

AdaCore TECH PAPER

# Co-Developing Programs and Their Proof of Correctness

Paper from Communications of the ACM



DOI:10.1145/3624728

## The SPARK programming language and analyzer.

BY RODERICK CHAPMAN, CLAIRE DROSS,  
STUART MATTHEWS, AND YANNICK MOY

# Co-Developing Programs and Their Proof of Correctness

TWENTY YEARS AGO, Sir Tony Hoare proposed a grand challenge to the computing research community: to develop a *verifying compiler [which] uses mathematical and logical reasoning to check the correctness of the programs that it compiles*. Hoare went on to set demanding success criteria for this effort: If the project is successful, a verifying compiler will be available as a standard tool in some widely used programming productivity toolset.<sup>26</sup>

While there have been some notable successes with program verification systems,<sup>a</sup> the use of such systems is still perceived as a niche activity for the most critical and specialized projects<sup>1,23,29,30,34</sup> Program verification systems based on automatic techniques have also emerged.<sup>9,11,14,16,33</sup> These systems occupy a middle ground

a We prefer this term, since we have found that *verifying compiler* is all too easily confused with *verified compiler* which is altogether a different beast.

in the landscape of verification techniques, between push-button tools that require minimal setup and fully interactive proof assistants. The approach has been called “auto-active”—a portmanteau of “automatic” and “interactive”—where users develop the proof and the program at the same time through the use of assertions and other contracts in the programming language.

The open source SPARK technology is a prominent member of that family, with a history dating back to 1987. The objective of this article is to explain the auto-active approach for co-developing programs and their proof of correctness, present the key design and technological choices that made SPARK industrially successful, and what this means for the future of SPARK or analyzers of that same family.

### The Programming Language Matters

**The language subset matters.** The seeds of SPARK can be traced back to the mid-1980s at the University of Southampton in the U.K.<sup>12</sup> Following modest success with verification of a Pascal subset, the team set an ambitious target: the design of a verification system that would be usable for the development of safety-critical software. In retrospect, this seems like an

#### » key insights

- Today, a few companies like NVIDIA are formally proving the correctness of their critical software using the SPARK language and verification technology.
- This advance was made possible by a careful design of the programming language, the specification language, and special “ghost” code intended for specification and verification. The support for reasoning about pointers is dependent on following an ownership policy like the one found in Rust.
- Industrial adoption rests on choosing the most appropriate assurance level for a given project, from prevention of basic defects, through memory-safety and type-safety, and finally to full functional correctness. SPARK supports that practical approach to formal proof.

absurd proposition; systems programming at the time was dominated by C which, with its overt dependence on pointer types, was considered out of the reach of verification tools.

The top-level design goals were as follows:

- **Soundness.** Verification results should be sound—that is, trustworthy for all compilers and target machines.

- **Sufficient completeness.** Verification of non-trivial properties should offer a tolerably low false-alarm rate.

- **Formality.** The language should be amenable to the development of an unambiguous formal definition.

- **Scalability.** Verification should scale to industrial code bases in reasonable time.

- **Modularity.** Verification of incomplete programs should be possible during their development.

- **Expressiveness.** The language should be usable for building embedded, real-time, and critical software systems, not limited to “toy” examples. The verification system should allow the specification of non-trivial correctness properties, not just a list of “common errors.”

The provision of *soundness* for all

compilers and target machines was particularly challenging. This meant the elimination of all undefined behavior (error cases where the language does not define a behavior) and, as a necessary simplification in an industrial tool, removal of all dependence on unspecified language features (cases where the language does not define a unique behavior) through subsetting or analysis.

The team judiciously chose Ada as their base language, which brought key enabling features to the table. In particular, Ada features:

- Modules (aka packages) that come in two parts—a “specification” and a “body”—allowing contracts to be added where required to provide strictly modular analysis.

- Function calls are expressions, while procedure calls are statements. Functions in SPARK are also free of side effects, which greatly simplifies the formal definition and eliminates dependence on expression evaluation order—for example in evaluating arguments of a call. Procedures can take inputs as in parameters, outputs as out parameters, and mutate in out parameters and global variables.

- User-defined scalar types. In Ada, it is normal to declare scalar types that model the problem-domain, not the target computer. These are critical in achieving an acceptable false-alarm rate for type-safety properties, especially freedom from integer or real overflow.

- Composite types (arrays and records) are “first class” in Ada, so they can be passed as parameters and returned from functions without the explicit use of pointers. With a local analysis of aliasing, soundness is preserved regardless of a compiler’s choice of parameter-passing mechanism.

The technology started to find its industrial niche in the early 1990s, when SPARK was selected for the development of all Risk Class 1 systems on the EuroFighter Typhoon program. Given the limitations of the automatic provers and available computing resources at the time, the program initially limited itself to adoption of the language subset and the information flow analysis offered by the SPARK analyzer.

The development of Ada2012 brought a significant change, adding contracts as first-class citizens in the language, effectively rendering SPARK’s special “annotation language” redundant. This brought about a reboot of the language and tools.

