is not strictly required) for all called functions are all that is needed for the SPARK tools to carry out full verification of a given SPARK function. For the called functions, other implementation languages can be chosen, such as C. The SPARK verification results are correct, provided that the other functions correspond to their specification.
Several different use cases for this technique are possible. In many projects, legacy C code exists and it would be undesirable to rewrite it; but new developments should happen in some stricter and safer environment such as SPARK. Often, device drivers for some component of the system are written in C. Or maybe only the most safety-critical component of a system shall be written in SPARK, and other components implemented in C. Another approach could be to progressively migrate the logic of the application in SPARK while keeping standard components such as drivers, libraries and OS written in C. For any such use case, using the above technique, it is possible to use SPARK as the implementation language only for the components where this is desired. The full SPARK specification and verification features can still be used on this part of the project.

```
typedef struct {
  void** data;
  int pointer;
  int len;
  int capacity;
} t_queue, *pqueue;

pqueue init_queue(int capacity);

void* queue_pop(pqueue q);

void queue_enqueue (pqueue q, void* elt);

bool queue_is_empty(pqueue q);

int queue_length(pqueue q);

void free_queue(pqueue q);
```

Fig. 1: The C interface of a simple queue implemented using a ring buffer.

Of course, the verification of the component(s) written in SPARK is only valid if the assumptions on the other components, which have been used to complete the SPARK verification, indeed implement their contract [10]. As the SPARK tools can only be applied to SPARK programs, other verification methods must be applied. Thanks to executable contracts, at least the contracts specified at the boundary between the SPARK and C code can be very easily verified using testing.
Overall, this approach provides a very pragmatic path to migration with quick return on investment. As soon as one subprogram is converted, migrated and proven correct, the overall safety of the application gets improved.

5. SPARK as a language for C specification

But the SPARK language can be useful even when the verification features and the programming language are not used. Verification is always done according to a specification, and the C language is very weak at expressing specifications. As such, and as we have shown in the previous section, the SPARK language has a specification semantics much richer than most programming languages, in particular compared to C. This includes some well-known Ada features (strong typing, parameter modes…) Ada 2012 additions (pre/post conditions, quantifiers…) and SPARK complements (data coupling, states…).
So one may want to use the SPARK specification language to specify C programs. This sounds like a surprising idea at first, but it is entirely possible and useful, as we will show in this section. We will also show an example of this use case at work.

```
type data_arr is array (int range <>) of Address;
subtype data_arr_constr is data_arr (int range 0 .. int'last);
type data_arr_access is access data_arr_constr;

type t_queue is record
   data : data_arr_access;
   pointer : aliased int;
   len : aliased int;
   capacity : aliased int;
end record;

type pqueue is access all t_queue;

function Is_Empty (Q : pqueue) return Boolean is (Q.len = 0);

function To_Arr (Q : pqueue) return data_arr is
 (if Q.pointer + Q.len <= Q.capacity then
   Q.data (Q.pointer .. Q.pointer + Q.len - 1)
  else
   Q.data (Q.pointer .. Q.capacity - 1) &
   Q.data (1 .. Q.len - (Q.capacity - Q.pointer)));

function Pop (A : data_arr) return data_arr is (A (A'First + 1 ..  A'Last));

function First (A : data_arr) return Address is (A (A'First));

function init_queue (cap : int) return pqueue with
  Post =>
   Is_Empty (init_queue'Result) and init_queue'Result.Capacity = cap;

function queue_pop (Q : pqueue) return Address with
  Pre  => not Is_Empty (Q),
  Post => Q.len = Q.len'old - 1 and
          Pop (To_Arr (Q)'Old) = To_Arr (Q) and
          queue_pop'result = First (To_Arr (Q)'Old);
```

Fig. 2: The corresponding SPARK interface with contracts.

The key idea is add a SPARK wrapper to a C function, with identical parameter profile, and redirect all calls to this C function in the program to call the wrapper instead. If the wrapper has SPARK contracts such as Pre- and Postconditions, these contracts get dynamically checked on every call. If the contracts are complete (i.e. capture most if not all of the requirements), this provides a great way to check the requirements of the software during testing in a very explicit yet convenient way.

