SPARK by Example: an introduction to formal verification through the standard C++ library

Léo Creuse
ISAE-SUPAERO, Université de Toulouse, France
leo.creuse@student.isae-supaero.fr

Joffrey Huguet
ISAE-SUPAERO, Université de Toulouse, France
joffrey.huguet@student.isae-supaero.fr

Christophe Garion
ISAE-SUPAERO, Université de Toulouse, France
christophe.garion@isae-supaero.fr

Jérôme Hugues
ISAE-SUPAERO, Université de Toulouse, France
jerome.hugues@isae-supaero.fr

ABSTRACT

This paper presents *SPARK by Example* [10], a guide for people wanting to get involved in formal verification of SPARK programs. *SPARK by Example* is inspired by *ACSL by Example*, a similar effort for C/ACSL programs, and provides detailed specification, implementation and proof of classic algorithms (array manipulation, sorting, heap etc). A comparison between ACSL and SPARK is done in the light of proof performance and ease of use.

KEYWORDS

formal verification, SPARK 2014, guide

ACM Reference Format:
Léo Creuse, Joffrey Huguet, Christophe Garion, and Jérôme Hugues. 2018. SPARK by Example: an introduction to formal verification through the standard C++ library. In *Proceedings of ACM SIGAda's High Integrity Language Technology International Workshop on Cyber-Security Interaction with High Integrity (HILT 2018).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Software security is a never ending concern, in particular for the years to come with the rapid growth of the Internet of Things for instance. Despite their best efforts to find the safest and most secure algorithms to build their software, developers can only trust their code if they are sure it will not encounter a runtime error and behave as specified in the design phase. This may be accomplished by using intensive testing, with a formal proof of the expected properties, or both. Verifying software with the help of formal methods can give programmers complete confidence in their code. Even if cybersecurity experts are mainly using so-called lightweight formal methods [28], formal verification techniques can be used to verify the correctness of high security software such as cryptography libraries [6, 25].

Formal verification techniques for imperative languages are well-known since the 70's [12, 15, 18] and with the advances in automatic theorem proving, they may now be used for real industrial cases. However, they remain quite complicated to get involved in: first, theoretical background may not be known by developers and second, developers may write their program in a language that is not well-fitted for such task. To help integrate formal verification in the development workflow, some languages have implemented toolsets that synergize with the rest of the programming environment. As such, verifying C programs with ACSL and Frama-C [9, 20] or SPARK programs with GNATprove [3] is relatively easy once the developer knows how to annotate code.

There is still a big learning curve to be able to use ACSL/Frama-C or SPARK/GNATprove on complex programs and developers need to be able to train on examples with gradually increasing difficulty. The aim of this paper is to present a new guide for SPARK 2014, *SPARK by Example* [10]. *SPARK by Example* is greatly inspired from *ACSL by Example* [8], a similar guide for the ACSL specification language. This guide will present classic algorithms from the Computer Science literature and how to specify, implement and prove them using SPARK 2014. Notice that this work has been performed by two undergraduate students, with no previous knowledge of formal methods nor Ada or SPARK programming.

The paper is organized as follows. Section 2 presents the SPARK 2014 language and toolset, and why the underlying work for this paper is necessary; Section 3 gives an overview of the algorithms proved in *SPARK by Example*; Section 4 explains the major differences between SPARK and ACSL; Section 5 compares SPARK and ACSL/Frama-C. Finally, Section 6 is dedicated to a conclusion.

2 LEARNING HOW TO DEVELOP AND PROVE PROGRAMS IN SPARK 2014

SPARK is both a formally analyzable subset of the Ada 2012 language and a set of tools that enable the user to ensure the integrity and correctness of programs through different levels (see [4] for a complete example). First, as SPARK is a subset of the Ada language, the user has access to most of the features making Ada a reliable and expressive language, such as strong typing system preventing potential conversion errors or a real implementation of arrays. SPARK also removes aspects of Ada that can be the root of software defects, such as Access types (the equivalent of pointers in C). Second, SPARK and its associated tool GNATprove allow the user to use flow and information analysis to get rid of errors such as reads from uninitialized variables or interference between parameters and global variables. Third, SPARK and GNATprove allow

HILT 2018, November 2018, Boston, Massachusetts USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

proof of the absence of run-time errors (AoRTE) like overflows or index checks when using arrays. Finally, SPARK allows the user to write contract-like formal specifications [22] for programs with the Ada 2012 syntax. These functional contracts can then be proved with the help of GNATprove. Notice that all these properties can also be checked at runtime if the program is compiled with the corresponding option (except flow analysis).

When using SPARK and GNATprove at their full capacity to develop a function or a procedure, the workflow is typically the following: write the function in SPARK, annotate it with preconditions and postconditions to specify its behavior, write loop invariants and assertions in the function body, use GNATprove to generate *verification conditions*, i.e. mathematical properties corresponding to the AoRTE and functional properties to be proved on the code, and finally prove them with automatic SMT solvers like CVC4, Z3 or Alt-Ergo[1]. By practising a little, this workflow comes naturally, but there is still an important learning curve if the developer has no previous experience in formal verification.