The language design was restarted from scratch, adopting Ada2012’s contract language as part of the core. While earlier versions of SPARK concentrated on solely static verification of contracts, Ada2012 allowed for contracts that could be verified both statically or dynamically, or some mix of both styles in the same program. We also took the opportunity to redevelop the entire analyzer, starting with the GNAT Ada compiler front end, and targeting the Why3 intermediate language and toolset,<sup>22</sup> supported by the plethora of SMT-based provers that had come to the fore. Given an annotated program, the SPARK analyzer generates formulas, aka verification conditions (VCs), which are sent to automatic provers. If all VCs are successfully proved, the program correctly implements its (partial) specification. The full compiler front end also allowed the return of several language features, such as generics, dynamic subtypes, and dynamically sized array

**Figure 1. SPARK specification as contracts.**

```

1  type Index is new Integer;
2  type Element is new Integer;
3  type Table is array (Index range <>) of Element
4  with
5      Dynamic_Predicate => Table'First = 1 and Table'Last >= 0;
6
7  procedure Set_Range_To_Zero (T : in out Table; From, To : Index)
8  with
9      Pre => From in T'Range and To in T'Range,
10     Post => T = (T'Old with delta From..To => 0);
11
12 function Is_Range_Zero(T : Table; From, To : Index) return Boolean is
13 (for all I in From..To => T(I) = 0)
14 with
15     Ghost,
16     Pre => (From >= 1) and (To in 0 | T'Range);
17
18 function Is_All_Zero(T : Table) return Boolean is
19 (Is_Range_Zero(T, T'First, T'Last))
20 with
21     Ghost;
22
23 function Search_First_Non_Zero(T : Table) return Index
24 with
25     Post =>
26         (declare
27             NZ : constant Index := Search_First_Non_Zero'Result;
28             begin
29                 (if NZ in T'Range then T(NZ) <= 0 and Is_Range_Zero(T, 1, NZ-1)
30                     else NZ = 0 and Is_All_Zero(T));

```

types, that had been excluded from the earlier designs.

The new version of SPARK, dubbed SPARK 2014 when it was first released in 2014, has undergone significant extensions with every passing year, adding safe support for object-oriented programming, concurrency, contracts on types, and pointers. The language subset is rigorously defined in a reference manual<sup>4</sup> that follows the style of the ISO/IEC definition of Ada. Similarly to how “legality rules” define what it means for a program to be accepted by the Ada compiler, “verification rules” define what it means for a program to be accepted by the SPARK analyzer.

Today, SPARK is a large subset of Ada, only excluding features that would defeat the objectives of reasonably specifying or automatically verifying programs.

**The specification language matters.** SPARK’s contract language inherits from decades of research on behavioral interface specification languages.<sup>24</sup> It started as special stylized comments, with a mathematical semantics. Work done by Chalin in the context of ESC/Java showed that this was a source of confusion for programmers,<sup>13</sup> which led to adopting an executable semantics for contracts as part of their inclusion in Ada2012; dividing by zero in contracts is now an error, like in code.

Another great benefit of executable contracts is that they can be tested and debugged like regular code. This can be very useful during development, as mistakes can be easily identified through testing and investigated through debugging. This can also be used to validate assumptions made during formal verification, by executing contracts during tests (typically integration or validation tests). Of course, an essential property is that a formally verified program should not fail at runtime. Hence, static checks performed by the SPARK analyzer should be a superset of the dynamic checks during execution. They cannot be the same in general, as dynamic checks are designed for the compiler to insert, which has limited knowledge of side effects and aliasing in particular, and the design of dynamic checks needs to find a balance between runtime efficiency and comprehensiveness.

Contracts in SPARK are attached to packages, subprograms, and types in

**Figure 2. SPARK implementation and verification code.**

```

1  -- first version
2  procedure Set_Range_To_Zero (T : in out Table; From, To : Index) is
3  begin
4      for J in From..To loop
5          T(J) := 0;
6          pragma Loop_Invariant (Is_Range_Zero (T, From, J));
7      end loop;
8  end Set_Range_To_Zero;
9
10 function Search_First_Non_Zero (T : Table) return Index is
11 begin
12     for J in T'Range loop
13         if T(J) ≠ 0 then
14             return J;
15         end if;
16         pragma Loop_Invariant (Is_Range_Zero (T, 1, J));
17     end loop;
18     return 0;
19 end Search_First_Non_Zero;
20
21 -- second version
22 procedure Set_Range_To_Zero (T : in out Table; From, To : Index) is
23     T_Entry : constant Table := T with Ghost;
24 begin
25     for J in From..To loop
26         T(J) := 0;
27         pragma Loop_Invariant (Is_Range_Zero (T, From, J));
28         pragma Loop_Invariant (for all K in T'Range =>
29             (if K not in From..To then T(K) = T_Entry(K)));
30     end loop;
31 end Set_Range_To_Zero;

```

the form of a pair of the contract name and value separated by an arrow symbol ( $\Rightarrow$ ). Contracts on types come in two flavors: *predicates*, which can never be violated, and *invariants*, which can be violated locally inside the package defining the type. Consider the specification expressed as contracts in Figure 1. Type `Table` on line 3 is an array of `Element`, whose size is computed dynamically, with a value of `Index` starting at 1 and ending at a positive index, or 0 for the empty array. The allowed range of values for the first and last indexes are specified here through a predicate, as the type system of SPARK otherwise allows arbitrary integer values for both.

Contracts on subprograms consist essentially in a *precondition* and a *postcondition*. The precondition states which combinations of values of input parameters and global variables are allowed when calling the subprogram. The postcondition states which combinations of values of output parameters and global variables are allowed when returning from the subprogram, usually as a relation with input values, where the input value of an expres-

sion  $X$  is denoted  $X'$ Old. Consider the procedure `Set_Range_To_Zero` on line 7. Its precondition states that the parameters `From` and `To` should denote valid indexes in the array `T`. Its postcondition states that the final value of `T` should be the same as its input value, with the values between indexes `From` and `To` zeroed out, using here a delta-aggregate for a compact expression. `Search_First_Non_Zero`, the procedure on line 23, has a default precondition of `True` and a postcondition stating that it returns the index of the first non-zero element if any, using a declare-expression to introduce a local name for the value returned.

