Teaching Deductive Verification Through Frama-C and SPARK for Non Computer Scientists

Léo Creuse[1], Claire Dross[2], Christophe Garion[1(✉)], Jérôme Hugues[1], and Joffrey Huguet[1]

[1] ISAE-SUPAERO, Université de Toulouse, Toulouse, France
{creuse,huguet}@student.isae-supaero.fr, {garion,hugues}@isae-supaero.fr
[2] AdaCore, Paris, France
dross@adacore.com

Abstract. Deductive verification of software is a formal method that is usually taught in Computer Science curriculum. But how can students with no strong background in Computer Science be exposed to such a technique? We present in this paper two experiments made at ISAE-SUPAERO, an engineering program focusing on aerospace industry. The first one is a classic lecture introducing deductive methods through the Frama-C platform or the SPARK programming language. The second one is the production by two undergraduate students of a complete guide on how to prove complex algorithms with SPARK. Both experiments showed that students with no previous knowledge of formal methods nor theoretical Computer Science may learn deductive methods efficiently with bottom-up approaches in which they are quickly confronted to tools and practical sessions.

1 Introduction

Formal methods are usually taught in Computer Science curriculum where students have a good background in theoretical Computer Science. But how to teach formal methods in a more "hostile" environment, for instance where students only have a minimal background in Computer Science? ISAE-SUPAERO engineering program is such an environment, as students are exposed to a small amount of hours in Computer Science and almost nothing in theoretical Computer Science.

However, as ISAE-SUPAERO is mainly oriented to aerospace industry, we think that introducing formal methods is crucial, at least for students choosing the Critical Systems final year major. In order to do so, we have first introduced a classic course in this major with two tracks on deductive verification: one using Frama-C and its plugin WP to verify C programs and another one using the SPARK programming language and its associated tool GNATProve. These two tracks are taught with different educational methods. The first one uses a classic top-down approach with theory on deductive verification presented before using

B. Dongol et al. (Eds.): FMTea 2019, LNCS 11758, pp. 23–36, 2019.
https://doi.org/10.1007/978-3-030-32441-4_2

tools, whereas the second one uses a bottom-up approach in which students uses SPARK from the beginning of the course and prove more and more complex properties on their programs.

We have also asked two undergraduate students to develop a complete guide on how to use SPARK to prove classic Computer Science algorithms (algorithms on arrays, binary heaps, sorting algorithms). This is inspired by *ACSL by Example*, a similar guide developed for the ACSL specification language for C programs, hence the name of our guide, *SPARK by Example*. The development of *SPARK by Example* showed us some pitfalls and difficulties encountered by beginners and we hope that the resulting document may help to understand how to specify and prove SPARK programs.

This paper is organized as follows. Section 2 presents ISAE-SUPAERO engineering program and why it is difficult to teach formal methods in this context. Section 3 presents a last year course on formal methods and focuses on tracks on deductive verification through Frama-C and SPARK. Section 4 presents *SPARK by Example*, a complete guide on how to prove 50 algorithms taken from the C++ STL. Finally, Sect. 5 is dedicated to conclusion and perspectives.

2 Why Is It Difficult to Teach Formal Methods at ISAE-SUPAERO?

2.1 The ISAE-SUPAERO Engineering Program

ISAE-SUPAERO is one of the leading French "Grandes Écoles" and is mostly dedicated to the aerospace industrial sector even if half of its students begin their career in other domains (research, energy, bank, IT, ...). Before entering ISAE-SUPAERO, students are selected through a national competitive exam for which they attend preparatory classes during two years. The ISAE-SUPAERO engineering program [14] is a three-year program during which students acquire common scientific and non-scientific background on aerospace industry and also choose elective courses to prepare themselves for their career. The three years are split into 6 semesters whose content is presented on table 1.