There are two main resources available to the community to learn coding and verifying programs with SPARK: the *SPARK 2014 user's guide* [3] and *Building High Integrity Applications with SPARK* [19]. The former requires familiarity with Ada and some previous knowledge on formal verification and the latter focuses on programming rather than verifying with SPARK. There is thus still a need for a "recipe" document that shows how to develop and prove SPARK programs through examples that should be classic Computer Science algorithms. Such a document would have for target audience developers with knowledge of imperative programming but with no previous knowledge of formal verification and provide detailed explanations on how to prove programs. Notice that two new online courses [1, 2] from AdaCore are now available and that they can be complemented by our document.

C programmers also have a specification and verification language, ACSL [5], and a platform for deductive verification, namely Frama-C [9, 20][2]. Burghardt et al. have described in *ACSL by Example* [8] the implementation and proof of algorithms taken from the C++ Standard Template Library for ACSL and Frama-C users. *ACSL by Example* is a good starting point for ACSL users as the algorithms used in the document are normally known by developers, specifications and hard technical details are clearly presented and all sources are available to replay the proofs.

Our goal is therefore to write *SPARK by Example*, an adaptation of *ACSL by Example* for the SPARK 2014 language to help beginners. We will also try in the following to emphasize the differences between ACSL/Frama-C and SPARK/GNATprove and to present pro and cons from both worlds.

SPARK by Example is available on [10]. The repository provides all SPARK sources and corresponding makefiles. Textual explanations and code snippets are given as navigable web pages to facilitate user interaction.

[1]The last step of the workflow, i.e. the interaction with the solvers to prove the verification conditions, is also managed by GNATprove

[2]Notice that Frama-C also provides other verification techniques, such as abstract interpretation for instance.

3 THE ALGORITHMS

The algorithms presented in *ACSL by Example* are all extracted from the C++ *Standard Template Library* (STL) [16, 24]. STL provides C++ programmers with standard generic algorithms, such as binary search or sorting algorithms. Algorithms are classified in *ACSL by Example* and therefore in *SPARK by Example* as follows:

- the first chapter deals with non mutating algorithms. These algorithms do not modify their input data. For instance, `find` returns the first index in an array where a value is located and `count` returns the number of occurrences of a value in an array.
  These algorithms are rather simple, but they serve as a starting point for the reader of *SPARK by Example*. They will thus be very important as they are used to present important information (how to define a contract, ghost functions, loop invariants, how to interpret counterexamples returned by provers etc).
- chapter 2 deals with maxmin algorithms and is a particular subset of non mutating algorithms. Its algorithms simply return the maximum and minimum values of an array.
- chapter 3 is about binary search algorithms. These algorithms work on sorted arrays and therefore have a temporal complexity of $O(\log n)$. The classic `binary-search` is presented in this chapter.
- chapter 4 deals with mutating algorithms, i.e. algorithms that modify their input data. They all are procedures that rotate, copy, or modify the order of elements in arrays to match properties. The algorithms in this chapter generally have two implementations: the first one is usually easier, because the content of the input array is copied in another array; the second implementation is done on the array itself and has sometimes lead to difficulties in the proof process.
  In the previous chapters, there has been no difference between the algorithm specification and implementation in C/ACSL or SPARK, but due to the availability of "real" arrays in SPARK, this is the first chapter in which important differences between *ACSL by Example* and *SPARK by Example* appear.
- chapter 5 on numeric algorithms is a special chapter because it mainly focuses on overflow errors. For instance, when returning the sum of the values in an array or the scalar product of two arrays, overflow errors may occur, particularly if the values are integer ones. Moreover, even if the final result is in the right range, the intermediate results can overflow and lead to an error. It is the only chapter that deals with these kinds of errors so it is a little bit besides the others.
- chapter 6 focuses on one particular data structure, namely the binary heap. It presents a concrete implementation of the classic heap structure as a record consisting of a fixed-sized array and a size attribute. It implements the basic algorithms dealing with heaps (`push_heap`, `pop_heap`) but also other algorithms such as `make_heap` that returns a heap created from an input array, or `sort_heap` that returns a heap of size `0` but with a sorted list inside of it.

- chapter 7 deals with sorting algorithms and is a quick chapter: `is_sorted` checks whether an array is sorted or not, and `partial_sort` partially sorts (with a specific definition) an array.
- finally, chapter 8 presents three classic sorting algorithms: `selection_sort`, `insertion_sort` and `heap_sort`.

In the *SPARK by Example* repository, each algorithms has its own `.ads` file for specification and `.adb` file for implementation. Predicate functions used in algorithms are located in the `spec` directory and lemmas used to help the provers are located in the `lemmas` directory. Each algorithm has also a companion Org file that describes the specification and implementation of the algorithm along with details and references.

4 DIFFERENCES BETWEEN ACSL AND SPARK 2014

In this Section, we will highlight some differences between ACSL and SPARK 2014 and their associated tools when proving algorithms from STL.

4.1 Arrays management

The first and fourth chapters of *ACSL by Example* deal with algorithms manipulating arrays. As SPARK is a subset of the Ada language and ACSL annotates C code, this is a first big difference between the two languages. On the one hand, the common way to manipulate arrays in C is to use the fact that an array identifier is decayed to a pointer to the first element of the array in most expressions. This leads to complex function signatures: for each array manipulated, a pointer to the first element of the array and an integer representing the size of the array are mandatory.