**Ghost code: Code for specification and verification.** Functions such as `Is_Range_Zero` on line 12 are only meant for verification. They are not needed in the final executable, which is typically compiled without contracts. In SPARK, such code is specially identified as *ghost code* by attaching the aspect `Ghost` to declarations, so that the compiler can discard it.


In an ideal situation, a programmer would only need to write contracts on an API to get the code of their library

proven. While this may work in some cases, this is not the most common case: A subprogram defined in the API may call other subprograms without specifications, and it may contain loops. Both calls and loops require special handling to be proved; that may require further specification of their behavior. A programmer may have to add contracts to called subprograms and invariants to loops.


Subprogram contracts are essential for modularity, scaling, and automation of proof. The SPARK analyzer can deal with subprograms that bear no contracts by inlining them at the point of call, but this makes VCs more complex, possibly to the point of defeating automatic provers. Thus, such inlining is reserved for private subprograms of a package. For other subprograms, the programmer must write a contract that summarizes the behavior of the subprogram for the analysis of its callers.

The case of loops is similar. When the maximum number of loop iterations is small, the SPARK analyzer can deal with loops by unrolling them, but this also makes VCs more complex and does not work on unbounded loops. In general, the programmer must write a *loop invariant* that states properties known to be maintained in the corresponding location at each iteration of the loop. Consider the implementation for the previous specification in Figure 2. Loop invariants summarize the modifications on variables that occur in previous loop iterations, like in `Set_Range_To_Zero` on line 2, and they accumulate information about values seen in prior iterations, like in `Search_First_Non_Zero` on line 10. Contrary to the original loop invariants from Hoare,<sup>25</sup> a loop invariant in SPARK can be located anywhere in the loop (although it is typically at the start or the end of the loop body), and it only has to hold when execution reaches that point in the code.

The loop invariant in procedure `Set_Range_To_Zero` (first version on line 2) is located at the end of the loop body on line 6. It specifies that table `T` holds value 0 between indexes `From` and `J`. This summarizes the behavior of the loop with enough precision for the postcondition of `Set_Range_To_Zero` on line 10 in Figure 1 to be proved. The invariant



**Systems programming at the time was dominated by C which, with its overt dependence on pointer types, was considered out of the reach of verification tools.**



is also provable inductively: First, the SPARK analyzer proves that the invariant holds during the first iteration of the loop; second, assuming that the invariant holds at an arbitrary iteration, the SPARK analyzer proves that it is preserved during the next.

Proving preservation is typically harder, as the loop invariant expression should not only hold at the corresponding program point, but it should be inductive with respect to the loop body. The elephant in the room is that the VC may contain less information than what the user assumes regarding variables read or written in the loop. A frequently forgotten part of the loop invariant is the so-called *frame condition*, which denotes the parts of a modified variable preserved inside the loop. While the SPARK analyzer has heuristics to generate the frame condition, this is not sufficient in all cases.

It is not sufficient to know that `T` in procedure `Set_Range_To_Zero` has been zeroed out between indexes `From` and the current loop index `J`. To prove the postcondition of the procedure, we should maintain in the loop invariant the information that, for all indexes out of the `From..To` range, `T` maintains the value it had at the entry of the subprogram. The SPARK analyzer generates the frame condition in this case, which is why the first version of `Set_Range_To_Zero` on line 2 is proved. In the second version on line 22, we add the frame condition explicitly in a second loop invariant on lines 28–29. Note the declaration of a ghost variable `T_Entry` on line 23 to hold the value of `T` at entry, so we can compare it with the values in `T` inside the loop invariant. The aspect `Ghost` attached to the declaration of `T_Entry` informs the compiler that this is a ghost variable, which should be deleted unless compiling with assertions.

Ghost variables are more generally useful whenever specification or verification code needs to refer to values which are not readily available in the code like `T_Entry`. Similarly, ghost functions are useful to express queries that are not part of the normal API, such as functions `Is_Range_Zero` and `Is_All_Zero`. Ghost procedures make it possible to group ghost declarations and statements. Ghost code makes it possible to instrument

the code without risking interfering with its behavior, as the SPARK analyzer checks that ghost code cannot have any effect on the functional behavior of the program. For example, a ghost procedure is not allowed to contain an assignment to a regular (non-ghost) variable. Therefore, the guarantees obtained by proving the implementation including ghost code hold also for the execution of the program where ghost code is deleted.

**Dealing with pointers through ownership.** In the absence of pointers, the SPARK analyzer can exclude problematic aliasing through a simple comparison of the “names” (including field names when referring to the subcomponent of a record) of actual parameters and global variables at each call site. This is sufficient for most SPARK programs, as the language provides separate features to pass parameters by reference and to specify the address of data in memory, so that source-level pointers are essential only to deal with heap-allocated memory. Until 2019, SPARK did not support pointers and heap-allocated memory, which was a good match with its use in real-time and embedded software. The lack of pointers, however, disallowed standard design patterns, such as user-defined recursive data structures or access to a part of a structure in place.

Support for pointers was introduced in 2019, based on the *ownership principle* popularized by Rust.<sup>27</sup> While the technical details presented later in this section may challenge the reader’s understanding, the idea is simple: A single object, the owner, is allowed to read or modify data through a pointer. Ownership is transferred through assignment statements and parameter passing. This is sufficient to rule out problematic aliasing. As a result, pointers are handled by the SPARK analyzer like optional values: A pointer can either be null or can contain a value, which is copied when copying the pointer. The resulting VCs are similar to those generated on programs without pointers, and they pose no specific difficulties to automatic provers. Note that the pointer value itself is intentionally not modeled. Otherwise, the SPARK analyzer would need to consider that allocating memory on the heap has a side effect (as a different pointer

value is returned each time), which would prevent doing it in (side-effect free) functions. Deallocation must be done manually, but the analyzer will make sure that no leaks occur.