Table 1. Content of semesters in ISAE-SUPAERO engineering program

Semester	Content
S1	Common core
S2	Elective courses & projects
S3	Common core
S4	Elective courses & projects
S5	Field of application (140 h) & Major of expertise (240 h)

Figure 1 show the ratio of Computer Science oriented courses in the S1 and S3 common core courses. There are only three Computer Science courses:

– A 40 h course on Algorithm and Programming in S1. This course focuses on basic algorithms, classic data structures (linked lists, binary search trees, graphs) and C is the associated programming language. No formal methods are discussed during the lecture, but algebraic specifications are used to specify data structures.
– A 40 h course on Object-Oriented Design and Programming in S3, focusing on object-oriented design principles and programming in Java.
– A 10 h course on Integer Linear Programming in S3 during which complexity theory is tackled up to NP-completeness.

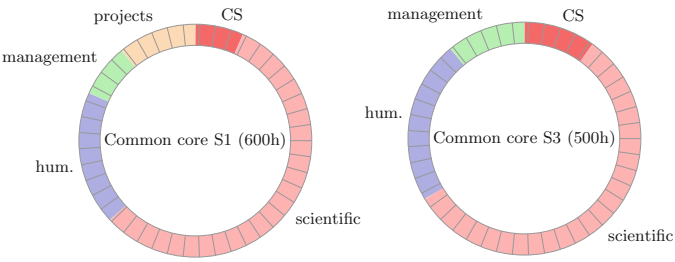

Fig. 1. Computer science part in S1 and S3 semesters

S2 and S4 are mainly dedicated to 30-h elective courses and some of them are Computer Science oriented, e.g. courses on functional and logic programming languages, implementation of control software in C or avionics architecture. However, none of the proposed course provides the students theoretical foundations of Computer Science nor formal methods.

The Computer Science, Telecommunications and Networks major of expertise in S5 have a Computer Science 240-h track mainly oriented towards Critical Systems Architecture. This tracks is composed of 6 courses presented on Table 2. The Model-Based Engineering course is composed of two parts: a 38-h part on SysML and SCADE and a 17-h part on formal methods.

2.2 The Challenges

In order to expose the students of the Critical Systems Architecture major to formal methods, we face several challenges:

– First, we cannot rely on elective courses proposed during S2 and S4 as the program does not enforce specific elective courses for each S5 major. We can only suppose that the students have been exposed to the 90 h of Computer Science courses in the common core, mainly on programming.
– Even if the scientific common core in S1 and S3 is rather large, non Computer Science scientific courses mainly rely on continuous mathematics, which are

Table 2. The courses of the critical systems architecture major

Course	Title	Volume
FITR301	Network and computer architecture	40 h
FITR302	Security	24 h
FITR303	Real-time systems	65 h
FITR304	Model-based engineering	55 h
FITR305	Distributed systems	35 h
FITR306	Conferences	21 h

not necessarily useful for formal methods. Particularly, students lack background on mathematical logic, calculability theory, programming languages semantics etc. On the other hand, students having attended the preparatory courses have a strong background in "classic" mathematics and are comfortable with mathematical proofs.
– There is only a 17-h slot in S5 to expose students to formal methods.

2.3 Why Teaching Formal Methods at ISAE-SUPAERO?

Given these pitfalls, one can wonder why teaching formal methods at ISAE-SUPAERO, particularly with a 17-h slot? We identify two points that justify this decision:

– As the main industrial sector of SUPAERO is aerospace, an introduction to formal methods seems legitimate, particularly for the students attending the Critical Systems Architecture major.
– Formal methods give more visibility to Computer Science as a *science*. As most students are exposed to Computer Science through programming courses, a non negligible part of them sees Computer Science as a technology, not a science. Notice that we do not consider programming as an minor part of a Computer Science curriculum, but it should not be the only one.

3 Introducing Formal Methods in 17 h

Instead of designing a course in which students are exposed to several formal methods, we decided to give students a brief introduction on the subject and then let them choose to learn a particular formal method through a dedicated track. As there are about 16 students in the major, this means that each track will be followed by 4 students. The tracks are the following:

– A track on *model-checking* through LTL and CTL, in which students model a system and prove some temporal properties on it

– A track on *abstract interpretation*, in which students implement an abstract interpreter on a tiny imperative language
– A track on *deductive methods* using SPARK
– A track on *deductive methods* using Frama-C.