In fact, much of the mechanical part of this outlined procedure can be automated: the --fdump-ada-spec option of gcc [11] can be used to generate a SPARK wrapper function with an identical parameter list, and the usage of Import and Export pragmas can achieve the rerouting of the calls. Compiling and linking everything together, and enabling assertions when the SPARK-part is compiled, will achieve the desired contract checking.

The idea here is to use C as the language for implementation, and SPARK as the language for specification. Said otherwise, in this mode we only use the “specification language” aspect of SPARK.

Of course, in this case, it is not possible to formally verify a C implementation against a SPARK specification, although the integration with some existing C verification techniques relying on similar technologies (integrated with Why3) could be envisioned at some point. What we are focussing on here is the executable semantics of SPARK specification, which can be checked at run-time. In other words, in this mode, it is possible to compile a hybrid C/SPARK application, and activate specification verification during execution.
This can have various benefits. One is during unit tests, to identify inconsistent calls or invariant breaches early on. It is also a great way to strengthen stubbing and verify that stubs are called with the same constraints as actual code, thus reducing the number of software to software integration errors.
If carefully placed, it can also be kept in final deployment, as a way to protect a component from misuse. The executable specification then acts as a barrier to users, making sure that invariants and assumptions are respected at run time. That is also a nice place to exhibit all the verification code that would otherwise end up being developed as defensive code. One direct application of that can be to contribute to the argumentation of Freedom From Interference [12] as required by ISO-26262 [13].
We now proceed to the promised example. It stems from a small C program which is part of the Why3 platform. The code implements a simple queue with enqueue and pop functions (see Fig. 1). The corresponding SPARK interface (see Fig. 2) has been generated with the `--fdump-ada-spec` switch of gcc, and then manually modified later to improve the mapping. For example we us a few tricks to use SPARK arrays instead of access types to represent the queue data. The SPARK contract uses a "model function" called `To_Arr` to map the queue structure, which wraps around the array in the implementation, to a plain array with elements in the right order for specification purposes. The contracts can are always expressed on that plain array instead of the complex actual data structure, and can now be expressed using straightforward SPARK array operations such as concatenation and slices. For example, the `queue_pop` function has a postcondition which states that the queue after the pop (simply named `Q`), mapped to the plain array, gives the same result as mapping the queue *before* the pop (using the syntax `Q'Old`) to the plain array, and then removing the first element. Looking carefully at the contract of `queue_pop`, one can see that in fact the complete desired functional behavior has been described using the contract.
The reader might ask if the same cannot be achieved using simple assertions at the beginning and end of the C function. But as the example below shows, such a complete contract is difficult to express in C because of the lack of high-level language features such as array concatenation and slices.
The `enqueue` function has not been specified, but this is not a problem, Using this approach, one can selectively apply it to only the desired functionality of a package.

6. Back to the C

So far, we've assumed that the final application was a mixed of SPARK and C files compiled and linked together. This requires the availability of both an Ada 2012 compiler and a compatible C compiler. While the number of platforms supporting Ada 2012 is quite large, it is often the case that there is no compiler available for more very specific or custom targets.

The absence of such technologies may render the whole discussion above futile for projects targeting such platforms. However, virtually all platforms come with a C-compiler.
There is an alternative to this, which is to consider the C language as an intermediate representation in the SPARK code generation chain. In other words, compile SPARK to C with a “standard” toolchain, and then C to assembly with the specific target compiler, which acts as a back-end in this context. It is important to realize that generated C code is not intended to be readable or modifiable. Although MISRA properties can be enforced during code generation as to ensure optimal portability and safety of the generated C layer, the SPARK and C programming languages are too far to generate C that would look like something written by a human being. In particular, the SPARK to C generator may go through expansion or optimization phases, or make choices to translate high-level SPARK concepts into low level C concepts in a way that is not efficiently manageable by a developer. But that is not a problem: just as one would not modify the assembly generated from a compiler except in very specific cases, one should not worry about modifying this C code – which in effect acts here as a universal assembly language.
Going this route is more than just generating C. Of course, all the benefits of SPARK, such as strong typing, runtime checks, static analysis and formal verification are still present.
Having a regular Ada cross compiler has still significant advantages over the SPARK to C generation path. Among other things, it simplifies the integration with tools such as debuggers and makes tool vendor validation easier. However, using the C language as an intermediate language here allows to virtually provide a universal SPARK compiler.