On the other hand, arrays are first-class citizens of the SPARK language and much information about a given array is available through attributes: the length of the array, the first and the last indexes (as array are not necessarily indexed from 0) and the range of values on which the index is defined. Moreover, a syntax for array slices is also available in SPARK: `A (J .. K)` allows the extraction of the subarray starting from index `J` to index `K` from array `A`. Algorithms and specifications are therefore easier to write and read in SPARK.

For instance, in the chapter on non-mutating algorithms, the predicate `HasValue` is defined in ACSL as shown on listing 1. In SPARK, there is only one definition, as shown on listing 2.

Listing 1: Definition of the `HasValue` predicate in ACSL

```
predicate HasValue{A}(value_type * a, integer m,
                      integer n, value_type v) =
  \exists integer i; m <= i < n && a[i] == v;
predicate HasValue{A}(value_type * a, integer n,
                      value_type v) =
  HasValue(a, 0, n, v);
```

Listing 2: Definition of the `HasValue` predicate in SPARK 2014

```
function Has_Value (A : T_Arr; Val : T)
   return Boolean is
      (for some I in A'Range => A (I) = Val);
```

Of course, we should not fail to mention the fact that manipulating slices in SPARK induces more complexity in the proof process, particularly for SMT solvers: using a slice creates a copy of the existing array to reason with and using variables in slice ranges implies universal quantification of these variables, which is usually a serious overload on the solvers. A more detailed example on how to manage complex assertions on slices may be found in the first chapter of *SPARK by Example* with the `Search` function.

4.2 A richer and stronger type system

SPARK benefits from the rich type system of Ada and we can for instance distinguish natural numbers (`Natural` type) from integers (`Integer` type) or strictly positive natural numbers (`Positive` type). These types are bounded in the Ada language and we have to keep in mind that the proof of a function or procedure can fail due to overflow errors. For instance, when trying to access index `J+1` of an array whose indexes are declared as `Positive`, we must ensure that `J < Positive'Last`, where `Positive'Last` denotes the upper bound of the `Positive` type. GNATprove emits a check to be proved for such expressions and if `J` cannot be proved to be less than `Positive'Last`, the check fails. Overflows are also verified for floating point values as well as indexes belonging to their array range (see [3] for more details). This makes writing programs and specifications longer, but ensures absence of runtime errors (AoRTE) if the program is proved. It also generally forces one to be more careful when writing code. The chapter "Numeric" of both *ACSL by Example* and *SPARK by Example* focuses on numeric overflow handling.

When searching for a value in an array A, a classic C program will return a value that is less than the length of A if the first occurrence of the value is at this index in A or the length of A if the value does not appear in A. In SPARK, the user has the possibility to use discriminated records to return an "optional" value as shown on listing 3.

Listing 3: Discriminated record in SPARK 2014

```
type Option (OK : Boolean) is record
  case OK is
     when True =>
        Value : Integer;
     when False =>
        null;
  end case;
end record;
```

With this kind of structure, the `Value` field is only accessible if `OK` is true. Of course, a `struct` could have been used with the same purpose in C, but the use of a discriminated record prevents the access to the `Value` field while the search is ongoing and the searched value not yet found, therefore forbidding the use of a meaningless variable. This also means that while the field `OK` is false, the user does not have to specify a value for `Value`. Of course, using such discriminated records implies the generation of checks to be proved.

A stronger type system is not always in favor of SPARK: in the `partial_sort` algorithm, the use of different data structures in Ada makes the proof more complex. This algorithm takes an array as an input, and builds a heap in order to sort only a part of the array. In

ACSL by Example a heap is used in the implementation. This heap is represented by a simple array and a integer representing the last index of the heap in the array, which simplifies the program as all the data is contained in a single memory location. In *SPARK by Example*, a heap is represented by a `heap` data structure, which is very optimized for manipulating heaps, but there are no operations available in `heap` allowing the manipulation of "additional" data (the rest of the input array). This requires the creation of multiple copies of the input data (the input array, and the heap structure) as pointers are not allowed in SPARK. This also leads to the necessity of an auxiliary function to correctly prove the algorithm. This algorithm can be found in the chapter on sorting of both *SPARK by Example* and *ACSL by Example*.

Of course, one can wonder if the previously described checks added by GNATprove (initialization checks, overflow checks, discriminant checks, array index checks or range checks) do not overload the user. In our experience, the verification conditions generated by these checks are rather easy to prove and, when not proved, help us to properly define forgotten conditions and ranges on our variables. Finally, notice that using the `-wp-rte` option of Frama-C enables the proof of AoRTE in C programs and that this option is used in most examples in *ACSL by Example*, but the runtime errors checked by Frama-C are less exhaustive than the ones checked by SPARK because of limitations of the C language[3].

4.3 No pointers

SPARK does not allow the use of access types (or pointers in C) in order to avoid aliasing, which is difficult to handle when trying to prove AoRTE or functional correctness of a function. In consequence, some algorithms presented in *ACSL by Example* are not directly implementable in SPARK or lose their interest due to a better expressiveness of SPARK.