The 17 h are divided as follows:

1. A 2-h introduction lecture during which we present what are formal methods, and what are their industrial uses. A small introduction to programming languages semantics is also done through a toy imperative language via denotational, operational and axiomatic semantics.
2. Each track has then six 2-h sessions to work on the corresponding technique. These sessions mix theoretical concepts and practical exercices.
3. Each track evaluates its students through a group project.
4. Each students group has then 30 min to present to the other groups the principles of the method they used, their results, the difficulties they encounter etc.
5. A 2-h industrial feedback is then done on how (aerospace) industry uses formal methods.

In the following, we will focus on the two tracks on deductive methods, particularly because they use two different pedagogical approaches.

3.1 Deductive Verification with Frama-C and SPARK

Deductive verification of programs is a formal method or technique that translates the problem of verifying a program annotated by assertions, invariants and contracts to the satisfiability problem of a particular mathematical logic formulas. Deductive verification relies on early work by Hoare [13], Floyd [12] and Dijkstra [9]. We are using two platforms for deductive verification:

– The Frama-C platform [16] with its WP plugin for deductive verification and the ACSL specification language [6]. Frama-C is dedicated to the analysis of C programs.
– The SPARK language [4] with its associated tool GNATProve. SPARK is a subset of the Ada programming language targeting formal verification.

3.2 A Top-Down Approach with Frama-C

The Frama-C track is built on a classic top-down approach: the students are first exposed to theory and then use the Frama-C tool. The first 2 h are dedicated to proof theory, particularly on formal systems for propositional and first-order logics. Floyd-Hoare logic is then presented and an introduction to weakest precondition calculus is done in 2 h. Students have then to manually annotate small imperative programs (e.g. factorial, euclidian division, greatest common divisor) to understand how weakest precondition works. Let us take for instance a factorial algorithm represented on Listing 1.1. The students have to discover the loop

invariant and variants for the while loop and then use Floyd-Hoare logic rules to annotate each line of the program with assertions about memory. These small exercises are rather useful as they show the students that most of the assertions can be computed automatically, that proof obligations need to reason with particular theories in order to be discharged and that invariants are crucial to prove the expected postconditions but are sometimes difficult to find.

Listing 1.1. A simple imperative program computing factorial

```
{N ≥ 0}
K := 0
F := 1
while (K ≠ N) do
      K := K + 1;
      F := F * K
od
{F = N!}
```

The next 3 sessions are dedicated to a presentation of the Frama-C platform and its WP plugin for deductive verification. The exposure to Frama-C and ACSL is done gradually through small C programs that students have to prove or for which specifications are incomplete or wrong and that students have to debug. For instance, Listing 1.2 presents the previous factorial algorithm written in C with an axiomatic for factorial. Pointers are also tackled, particularly memory separation, which is more difficult.

Listing 1.2. A C program computing factorial

```
/*@ axiomatic factorial {
  @   predicate is_fact(integer n, integer r);
  @   axiom zero:
  @     is_fact(0, 1);
  @   axiom succ:
  @     \forall integer n, integer f;
  @     n>0 ==> is_fact(n-1, f) ==> is_fact(n, f*n);
  @
  @   logic integer fact(integer i);
  @   axiom fact1: \forall integer n;
  @                is_fact(n, fact(n));
  @   axiom fact2: \forall integer n, integer f;
  @                is_fact(n, f) ==> f == fact(n);
  @ }
  @*/

int factorial(int n) {
    int k = 1;
    int f = 1;
```

```
    while (k <= n) {
        f = f * k;
        k = k + 1;
    }

    return f;
}
```

The project the students have to work on is rather classic: it consists in specifying, implementing and proving a small library on strings consisting of three functions:

```
int strlen(const char *str);
void strsubstring(char *dst, const char *src,
                  int start, int length);
void strappend(char *dst, const char *src);
```

3.3 A Bottom-Up Approach with SPARK

The track on deductive methods with SPARK is guided by a different methodology proposed by [5]. This approach is based on different levels of verification applied to SPARK code. The students are gradually exposed to the SPARK language and associated tools:

- On stone level, students have to write valid SPARK code. As a subset of the Ada language, SPARK has some limitations compared to Ada. As students have not been exposed to Ada programming before this course, attaining stone level is just learning a new language syntax and rational for them.
- In order to validate bronze level, SPARK programs must be such that there is no uninitialized variables nor interferences between parameters nor global variables.
- Silver level corresponds to Absence of RunTime Errors (AoRTE). These errors are typically overflows or underflows on integers, or accesses outside the range of an array. In order to validate SPARK programs on the silver level, students have to add preconditions to their functions, e.g. to restrict the possible value of a parameter to avoid possible overflows. They also have to write postconditions when several functions are interacting and are thus initiated to modular proof.
  A typical exercise is to specify a simple stack implementing with an array and offering Initialize , Push and Pop functions. Of course, manipulating an array may lead to illegal accesses, and the students have to add preconditions and postconditions to avoid them.
- Finally, gold level corresponds to functional correctness of SPARK programs. In order to prove that a program is correct, students have to understand how to write more complex contracts for functions and to add invariants and variants if some functions include loops through simple exercises. For instance, Listing 1.3 presents a Find_Int_Sqrt function implementing integer square root algorithm for which the students had to find a loop invariant.

Listing 1.3. A SPARK program computing integer square root

```
function Find_Int_Sqrt (N : in Natural) return Natural
is
   Lower, Upper, Middle: Natural;
   Maximum_Root: constant Natural := 46341;
begin
   Lower := 0;

   if N >= Maximum_Root then
      Upper := Maximum_Root;
   else
      Upper := N + 1;
   end if;

   loop
      --  Add a pragma Loop_Invariant and a
      --  pragma Loop_Variant here.

      exit when Lower + 1 = Upper;
      Middle := (Lower + Upper) / 2;
      if Middle * Middle > N then
         Upper := Middle;
      else
         Lower := Middle;
      end if;
   end loop;
   return Lower;
end Find_Int_Sqrt;
```

The project proposed to the students was to prove a small part of the Ada.Strings.Fixed GNAT standard library. This library consists in functions and procedures manipulating fixed size strings. For instance, the Index function shown on listing 1.4 searches for the first or last occurrence of any of a set characters, or any of the complement of a set of characters.

Listing 1.4. The index function from the GNAT standard library

```
function Index
  (Source : String;
   Set    : Maps.Character_Set;
   Test   : Membership := Inside;
   Going  : Direction  := Forward) return Natural;
```

Students were given 12 functions with their complete implementation and some specification written in natural language. They had to formally specify and prove these 12 functions.

3.4 Comparison of Both Approaches

First, let us notice that both groups of students manage to complete the assigned projects. Even if there were more functions to specify and prove in the SPARK project, the C functions used pointers as parameters, which was difficult to handle for students (particularly the problem of memory separation and the amount of specification needed to prove programs using pointers).

Of course, a top-down approach in such a small amount of time is not efficient. Starting from theory, particularly proof theory, takes a lot of time and there is not enough time to manipulate Frama-C. On the contrary, the bottom-up approach chosen for the SPARK track was more efficient: hands-on session from the beginning of the track is clearly an advantage as students immediately see the benefits of deductive verification on one particular aspect (control flow, AoRTE, functional correctness). Student presentations also shown that students having attended the SPARK track had more hindsight than the ones having attended the Frama-C track.

Finally, notice that the comparison between both approaches may be biased as C is a more difficult language than SPARK, particularly when using pointers.

4 The SPARK by Example Experiment: Learning from Examples

4.1 What Is SPARK by Example?

When wanting to learn to prove C programs using Frama-C and the WP plugin, *ACSL by Example* [7] is a good entry point. *ACSL by Example* is a booklet presenting how to prove with Frama-C more than 50 algorithms extracted from the C++ *Standard Template Library* (STL) [18,19]. Each algorithm in *ACSL by Example* comes with detailed explanations on how to specify it, how to prove it and what possibly are the lemmas needed for the proof. It is therefore a good companion for who wants to learn deductive methods with Frama-C and ACSL.

Even if some interactive learning tools for SPARK have been developed recently [1,2]) and good learning material is already available [3,15], we thought that a "recipe" document in the spirit of *ACSL by Example* was lacking for the SPARK community. Léo Creuse and Joffrey Huget were two second year students willing to learn formal methods during their S4 at ISAE-SUPAERO. We asked them to produce *SPARK by Example*, a SPARK equivalent of *ACSL by Example*, with the following constraints:

- All algorithms proved in *ACSL by Example* should be specified, implemented and proved in SPARK. SPARK Community 2018, freely available, must be used.
- A complete documentation must be produced for each algorithm detailing the specification and the implementation of the algorithm as well as proof "tricks" used for difficult proofs.