7. DO-178C certification considerations

The various tools and technologies presented so far still need to be properly articulated around DO-178C to be usable in certified avionics context. This implies both references to the certification objectives that can be targeted and corresponding qualification or certification material.
Looking at the SPARK language, its contributions to certification credits start at the planning phase on DO-178, targeting activities such as 4.4.1.a “software development should be chosen to reduce its potential risk to the software being developed”. SPARK is exempt of a number of programming vulnerabilities without the need of any additional coding standard or tooling. When looking at other potential credits, 6.3.4.f comes to mind - “accuracy and consistency”. SPARK can help demonstrating absence of a number of problems identified there, such as fixed point arithmetic overflow and resolution, floating point arithmetic, used of uninitialized variable and unused variables. To achieve this specific objectives, the verification toolchain associated with SPARK needs of course to be qualified TQL-5.
If data flow is specified in SPARK, the verification of its implementation as requested by 6.3.4.b can also be automatically verified by the TQL-5 qualified verification chain.
Going one step further, SPARK can be taken advantage of following methodologies experimented by Airbus [14] and described in the formal proof supplement DO-333. In this case, formal proof can be used to replace some or all of the low level testing. This is assuming both a higher level of tool qualification (TQL-4) together with an argument

demonstrating that the object code is indeed correctly translated from the source code (so called “preservation of properties”).
Regarding the last technology presented, the SPARK-to-C compiler, it’s important once again to realize that in this configuration, the C language is merely an intermediate step in the compilation process. In other words, the generated C code should not be considered as source code by DO-178 definitions, but rather as an intermediate representation within a toolchain. As a result, the source code is SPARK, and the aggregated {SPARK-to-C, C-target} compiler should follow the same recommendations and objectives as other compiler. For example, element to stored in version control is the SPARK source code, structural code coverage has to be achieved at the SPARK level, a source to object traceability study has to be performed at level A, etc.

8. Conclusion

We’ve demonstrated in this paper that the entry barrier to technologies such as SPARK can be lowered significantly. SPARK interoperates well with C programs, so that usage of the SPARK language can easily be limited to new code or particularly critical code at first. Or one can add SPARK contracts to C code without any SPARK implementation, and still benefit from a powerful language to express requirements and runtime checking. Finally, thanks to the SPARK-to-C technology, SPARK can be applied even in a context where no Ada-2012-Compiler is available for the selected target.
Technologies to support these methodologies are being developed. Others tools could have been mentioned as well and be described in other papers – notably the role of C or SPARK code generation from modeling languages such as Simulink. Overall, these tools and methodologies demonstrate not only the feasibility but the low risk associated to an iterative path of experiment and deployment.

References

[1] SPARK, http://www.spark-2014.org
[2] Why3 verification platform, http://why3.lri.fr
[3] CVC4, http://cvc4.cs.nyu.edu/
[4] Alt-ergo, http://alt-ergo.ocamlpro.com/
[5] Z3, https://z3.codeplex.com/
[6] Isabelle, https://isabelle.in.tum.de/
[7] Coq, https://coq.inria.fr/
[8] Paul E. Black, Michael Kass, Michael Koo & Elizabeth Fong, *Source Code Security Analysis Tool Functional Specification*, NIST, 2011)
[9] Ada 2012 Reference manual, http://www.ada-auth.org/standards/ada12.html
[10] Johannes Kanig, Rod Chapman, Cyrille Comar, Jérôme Guitton, Yannick Moy, Emyr Rees: Explicit Assumptions - A Prenup for Marrying Static and Dynamic Program Verification. TAP 2014: 142-157

[11] Emmanuel Briot - AdaCore Gem 59: Generating Ada Bindings for C Headers, http://www.adacore.com/adaanswers/gems/gem-59/
[12] Handbook for Functional Safety, Software Partitioning Edition, JASPAR H-FSS-07-0002, 2013
[13] Road Vehicles - Functional Safety, ISO 26262-1, 2010
[14] Frank Dordowsky, *An experimental Study using ACSL and Frama-C to formulate and verify Low-Level Requirements from a DO-178C compliant Avionics Project*