For instance, `backwards-copy` is an algorithm used in C/C++ when two arrays share one or more locations in memory: one is copied in the other but "from right to left". In some cases, it enables the user to avoid aliasing issues when copying arrays. In SPARK, since two arrays have necessarily two distinct locations in memory, there would be no sense in defining a function or procedure that does this backward copy. Another example is `remove`, an algorithm that removes every occurrence of a value in an array (and therefore, reduces the "length" of the array). The memory management of SPARK does not allow one to change the size of an array when it is initialized, so this algorithm was not implemented in *SPARK by Example*.

4.4 Specifications are executable

The differences between SPARK and ACSL do not stop at syntactical level, but also appear in how the user interacts with the toolset or is helped by the verification tools to ensure that software is free from defects.

SPARK and ACSL differ in their ability to generate runtime checks in the executable from the code and annotations. When compiling SPARK 2014 code, it is possible by the addition of a simple compile option to instruct the compiler to add runtime checks for all the assertions, loop invariants and pre/postconditions. This is of great help when debugging an implementation that does not respect its specification, as some simple program tests can be written to quickly check if an assertion fails. This can also be achieved in ACSL, but requires the use of the E-ACSL plugin [11, 26], which reduces the feature-set of ACSL[4].

Although it is possible to generate runtime checks for both specification languages, the user has to trust the compiler to include runtime checks at the correct place in the executable code. Whereas there are no guarantees for E-ACSL code generation, the SPARK compiler, GNAT, has been proven to correctly include SPARK-relevant runtime checks and to not omit any of them for a small subset of SPARK, therefore guaranteeing that any well-formed terminating SPARK program does not lead to erroneous execution for this subset [29] and this approach could be extended to a significant subset of SPARK.

4.5 Axiomatic definitions

In order to correctly specify the behavior of a program, the user may need to give axiomatic definitions. We will use the example of the `Count` function that returns the number of occurrences of a value `v` in an array `A`. *Count* can be defined inductively. Given an array A of size n and a value v:

- $Count(A, v) = 0$ if A is empty
- $Count(A, v) = 1 + Count(A[0..n-2])$ if $A[n-1] = v$ and $A[0..n-2]$ is the same array than A but without its last element
- $Count(A, v) = Count(A[0..n-2])$ if $A[n-1] \neq v$

Axiomatic definitions can be written directly in ACSL. For instance, listing 4 presents an axiomatization of `Count` given in *ACSL by Example*.

Listing 4: ACSL axiomatic definition of `Count`

```
axiomatic Count {
  logic integer Count{L}(value_type* a, integer m,
       integer n, value_type v)
                reads a[m..n-1];

  axiom CountSectionEmpty{L}:
    \forall value_type *a, v, integer m, n; n <= m
         ==>
            Count(a, m, n, v) == 0;

  axiom CountSectionHit{L}:
    \forall value_type *a, v, integer n, m; m < n
        ==> a[n-1] == v ==>
            Count(a, m, n, v) == Count(a, m, n-1,
                v) + 1;

  axiom CountSectionMiss{L}:
    \forall value_type *a, v, integer n, m; m < n
        ==> a[n-1] != v ==>
            Count(a, m, n, v) == Count(a, m, n-1,
                v);

  axiom CountSectionRead{K,L}:
```

[3] Runtime errors checked by Frama-C are mainly integer overflows, null pointer dereferencing and the validity of some casts.

[4] This plugin was not used for the verification of the functions in *ACSL by Example*, to the best of our knowledge.

```
    \forall value_type *a, v, integer m, n;
        Unchanged{K,L}(a, m, n) ==>
            Count{K}(a, m, n, v) == Count{L}(a, m,
                n, v);
}
```

`Count` is defined as a function that takes an array `a`, two integers `m` and `n` representing the array slice in which the value `v` has to be counted. The `L` parameter indicates that `Count` can be evaluated at different locations (C labels or ACSL logic labels like`LoopEntry` for instance). Axiom `CountSectionEmpty` defines the behavior of `Count` when the slice is empty ($n \leq m$), axioms `CountSectionHit` and `CountSectionMiss` give the inductive property of `Count` and axiom `CountSectionRead` specifies that array `a` has not changed between program location `K` and `L`, then `Count` should return the same result when evaluated at location `K` and at location `L` (`Unchanged` is a logic function defined elsewhere).

Listing 5 presents a definition of the `Count` function in SPARK 2014. This definition is available as an example in Section 7.9 of [3] where `Count` is called `Occ`.

Listing 5: The definition of `Count` in SPARK 2014

```
function Remove_Last (A : T_Arr) return T_Arr
    is
  (A (A'First .. A'Last - 1))
with Pre => A'Length > 0;

function Count_Def (A : T_Arr; Val : T) return
    Natural is
  (if A'Length = 0 then 0
   elsif A (A'Last) = Val then
        Count_Def (Remove_Last (A), Val) + 1
   else Count_Def (Remove_Last (A), Val))
with Post => Count_Def'Result <= A'Length;
pragma Annotate (Gnatprove, Terminating,
    Count_Def);

function Count (A : T_Arr; Val : T) return
    Natural is
  (Count_Def (A, Val))
with Post => Count'Result <= A'Length;
```

All functions are contained in a package with `Ghost` aspect omitted here for readability. Ghost functions or variables are only used for specification purposes and cannot be used in "real" code. Let us also notice that a restriction in SPARK does not allow it to use the postcondition of a recursive function when called from an assertion. We thus need to wrap `Count_Def` inside `Count` to be able to use the postcondition defined in `Count_Def`. As for any recursive function, the termination of `Count_Def` is not supported by SPARK, hence the `pragma` indicating that `Count_Def` terminates.