- Wrong or incomplete specifications that sometimes seem natural for beginners should also be described in order to explain why they are wrong.
- All proofs must be done in the spirit of [10], i.e. only with automatic SMT solvers and no interactive proof assistants, whereas some proofs in *ACSL by Example* need Coq.

Léo and Joffrey managed to specify, implement, prove and document all algorithms in less than 3 months without previous knowledge of formal methods nor SPARK. *SPARK by Example* is available on [8] and is provided as a Github project with all documentation directly readable from the website. All code source, installation instructions and build artefacts are also provided.

4.2 A Corpus of Proved Algorithms

Algorithms are classified in *ACSL by Example* and therefore in *SPARK by Example* in different chapters.

- 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 $\mathcal{O}(\log n)$. 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 short 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.

SPARK by Example is therefore a rather complete document to learn how to use SPARK to prove classic algorithms. Chapters have an increasing difficulty and we hope that the explanations we have written for difficult notions or proofs are sufficiently clear to help beginners.

4.3 Lessons Learnt

What have we learnt from this experiment? First, it is possible to prove relatively complex algorithms without previous knowledge of formal methods in a relatively small amount of time. Of course, this has to be put in perspective as our students can rely on *ACSL by Example* to understand how to prove most of the algorithms. On the other hand, they had to understand how to build complex proof with lemma functions in SPARK which was clearly not the approach chosen by *ACSL by Example*.

Second, we encounter two difficult points when proving complex algorithms:

- 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 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 an automatic prover.
  In SPARK, there is no proper construction of lemmas as in ACSL. 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.
Therefore, the prover of a program must understand which lemmas are needed for his or her proofs, prove the lemmas and manually add instances of the lemmas in the code in order to help the provers. This is quite equivalent to give a sketch of proof to the provers. Fortunately, certain forms of "templates" appear when implementing lemmas for proving them, so it becomes easier and easier. Moreover, the GNATprove tool is of great help when inserting lemmas in the code to be proved to understand where to place them.

– Auto-active proof of a program is a kind of proof in which the verification conditions (VC) generated from the specification and assertions of the program are discharged only by automatic provers. Most of the time, SMT solvers like Alt-Ergo, CVC4 or Z3 are used as they embed first-order theories that are suitable for program verification (bitvectors or arrays theories for instance). However, SMT solvers are limited. For instance, universal quantifiers in formulas are not handled in a complete way. Therefore, when a SMT solver cannot discharge a VC, there may two several explanations:
  - The VC is effectively false
  - The VC should be discharged but is not due to solver limitation

  The second case may be difficult to understand and solve by beginners, because they have to understand for instance that SMT solvers instantiate formulas involving universal quantifiers using instantiation patterns called triggers. Therefore, when trying to prove a VC with universal quantification, adding a new trigger through may ease the solvers task. We face this problem in a first version of *SPARK by Example* where we had to split a complex assertion on arrays involving nested quantifiers using several auxiliary functions that were used as triggers for the solvers. Notice that SMT solvers are improving quickly and that some VC that cannot be discharged by a solver may be discharged easily by a future version.

5 Conclusion

We presented in this paper two teaching experiments on deductive methods for program verification. The first one takes the form of a classic course on formal methods, but in a very small amount of time, i.e. 17 h. We showed that it is possible for beginners to use tools like Frama-C/WP or GNATProve to verify imperative programs written in C or SPARK. Experience shows also that bottom-up approaches, in which students are using the tools from the beginning of the course and prove more and more complex properties are better than top-down approaches often used in the French "Grandes Écoles" system in which theory is first exposed before practising. The second experiment pretended the production of teaching material, namely *SPARK by Example*, to evaluate students capacity to learn deductive methods. It showed that it is possible for non-experts to use

SPARK to prove relatively complex programs, as an implementation of binary heaps was entirely proved using only SMT solvers. We hope that *SPARK by Example* will also be a useful tool to learn SPARK and deductive methods.