Even with the restrictions imposed by the ownership policy, it is possible to create recursive data structures in SPARK, as long as they have no cycles or sharing. For example, simple lists or trees are fine, but doubly linked lists or directed acyclic graphs cannot be constructed. Naive iteration over a recursive data structure using a loop is inherently incompatible with ownership, as the handle used for iteration is an alias of the underlying structure. Rewriting the code from Figure 2 to use pointers gives us the variant in Figure

3. Procedure `Set_List_To_Zero` (incorrect version on line 25) implements a naive iteration. The declaration of iterator `X` causes the parameter `L` to lose the ownership of the list. As `X` iteratively drops its handle to each element of the list, there is no way to return ownership to `L` at the end of the procedure. On the contrary, recursive traversals of such structures do not conflict with the ownership policy of SPARK, as can be seen in the ghost function `Is_All_Zero` on line 8. Indeed, as functions have no side effects in SPARK, aliasing between their parameters is not a problem.

Using recursion—if only for ghost functions—makes it necessary to prove termination using a Subpro-

**Figure 3. SPARK program with pointers.**

```

1  type List_Cell;
2  type List is access List_Cell;
3  type List_Cell is record
4    Data : Integer;
5    Next : List;
6  end record;
7
8  function Is_All_Zero(L : access constant List_Cell) return Boolean is
9    (L = null or else (L.Data = 0 and then Is_All_Zero(L.Next)))
10 with
11   Ghost,
12   Subprogram_Variant => (Structural => L); -- structural recursion on L
13
14 function At_End(X : access constant List_Cell)
15   return access constant List_Cell is (X)
16 with
17   Ghost,
18   Annotate => (GNATprove, At_End_Borrow); -- only for prophecy values
19
20 procedure Set_List_To_Zero(L : access List_Cell)
21 with
22   Post => Is_All_Zero(L);
23
24 -- incorrect version
25 procedure Set_List_To_Zero (L : access List_Cell) is
26   X : List := L; -- ownership is transferred to X
27 begin
28   while X ≠ null loop
29     X.Data := 0;
30     X := X.Next; -- the first cell of X is no longer accessible
31   end loop; -- how can the ownership be returned to L?
32 end Set_List_To_Zero;
33
34 -- correct version
35 procedure Set_List_To_Zero(L : access List_Cell) is
36   X : access List_Cell := L; -- ownership is given to X temporarily
37 begin
38   while X ≠ null loop
39     X.Data := 0;
40     X := X.Next; -- X now designates a substructure of L
41     pragma Loop_Invariant
42       (if Is_All_Zero(At_End(X)) then Is_All_Zero(At_End(L)));
43   end loop;
44   -- the ownership returns to L automatically
45 end Set_List_To_Zero;

```

gram\_Variant contract to provide a metric which can be shown to decrease between recursive calls. For structural variants like the one used in `Is_All_Zero` on line 12, the SPARK analyzer attempts to show that recursive calls occur on strict substructures. Since the ownership policy disallows cyclic structures, this is enough to ensure termination.

Returning to iteration, the language has a facility to temporarily transfer ownership of a memory cell. This is possible in SPARK through the concept of *borrow*s. The big idea is as follows. At the declaration of certain local objects, called *borrowers*, the transfer of ownership is considered to be temporary for the duration of the lifetime of the object. The original object is said to be *borrowed* and cannot be used for the duration of the borrow—it can no longer be used to access the designated memory cell. Ownership returns to the borrowed object automatically at the end of the borrow. In SPARK, procedure parameters and local objects of anonymous pointer type are treated as borrowers. As an example, the type of `X` was changed to an anonymous pointer type on line 36, materialized by the use of keyword `access`, in the correct version of procedure `Set_List_To_Zero` in Figure 3.

Note that, whereas the first assignment in the loop on line 39 uses `X` to modify the underlying structure, the second assignment on line 40 only modifies `X` so that it designates a substructure of itself. It is called a *reborrow*. It is also possible to create subprogram parameters and local objects as *observers*, which can only be used to read the underlying structure using an anonymous pointer type introduced with the syntax `access constant T`. In this case, the observed objects remain readable during the lifetime of the observer.

The proof of programs using local borrowers sometimes requires the use of so-called *prophecy values*, which designate the values of the borrowed object and the borrower at the end of the borrow.<sup>36</sup> This is especially useful in invariants of loops, which iterate over recursive data structures. As an example, let us consider the contract of `Set_List_To_Zero` on line 22. For it to be provable, the invariant in

the loop should make it possible to derive the value of `L` from the value of `X` at the end of the borrow. This is possible using the special identity function `At_End` marked with the annotation `At_End_Borrow` on lines 14–18, to designate prophecy values. When referring to `At_End(X)` and `At_End(L)` in the loop invariant on lines 41–42 in `Set_List_To_Zero`, instead of `X` and `L`, we designate the values of `X` and `L` at the end of the borrow. That loop invariant expresses that if the borrower list `X` only contains value zero at the end of its scope, then the borrowed list `L` will also only contain value zero at that point.

This loop invariant is all that is needed to prove the postcondition of `Set_List_To_Zero`. Note that the analyzer does not do any lookaheads in the code to prove the loop invariant itself, but only uses information available at the current program point. Here, the preservation of the loop invariant follows from the fact that the first cell of `X` cannot be modified after the reborrow on line 40. After this point, it can no longer be accessed through `X`, and modifications through `L` during the borrow are illegal. As a result, the SPARK analyzer can prove that its value at the end of the borrow will necessarily be zero.