The definition of `Count_Def` provided in listing 5 is not an axiomatic definition *per se*. `Count_Def` is defined as an *expression function*, i.e. a function without a body but defined with a single expression. The axioms presented before are implemented as `if-then-else` constructs. Therefore, even if axiomatic definitions are not directly available in SPARK, expression functions may be used to define Ada functions that represent such definitions. Notice that ghost imported functions without a body but with contract cases to represent the axiom may also be used to represent such axiomatic definitions, but we have not evaluated such a solution in *SPARK by Example*.

4.6 Getting counterexamples

One of the features of GNATprove is the fact that each time a check (assertion, loop invariant, precondition or postcondition) fails to prove, the tool will use the model generated by the solvers to find, if possible, a counterexample, which can greatly help the user to detect uncovered cases in annotations [17]. Obtaining counterexamples in ACSL is possible only using a subset language from ACSL, E-ACSL, which enables the execution and runtime verification of ACSL contracts, and with the StaDy plugin [23]. Unfortunately, the examples are generated through testing, which can lead to an explosion in the number of cases to cover by testing if the cyclomatic complexity of the program is big .

Although counterexamples are of great help in the process of correctly annotating a program, the generation of these counterexamples is not yet perfect. In some cases the user can be given a spurious counterexample, particularly when the provers are interrupted during their proof trial (see again [17] for more details). In that case, the user has to turn to manual testing in order to find the cause of the proof failure.

4.7 Handling complex proofs

The proof of some verification conditions often requires reasoning on properties that can not be directly handled by SMT or automatic solvers, for instance inductive properties. In this case, the proof may be achieved with a proof assistant like Coq or the new proof assistant embedded in GPS (see Section 7.9.3.5 of [3]), or discharged by an automatic solver guided by *lemmas*. Lemmas are mathematical theorems, possibly with hypotheses, that must be added to the theories available to the SMT solvers to prove the verification condition. Of course, lemmas must also be proved, either using a proof assistant or by an automatic prover.

In SPARK, there is no proper construction of lemmas as in Frama-C. To work around this limitation, the user has to define a procedure and use contract-based programming to write the lemma: hypotheses of the lemma are the preconditions of the procedure, conclusions of the lemma are its postconditions. This "emulation" requires them to be instantiated within the code to prove, whereas lemmas in Frama-C are automatically used by the provers when necessary. The main advantage of the SPARK approach is the fact that the user can help the solver to prove the lemma using an actual implementation of the procedure, whereas some lemmas in *ACSL by Example* have to be proven with Coq when the SMT solvers fail to prove them.

Let us take an example: in *ACSL by Example* lemmas are defined for `Count`. Let us consider the first one stating that the number of occurrences of `v` in `a[0..n]` is the number of occurrences of `v` in the slice `a[0..n-1]` plus the number of occurrences of `v` in the singleton `a[n..n]` as presented on listing 6.

Listing 6: A lemma about `Count` in ACSL

```
lemma
  CountOne:
    \forall value_type *a, v, integer n;
      0 <= n ==>
```

```
      Count(a, n+1, v) == Count(a, n, v) +
                          Count(a, n, n+1, v);
```

In SPARK, this lemma can be expressed as a procedure, as shown on listing 7.

Listing 7: A lemma about `Count` in SPARK

```
   procedure Lemma_Count_Section_One (A : T_Arr)
   with Pre  => A'Length > 0,
        Post => (for all V in T => Count.Count (A,
             V) =
             Count.Count (A (A'First .. A'Last -
                 1), V)
           + Count.Count (A (A'Last .. A'Last), V)
             );
```

Before using lemmas in proofs, they have to be proved. With ACSL, a direct proof of lemmas with Frama-C and automatic solvers or proof assistants can be attempted, but SPARK requires lemmas to have an implementation to prove using automatic solvers[5]. This might seems counter-intuitive, as lemmas represents mathematical properties, but the implementation of a lemma can be viewed as a proof sketch to help the solvers. Implementation of the previous lemma is presented on listing 8. We first define an auxiliary lemma that proves that the main lemma is correct for a particular value `V`. The implementation of the auxiliary lemma is the sketch of the proof: just check the last element of the array and prove that `Count` behaves correctly on the slice of the array only with the last element. This auxiliary lemma is finally used in the implementation of the main lemma.