Due to the structure of ISAE-SUPAERO engineering program, students sometimes lack knowledge or background that is useful to specify programs or fully understand how the tools work. For instance, writing complex specifications requires some understanding on how memory is represented in the tools or knowledge of programming language semantics that is currently not taught at ISAE-SUPAERO. Understanding how SMT solvers work and why they may fail to prove a verification condition is also important to handle complex proofs.

Concerning the S5 lecture, industrial feedback given on the last session of the course is really important for the students. They understand that these techniques, although rather mathematical, are used in industry, particularly in aerospace. Qualification and certification are also addressed during the session to show students that a technique or a tool, however attractive or powerful it is, must be incorporated in a global process.

Some ideas arise from these experiences:

- First, we may begin with a more suitable language for deductive verification. In particular, C is a difficult language to use, mainly due to pointers arithmetics, even if ACSL is a really nice specification language to manipulate. WhyML and the Why3 platform [11,20] seem to be natural candidates for a first initiation to deductive verification.
- Create a S4 optional course on reliable software using SPARK. This course would be based on the bottom-up approach described previously and incorporate theoretical sessions when necessary to help students. For instance, a 3-h lecture and practical session on how SMT solvers work would be useful.
- Add more formal methods during the curriculum, particularly in the Critical Systems major. For instance, TLA+ [17] could be introduced in the distributed systems course to show students that formal methods can be used for distributed algorithms.

References

1. AdaCore. Advanced SPARK - online course (2018). https://learn.adacore.com/courses/advanced-spark/index.html
2. AdaCore. Introduction to SPARK - online course (2018). https://learn.adacore.com/courses/intro-to-spark/index.html
3. AdaCore and Altran UK Ltd. SPARK 2014's User Guide (2018). http://docs.adacore.com/spark2014-docs/html/ug/index.html
4. AdaCore and Altran UK Ltd. SPARK 2014's User Guide (2019). http://docs.adacore.com/spark2014-docs/html/ug/index.html
5. AdaCore and Thales. Implementation Guidance for the Adoption of SPARK (2018). https://www.adacore.com/books/implementation-guidance-spark
6. Baudin, P., et al.: ACSL: ANSI/ISO C specification language (2018). https://frama-c.com/download/acsl-implementation-Chlorine-20180501.pdf

7. Burghardt, J., Gerlach, J.: ACSL by Example (2019). https://github.com/fraunhoferfokus/acsl-by-example
8. Creuse, L. et al.: SPARK by Example (2018). https://github.com/tofgarion/spark-by-example
9. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of program. Commun. ACM **18**(8), 453–457 (1975)
10. Dross, C., Moy, Y.: Auto-active proof of red-black trees in SPARK. In: Barrett, C., Davies, M., Kahsai, T. (eds.) NFM 2017. LNCS, vol. 10227, pp. 68–83. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-57288-8_5
11. Filliâtre, J.-C., Paskevich, A.: Why3 — where programs meet provers. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 125–128. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37036-6_8
12. Floyd, R.W.: Assigning meanings to programs. In: Schwartz, J.T. (eds.) Mathematical Aspects of Computer Science. American Mathematical Society, pp. 19–32 (1967) ISBN: 0821867288
13. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969)
14. ISAE-SUPAERO. The ISAE-SUPAERO engineering program (2019). https://www.isae-supaero.fr/en/academics/ingenieur-isae-supaero-msc/engineering-program/
15. McCormick, J.W., Chapin, P.C.: Building High Integrity Applications with SPARK. Cambridge University Press, Cambridge (2015)
16. Kirchner, F., et al.: Frama-C: a software analysis perspective. Formal Asp. Comput. **27**(3), 573–609 (2015). https://doi.org/10.1007/s00165-014-0326-7
17. Lamport, L.: Specifying Systems: The TLA+ Language and Tools forHardware and Software Engineers. Addison-Wesley Professional, Boston (2002)
18. Plauger, P.J., et al.: C++ Standard Template Library. Prentice Hall PTR, New Jersey (2000)
19. International Organization for Standardization (2011), ISO/IEC 14882:2011
20. The Toccata team. Why3. Where programs meet provers (2018). http://why3.lri.fr/