### Practical Formal Verification Is Not All-or-Nothing

**Levels of software assurance.** With the exception of carefully crafted training examples, conducting a proof of any industrial code of typical size and complexity against its full functional specification can be highly challenging, requiring training, experience, and expert knowledge. Later in this section, we will reflect on why and when it may be appropriate to invest such effort and resources—in the cases where the very highest levels of software integrity and assurance are justified by the costs of potential failure. However, to dismiss formal verification in the general case because it is challenging is to miss the point: Formal verification is not an all-or-nothing endeavor and there are many dimensions to how it can be applied that make it an economically realistic option in a wide range of scenarios.

SPARK has always supported a range of different verification tech-

niques that can be selected by the end user to provide a combination that both suits the application and fits into an overall verification strategy. More recently, this range of options has been encoded into guidance, which defines a scale of software assurance levels<sup>20</sup> pictured in Figure 4. The essence of the analysis involved in each of these levels is described below, along with examples of their successful application.

*Bronze: Correct data and information flow.* Ensuring that a program has correct data and information flow is a prerequisite for each of the levels of assurance that we may wish to target. For source code which conforms to the basic rules of the SPARK language, we can achieve this verification automatically, giving us a guarantee that the code has the following properties:

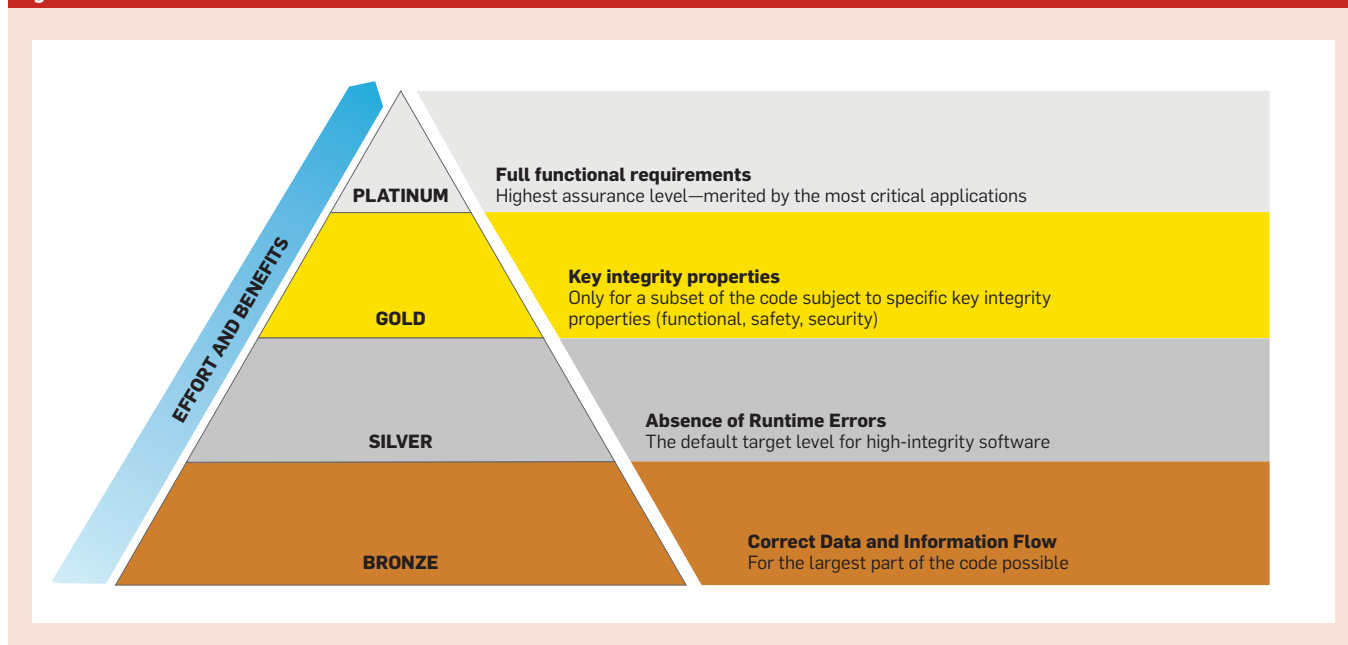
- **Correct initialization:** All variables are initialized before being read for the first time, preventing an undefined behavior that would confound any further verification activities.

- **No aliasing:** It is a principle of the SPARK language that, within any given context, variables should uniquely reference physical memory locations. This includes preventing any potential confusion between global variables and subprogram parameters within a subprogram body.

SPARK also provides an option to use flow analysis to check user-specified data-dependency relations within a program.

*Silver: Absence of runtime errors.* With the foundations of Bronze level in place, we can start to use proof to gain extra assurance about the correctness of our SPARK software. At Silver level, the VCs generated are those required to show that the program will not raise any runtime exceptions. This level of assurance is often referred to as (showing) Absence of Runtime Errors (AoRTE). Runtime exceptions are a native mechanism built into the Ada language to ensure there can be no erroneous behavior within a running program—raising an exception if such a condition arises which can then be safely handled. At a detailed level, there are many exceptions that can potentially be raised in a SPARK program. But broadly speaking, these divide into two main classes that relate

Figure 4. Assurance levels with SPARK.



to type-safety (ensuring that numeric types stay within range; ensuring no divide-by-zero) and memory-safety (ensuring that array indexing stays within bounds; ensuring that pointer dereferences are valid).