Listing 8: The implementation of the lemma on `Count`

```
procedure Lemma_One_Single (A : T_Arr; V : T) with
  Pre => A'Length /= 0,
  Post => Count.Count (A, V) =
          Count.Count (A (A'First .. A'Last - 1),
              V) +
          Count.Count (A (A'Last .. A'Last), V);

procedure Lemma_One_Single (A : T_Arr; V: T) is
begin
 if A (A'Last) = V then
  pragma Assert(Count.Count(A (A'Last..A'Last), V)
       = 1);
 else
  pragma Assert(Count.Count(A (A'Last..A'Last), V)
       = 0);
 end if;
end Lemma_One_Single;

procedure Lemma_Count_Section_One (A : T_Arr) is
begin
   for V in T loop
      Lemma_One_Single (A, V);
   end loop;
end Lemma_Count_Section_One;
```

Fortunately, certain forms of "templates" appear when implementing lemmas for proving them, so it becomes easier when using the same predicates. The example above presents a common way to prove a property on every value of a type `T`. The method used here can be adapted to a lot of contexts: a lemma to prove the property on a single value is written (`Lemma_One_Single`), and another lemma is written, using a loop that calls the previous lemma each time, and storing the property for every value before the current one (`Lemma_Count_Section_One`). It has been useful for instance in the chapter on heaps, where a lot of lemmas are used in order to prove the algorithms, but they are all proven in a similar way. Therefore, the only big step is understanding how to help solvers to prove lemmas by decomposing the proof in several procedures. After that, writing lemmas is not overwhelming.

As already written, an important aspect of SPARK lemmas is that they need to be instantiated in order to be taken into account by the provers. Manually inserting the lemmas in the code to be proved might seem tedious, but the output of GNATprove, indicating for each assertion or loop invariant the missing preconditions, is of great help for locating the right placement of the lemmas, or even knowing if a lemma is required at all.

[5]Notice that lemmas may have a `null` implementation and be discharged by a proof assistant.

5 RESULTS

There are two aspects of SPARK that we want to evaluate through the production of *SPARK by Example*: proof performance and user-friendliness. Notice that all results for *SPARK by Example* has been produced using GNAT Community 2018 Linux version on a computer with an Intel Core i7-4810MQ CPU and 16GB of RAM. A timeout of 10 seconds per verification condition is sufficient for all the proofs.

5.1 Proof performance

SPARK by Example and *ACSL by Example* both provide a rich set of algorithms on which we can base a comparison on the respective proof performances of SPARK and ACSL/Frama-C. Of course, all presented algorithms are proved in both projects. To compare them, we have chosen two metrics: the number of verification conditions generated for each algorithm/chapter and the proof success rate from the SMT solvers. The first one gives an idea of the work that has to be done by the solvers. The second one shows if the specification and the implementation proposed in both projects is suitable for automatic verification.

Concerning the first metric, the results presented in Table 1 were obtained by counting all the verification conditions generated from algorithms in the chapter on binary heaps, the one with the most difficult algorithms to prove. These verification conditions include all the runtime checks, such as variable initialization and range checks. Looking at the detailed example of `pop_heap` algorithm on Table 3, such verification conditions represent about half of the total verification conditions and most of them are easily proved, either by flow analysis or by SMT solvers. Table 2 shows the results for the chapter on heaps in *ACSL by Example*. When looking at the number of verification conditions (excluding initializations and runtime checks that are natively available in SPARK/Ada), they are around the same between *ACSL by Example* and *SPARK by Example*. Notice that these results do not represent a comparison of CVC4, Z3 and Alt-Ergo capacities.

algorithm	total VCs	CVC4	Z3	unproved	flow
is_heap_p.adb	30	25	1	0	4
push_heap_p.adb	203	155	3	0	44
pop_heap_p.adb	377	304	2	0	71
make_heap_p.adb	131	101	4	0	26
sort_heap_p.adb	31	29	1	0	1
total	772	614	11	0	146

Table 1: Verification conditions generated for the chapter on heaps in *SPARK by Example*

algorithm	total VCs	Qed	AE	CVC4	Z3	Coq	unproved
is_heap	24	6	18	0	0	0	0
push_heap	96	30	59	3	3	1	0
pop_heap	93	44	45	2	0	1	1
make_heap	34	10	23	0	1	0	0
sort_heap	50	10	38	1	0	1	0
total	297	100	183	6	4	3	1

Table 2: Verification conditions for the chapter on heaps in *ACSL by Example*

analysis	total VCs	CVC4	Z3	unproved	flow
initialization	71	0	0	0	71
check	123	123	0	0	0
assertion	50	49	1	0	0
loop	45	45	0	0	0
precondition	74	74	0	0	0
postcondition	6	5	1	0	0
contract	8	8	0	0	0
total	377	304	2	0	71

Table 3: Verification conditions for the `pop_heap` algorithm

One can wonder if the greater number of verification conditions generated by GNATprove for the `pop_heap` algorithm (377 against 93 for *ACSL by Example*) may not be a serious drawback. First, as stated before, a great part of these verification conditions is easily checked and does not take a big amount of time to be proved. The most difficult verification conditions to prove are the ones about loop invariants or postconditions and contract cases. These are the real bottlenecks in term of proof difficulty for the solvers. Moreover, from our point of view, these extra verification conditions give a lot of confidence in the code, as they cover typical programming faults that should not appear (initialization of local variables etc). Finally, notice that a postcondition of `pop_heap` is not yet proved in *ACSL by Example*, namely the one stating that the resulting array is a permutation of the original one. This postcondition was one of the most difficult to prove in *SPARK by Example* and needed complicated lemmas, which may explain the difference.