In our recent experience of the industrial application of SPARK, we have come to regard Silver as the default minimum we should aim for. This is because it represents a very significant increase in software integrity and reliability for what is typically a relatively small additional investment. Conclusively demonstrating AoRTE by dynamic testing alone is a practical impossibility for all but the most trivial of programs. Yet, this is a common class of errors which can introduce significant hazards into digital systems, making them vulnerable to buffer overflow exploitations, denial-of-service attacks, or loss of safety functions. It also gives us the option to disable runtime exception checking while having the certainty that such errors will not occur.

Among the successful industrial applications at Silver level, perhaps the best example owing to its significance and size is iFACTS,<sup>11</sup> a set of tools used by Air Traffic Controllers at NATS, the U.K.'s leading air traffic services provider. iFACTS provides tools for trajectory prediction, conflict detection, and monitoring aids, and replaced traditional paper information strips with

an electronic display. iFACTS enables air traffic controllers to increase the amount of air traffic they can handle, providing capacity increases and enhancing safety. The iFACTS source code consisted of more than 200kloc of SPARK, from which over 120,000 VCs are generated to prove AoRTE. iFACTS was successfully introduced into NATS' Swanwick Area Centre in November 2011 and is now a core part of the Area Control operation.

*Gold: Proof of functional properties.* Gold level introduces an element of functional correctness into the program, which will be co-developed along with its implementation, using SPARK contracts to specify what a particular abstract data type, procedure, or function should do.

The idea at Gold level is to focus our specification effort on the key properties we want the program to deliver. These properties will depend on the nature of the application. So, for example, a safety-related system of the type found in rail or aerospace will have key safety requirements relating sensor inputs to the state of the outputs. We might specify that an alarm should sound if a particular sensor input is outside a safe range.

Once we have specified the key properties and we have a candidate implementation, then we can prove that the properties hold. The SPARK analyzer will translate our combined

specification and implementation into a set of VCs (added to those it already automatically generated to demonstrate AoRTE), and then attempt to prove each one in turn using the auto-active approach described later in this article.

Although extra effort is required at Gold level, there are two important points to note. First, if errors are undetected at the coding stage, they will leak into later stages of the lifecycle. They might be detected during testing—but there are no guarantees of that—or they might in the worst case remain in the delivered system. It is well understood that the cost of fixing errors increases the later they are found in the development and deployment cycle, so additional effort at the coding stage can be seen as an investment that prevents incurring greater costs at a later date. Second, this incremental approach to assurance means we are focusing our effort on properties and areas where additional effort is justified by the cost of potential failure.

There are a number of industrial examples which have successfully achieved Gold level. In this article, we focus on two more recent examples, SPARKNaCl and NVIDIA, which are presented in the sidebars.

*Platinum: Full functional correctness.* Beyond Gold level, we can continue to add contracts to the code until



every subprogram has a fully functional specification. By this we mean that every subprogram has a postcondition that specifies the value of each of its outputs and a precondition as required to constrain the input space. Further type invariants may also be added over and above those already present from Gold level. Once the implementation has been completed against this full specification and all VCs generated by the analyzer have been proved, we have reached Platinum level of SPARK assurance.

Due to the additional effort involved in developing the specification and proof to this level, Platinum will

## SPARKNaCl

SPARKNaCl is an open source rewrite of the TweetNaCl cryptographic library. The code adopts an auto-active verification style and achieves a completely automated proof of absence of runtime errors and some key correctness properties. Given SPARK's guarantees of correctness, the code can be compiled with aggressive optimization, and all contracts and runtime checking disabled. Verification also led to several proof-driven optimizations in the code. On a 32-bit RISC-V target, the resulting code is more than three times faster than TweetNaCl for a single Ed25519 signature generation.<sup>15</sup>

## NVIDIA Corp.

NVIDIA selected SPARK to face the challenge of delivering safer and more secure products without incurring a large increase in development time and cost. NVIDIA began implementing SPARK in its security strategy in 2019 on select pieces of firmware, then expanded its use by training additional personnel in SPARK and eventually developing an in-house training program. Several NVIDIA teams use SPARK for a wide range of applications, including high-integrity data pipelining, image authentication and integrity checks for the overall GPU firmware image, hypervisors, BootROM, secure monitor firmware, device drivers for automotive safety, and formally verified components of an isolation kernel for an embedded operating system.<sup>5</sup>

only be appropriate for the most critical applications. However, it is worth considering a reduction in unit testing for functional verification if Platinum-level proof has been achieved, since we know that the program will return the correct result for all inputs, not just for those we have been able to test.

Although it is not typical to aim for Platinum level, it has been successfully achieved on a number of industrial applications, most notably:

- ▶ SHOLIS, an instrumentation system that assists with the safe operation of helicopters on naval ships, saw the first successful application of the U.K. Ministry of Defense's DEFSTAN 00-55, a standard for high-integrity software development for military applications. SHOLIS is also an interesting example of how we can partition the source code into different integrity levels and apply different levels of verification within the same program: Platinum for the SIL-4 (highest integrity) subset and Silver for the remainder.<sup>28</sup>

- ▶ Tokeneer, a biometric access-control system, was developed for the NSA to demonstrate that the application of formal methods was practicable for high-assurance, secure systems.<sup>6</sup>

- ▶ GNAT Light Runtime Library is a recent example of the targeted use of full functional proof on a critical subset of the Ada runtime library, subject to certification in the context of various certification standards.<sup>38</sup>

**Pushing the boundary of automation.** The highest levels of software assurance necessitate the automation of proof. This rests ultimately with the automatic provers that are called in the back end of the SPARK analyzer, as illustrated in Figure 5. While the use of Why3 as proof platform allows in principle calling more than a dozen automatic provers, the SPARK analyzer mostly uses three of the Satisfiability Modulo Theories (SMT) family of provers, plus a constraint solver. These open source provers were selected because they collectively provide the best results on the VCs generated by SPARK.

In real life, while a majority of VCs are proved by more than one prover, a small percentage is typically proved by only one prover. On large projects with many thousands of VCs, this small percentage amounts to hundreds or thousands of VCs, which emphasizes

the importance of making provers collaborate.<sup>10</sup>

This collaboration comes naturally from the different strengths and weaknesses of individual provers, so that one prover may prove more easily a given category of VCs. We exploit further this division of work by specializing the generation of VC for each prover. For example, while all three SMT provers support the theory of bitvectors to represent machine integers, we chose to represent all machine integers as mathematical integers (type `int` in SMT-LIB2 format) for the prover Alt-Ergo,<sup>17</sup> while representing some of them as bitvectors for provers cvc5<sup>7</sup> and Z3,<sup>19</sup> to benefit from the specific strengths of each prover.

The SPARK analyzer exploits the collaboration of provers at a very fine-grain level. Every conjunct in every assertion leads to a distinct VC which can be tackled by any prover. Such massive parallelism benefits greatly from modern multicore architectures and/or using cloud computing resources to run the provers.

**The auto-active approach.** As properties or code gets more complex, automatic provers cannot prove every property that is provable in theory. As a result, it is sometimes necessary to supply guidance to the underlying provers, most commonly in the form of intermediate assertions in the code. While trying to understand why some property is not verified by the analyzer, the programmer adds assertions, typically containing intermediate steps in the reasoning toward the property of interest until the intermediate assertions and the final property are proved automatically.

This approach, using a mix of automated verification and user-supplied assertions, is called auto-active verification.<sup>32</sup> The process can involve not only simple assertions but whole pieces of ghost code, as the verification process becomes closer to programming itself. The help provided by the analyzer to identify provability issues<sup>37</sup> and facilitate interaction with the users inside IDEs is paramount here, as work at the source level has replaced inspection of formulas in earlier generations of SPARK and similar technologies. That includes in particular counterexamples generated by provers, that is,

values of inputs that lead to a failure.<sup>18</sup>

We have identified a number of “proof patterns” for Ghost code:

*Exhibiting a witness.* Proving existentially quantified formulas is hard for provers, as it requires guessing an appropriate value. To help the analyzer, it is possible to exhibit a witness—a value which has the expected property. For example, to prove the existential property (for some  $X$  in  $A..B \Rightarrow \text{Prop}(X)$ ), one can prove first  $\text{Prop}(\text{Witness})$  for a value  $\text{Witness}$  in the range  $A..B$ .

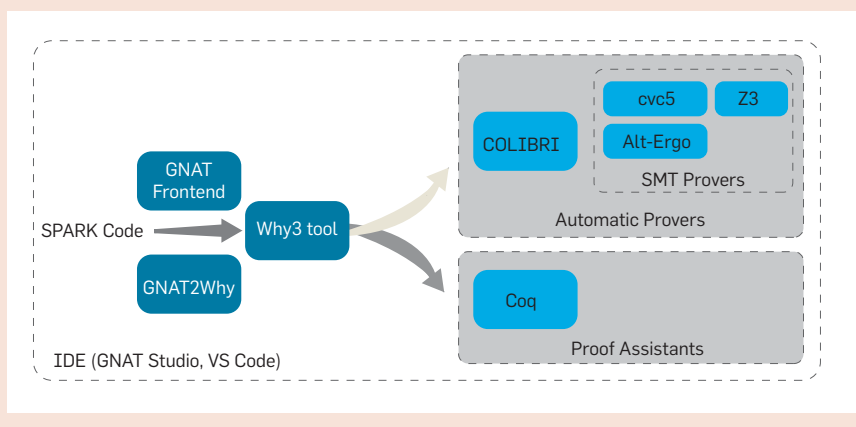
*Reasoning by induction.* In general, inductive reasoning is out of reach of automated provers. It is possible to help the analyzer using a loop with a loop invariant or using recursive calls. For example, to prove that any two elements in  $T$  are in the same order as their respective indexes from the knowledge that consecutive elements are sorted, one can write a loop increasing the maximum distance  $\text{Len}$  between such indexes, consisting only of a loop invariant stating that elements whose indexes are no more than  $\text{Len}$  apart are sorted.

*Factoring out common reasoning.* A lemma is a ghost procedure with a contract but no other effect. The precondition contains the premises of the lemma and the postcondition is the conclusion. When the procedure is analyzed, the analyzer verifies the lemma itself, namely, that the conclusion follows from the premises in every context. The body might contain ghost code to help the proof. As SPARK analysis is modular on a per subprogram basis, when the procedure is called, the premises are verified and the conclusion is assumed.

*Simplifying the proof context.* Lemmas have the advantage of reducing the proof context, as only the relevant hypotheses are kept as a precondition. It makes it easier for the automatic provers to find a proof. Some lemmas can be proved with an empty body even though the same assertion was not verified in the subprogram context. It is the case in particular for non-linear arithmetic lemmas which are theoretically provable but hard for provers. SPARK comes with a standard library of such lemmas.

**Full functional correctness is challenging.** Researchers Rustan Leino and Michał Moskal once said, “Pro-

Figure 5. Structure of the open source SPARK analyzer.



gram verification is unusable. But perhaps not useless.”<sup>32</sup> Indeed, proving full functional correctness of programs will remain hard for the foreseeable future. As we saw, this is only one of the options when considering assurance levels. But there are cases where we would like to achieve that Platinum level of verification. The auto-active approach makes it possible, but some difficulties remain.