On the machine described at the beginning of the Section, the proof of all chapters of *SPARK by Example* with "unlimited" parallel processes takes 11 minutes. The proof of the corresponding chapters of *ACSL by Example* in sequential mode[6] takes 12 minutes. Parallelizing proofs in *ACSL by Example* reduces the global proof time to 8 minutes. Using Memcached [21] with SPARK allows to cache proofs of lemmas heavily used instead of reproving them each time they are needed and hence diminishes the time needed to prove all chapters of *SPARK by Example* to 8 minutes. We can conclude that even if there are more assertions to prove in *SPARK by Example*, the time needed to prove all chapters is roughly the same as for *ACSL by Example*.

Concerning the second metric, we wanted to verify if every verification condition generated by *SPARK by Example* can be discharged by SMT solvers, hence avoiding developers to use manual proof assistants like Coq. This has been effectively verified, as all verification conditions in all chapters are proved by CVC4 and Z3, without the help of Coq. Of course, this implies that lemmas has to be written and instantiated in the code, but our experience shows that writing such lemmas helps to understand better the specification and the implementation of the algorithms. On the other hand, in *ACSL by Example* some lemmas need to be discharged with Coq, hence forcing the user to be familiar with such a proof assistant. Of course, these manual proofs mainly concerns mathematical lemmas that can be reused in many programs, hence the effort. Notice also that we have not tried to use instantiated lemmas in *ACSL by Example* code to verify if manual proof can be avoided in *ACSL by Example*.

5.2 Ease of use of SPARK and GNATprove

The other, and less objective, aspect of SPARK that can be evaluated through *SPARK by Example* is its ease of use, and how beginner-friendly it can be. As mentioned previously, any software developer that wishes to enter the world of software reliability and formal verification should learn some theoretical work (Floyd-Hoare's logic for instance), a specification language, and the associated workflow to prove a program. The work described in *SPARK by Example* was at 90% done by two undergraduate students with no prior knowledge of Ada nor formal verification methods and with only a background in functional programming and imperative programming in C. In the span of three months, they learned the basics of formal verification, translated, proved and documented the proof of around 50 algorithms from *ACSL by Example*. Of course, lots of implementations and proofs are almost directly translated from ACSL/C to SPARK, but some of them, particularly in the chapter on heaps or mutating algorithms, need special care.

For programmers coming with a C background, the first SPARK programs are easy and pleasant to write, mainly due to the efficient type system and array management. Using GNATprove to check AoRTE is also a great advantage as they were not obliged to use manual proof assistant to discharge verification conditions. On the other hand, understanding why counterexamples are "wrong", managing complex quantifiers imbrication in assertion or writing

[6] This means that solvers are used in a sequential pipeline, each solver passing on the next solver the proof obligations it cannot prove.

lemmas may be difficult, but we hope that *SPARK by Example* gives beginners a good starting point to understand these points.

From our point of view, this work shows that formal verification of complex algorithms through SPARK is accessible to a beginner and can be quickly learned, even more with the help and experience of people who went through the learning curve, and documented their tips and tricks in *SPARK by Example*.

6 CONCLUSION

We have presented in this paper as summary and analysis of *SPARK by Example*, a guide for beginners to learn how to specify, implement and prove SPARK programs with GNATprove and SMT solvers. As in *ACSL by Example*, the examples given in *SPARK by Example* are provided by the Standard C++ Library and range from classic non-mutating algorithms to binary heap implementation. Our implementations of the algorithms are all proved with automatic SMT solvers in a reasonable time, both for absence of runtime errors and functional correctness.

Our main objective was not only to produce a complete guide to share our experience with other users, but also to show that formal verification through deductive methods with SPARK/GNATprove is feasible for beginners, as the two main contributors of *SPARK by Example* had not previous knowledge of formal methods nor SPARK before.

Of course, we have several directions to improve and continue this work. First, we may try to compare more accurately SPARK and ACSL by trying to use the same proof techniques in the implementation of complex algorithms like the one on heaps. The greater number of verification conditions generated for some complex algorithms in *SPARK by Example* not only comes from the AoRTE checks provided by GNATprove but also from the use of lemmas to help the automatic provers. We should try to use the same techniques for the algorithms of the chapter on heaps in *ACSL by Example* to objectively compare both approaches. Second, new algorithms may be included in *SPARK by Example*, for instance the work on Red-Black trees done by Claire Dross and Yannick Moy [13]. This may lead to a gallery of certified programs in the spirit of what has been done for the Why3 platform [7, 14, 27]. A future objective would also be to show how to integrate more closely formal specifications and proofs with testing by providing a simple framework to generate test cases from specifications. Finally, a more formal document should be produced with more theoretical explanations on formal methods.

ACKNOWLEDGMENTS

The authors would like to thank Claire Dross and Yannick Moy from AdaCore for their meaningful comments and help on *SPARK by Example*.

REFERENCES

[1] AdaCore. 2018. *Advanced SPARK – online course.* https://learn.adacore.com/courses/advanced-spark/index.html
[2] AdaCore. 2018. *Introduction to SPARK – online course.* https://learn.adacore.com/courses/intro-to-spark/index.html
[3] AdaCore and Altran UK Ltd. 2018. *SPARK 2014's User Guide.* http://docs.adacore.com/spark2014-docs/html/ug/index.html
[4] AdaCore and Thales. 2017. Implementation guidance for the adoption of SPARK. https://www.adacore.com/books/implementation-guidance-spark
[5] Patrick Baudin, Pascal Cuoq, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. 2018. *ACSL: ANSI/ISO C specification language.* https://frama-c.com/download/acsl-implementation-Chlorine-20180501.pdf
[6] Stefan Berghofer. 2017. Development of Security-Critical Software with SPARK/Ada at secunet. (Presented at) Frama-C & SPARK days: Formal Analysis and Proof for Programs in C and Ada. https://frama-c.com/download/framaCDay/FCSD17/talk/09_Berghofer.pdf
[7] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. 2011. Why3: Shepherd Your Herd of Provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*. Wrocław, Poland, 53–64. https://hal.inria.fr/hal-00790310.
[8] Jochen Burghardt and Jens Gerlach. 2018. ACSL by Example. https://github.com/fraunhoferfokus/acsl-by-example
[9] CEA List. 2018. Frama-C. https://frama-c.com
[10] Léo Creuse, Christohe Garion, Jérôme Hugues, and Joffrey Huguet. 2018. SPARK by Example. https://github.com/tofgarion/spark-by-example
[11] Mickaël Delahaye, Nikolai Kosmatov, and Julien Signoles. 2013. Common specification language for static and dynamic analysis of C programs. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC '13, Coimbra, Portugal, March 18-22, 2013*, Sung Y. Shin and José Carlos Maldonado (Eds.). ACM, 1230–1235. https://doi.org/10.1145/2480362.2480593
[12] Edger W. Dijkstra. 1975. Guarded commands, nondeterminacy and formal derivation of program. 18, 8 (1975), 453–457.
[13] Claire Dross and Yannick Moy. 2017. Auto-Active Proof of Red-Black Trees in SPARK. In *NASA Formal Methods*, Clark Barrett, Misty Davies, and Temesghen Kahsai (Eds.). Springer International Publishing, Cham, 68–83.
[14] Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3 — Where Programs Meet Provers. In *Proceedings of the 22nd European Symposium on Programming (Lecture Notes in Computer Science)*, Matthias Felleisen and Philippa Gardner (Eds.), Vol. 7792. Springer, 125–128.
[15] Robert W. Floyd. 1967. Assigning meanings to programs. In *Mathematical aspects of computer science*, J. T. Schwartz (Ed.). American Mathematical Society, 19–32.
[16] International Organization for Standardization. 2011. *ISO/IEC 14882:2011.*
[17] David Hauzar, Claude Marché, and Yannick Moy. 2016. Counterexamples from Proof Failures in SPARK. In *Software Engineering and Formal Methods (Software Engineering and Formal Methods)*. Springer, Vienna, Austria. https://hal.inria.fr/hal-01314885
[18] C. A. R. Hoare. 1969. An axiomatic basis for computer programming. 12, 10 (1969), 576–580.
[19] John W. McCormick and Peter C. Chapin. 2015. *Building High Integrity Applications with SPARK.* Cambridge University Press.
[20] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2015. Frama-C: A software analysis perspective. *Formal Asp. Comput.* 27, 3 (2015), 573–609. https://doi.org/10.1007/s00165-014-0326-7
[21] Memcached community. 2018. Memcached. https://memcached.org/
[22] Bertrand Meyer. 1992. Applying "Design by Contract". *IEEE Computer* 25, 10 (1992), 40–51. https://doi.org/10.1109/2.161279
[23] Guillaume Petiot, Nikolai Kosmatov, Bernard Botella, Alain Giorgetti, and Jacques Julliand. 2016. Your Proof Fails? Testing Helps to Find the Reason. In *Tests and Proofs - 10th International Conference, TAP 2016, Held as Part of STAF 2016, Vienna, Austria, July 5-7, 2016, Proceedings (Lecture Notes in Computer Science)*, Bernhard K. Aichernig and Carlo A. Furia (Eds.), Vol. 9762. Springer, 130–150. https://doi.org/10.1007/978-3-319-41135-4_8
[24] P.J. Plauger, Meng Lee, David Musser, and Alexander A. Stepanov. 2000. *C++ Standard Template Library.* Prentice Hall PTR.
[25] Alexander Senier. 2018. SPARK Cryptographic Library. https://git.codelabs.ch/?p=spark-crypto.git
[26] Julien Signoles. 2018. *E-ACSL: Executable ANSI/ISO C Specification Laguage.* http://frama-c.com/download/e-acsl/e-acsl.pdf
[27] The Toccata team. 2018. Why3. http://why3.lri.fr/
[28] Jeffrey Voas and Kim Schaffer. 2016. Insights on Formal Methods in Cybersecurity. 49, 5 (2016), 102–105. https://doi.org/10.1109/MC.2016.131
[29] Zhi Zhang, Robby, John Hatcliff, Yannick Moy, and Pierre Courtieu. 2017. Focused Certification of an Industrial Compilation and Static Verification Toolchain. In *Software Engineering and Formal Methods*, Alessandro Cimatti and Marjan Sirjani (Eds.). Springer International Publishing, Cham, 17–34.