To both simplify the annotation process and make the verification easier, some mathematical concepts are provided as libraries along with the SPARK analyzer. In particular, Ada2022 defines libraries for big (unbounded) integers and big rational numbers. The SPARK analyzer offers built-in support for them so they can be used in an efficient way. The big integers of Ada are handled by the analyzer as mathematical integers, which avoids the need for overflow checks in contracts and allows verifying algorithms that deal with very large numbers, for example in crypto libraries. Big rationals can be used to reason about the rounding error in floating point algorithms, for example.

SPARK also provides a functional containers library with unbounded sequences, sets, and maps. As opposed to containers generally used in programming languages, the main concern in their design is not efficiency, but the simplicity of the model used for the verification, as well as a proximity with the structures generally used in higher-level reasoning. Similarly to big numbers, these containers are meant to be used primarily in contracts, where efficiency might not be an issue.

Using this library to model the content of an array as a multiset, it is possible to prove the full functional correctness of a sorting algorithm, with as much ghost code as normal code.<sup>2</sup> The specification and verification of complex data structures or numerical algorithms requires notably more ghost code than normal code. For the specification of a library of red-black trees for bare-metal target (no dynamic allocation), the size of contracts (including data structure invariants) was twice the size of the code, and the size of ghost code was five times the size of the code.<sup>21</sup> For the specification of a library of multi-precision integer arithmetic as part of the GNAT compiler runtime library, the size of contracts was only a tenth of the size of the code, but the size of ghost code was again five times the size of the code.<sup>38</sup>

With as much ghost code to guide automatic provers, many VCs are proved by only one prover. Hence, changes in the code or in the analyzer may lead to some VCs not being proved anymore by any prover. This fragility of automatic proofs of functional correctness has a cost for users, who need to occasionally *repair the proof* by engaging again in auto-active proof. Advances in automatic proof technology will be needed to improve this situation.

**What you get is only as strong as your assumptions.** John Rushby, in his pioneering work on the use of proof for critical software,<sup>39</sup> pointed at two caveats of proof compared to tests: the need for *internal consistency* to ensure that specifications are not inconsistent, as this would allow proving anything, and the need for *external fidelity* to ensure

that specifications are correctly formalizing the environment in which the software operates, otherwise voiding the guarantees provided by proof. We are collectively referring to these caveats as proof *assumptions* that should be listed and reviewed.

Some assumptions are explicitly acknowledged by users when they justify that some checks cannot be violated in their operational context. Other assumptions relate to the boundary of the SPARK program, which may be linked with libraries in other programming languages, or rely on the behavior of an operating system or a specific hardware. All these assumptions should be carefully reviewed by users to ensure the verification results provided by the analyzer are correct for the target context.<sup>3</sup>

### SPARK in the Future

SPARK continues to evolve to support programming language features used in critical software and to reap benefits from improvements in the underlying automatic prover technologies. SPARK is also used as the target programming language for code generators from higher-level representations in domain-specific languages. The challenges in proving generated code are different from those with manually written code, as generated code is more regular but also much larger in general.


Since the publication of the results of the Spec# experience in 2011 in *Communications*,<sup>8</sup> some parts of the software industry have adopted program proofs as one of the paradigms of programming. Further adoption rests essentially with education of the active and future workforce, which will be helped by the publication of more resources on the auto-active approach.<sup>31</sup>

But as we push the boundary of what can be automated, we are faced with the well-known Left-Over Principle of automation: Automation fails humans precisely on the more complex cases, where humans would need more help.<sup>35</sup> A large part of our effort is therefore dedicated to designing better interaction mechanisms to gracefully transition from automation to interaction where needed.

Finally, with more software teams reconsidering their choice of programming languages for producing safe

and secure software, the future looks bright for SPARK!

### Acknowledgments

SPARK is the result of decades of work from past and current members of the SPARK development team, whom we represent here. The current version of SPARK owes much to the collaboration with research team Toccata of Inria on Why3. The anonymous reviewers made many suggestions to make this article more accessible. We are indebted to all of you. 

### References

- Abrial, J. Formal methods: Theory becoming practice. *J. Univers. Comput. Sci.* 13 (2007), 619–628.
- AdaCore. *A Concrete Example: A Sort Algorithm* (2023); <https://bit.ly/41KawzQ>
- AdaCore. *Managing Assumptions* (2023); <https://bit.ly/41RDX35>
- AdaCore. *SPARK Reference Manual* (2023); <https://bit.ly/3TIsDEx>
- AdaCore and NVIDIA. *Case Study* (2022); <https://bit.ly/4aKXvKo>
- AdaCore. *Tokeneer* (2008); <https://www.adacore.com/tokeneer>
- Barbosa, H. et al. cvc5: A versatile and industrial-strength SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*. D. Fisman and G. Rosu (Eds.). Springer Intern. Publishing (2022), 415–442.
- Barnett, M. et al. Specification and verification: The spec# experience. *Commun. ACM* 54, 6 (Jun. 2011), 81–91; 10.1145/1953122.1953145
- Baudin, P. et al. The dogged pursuit of bug-free C programs: The frama-C software analysis platform. *Commun. ACM* 64, 8 (Jul. 2021), 56–68; 10.1145/3470569
- Bobot, F. et al. Why3: Shepherd your herd of provers. In *Proceedings of Boogie 2011: First Intern. Workshop on Intermediate Verification Languages* (2011), 53–64; <https://hal.inria.fr/hal-00790310>
- Bobot, F., Filliâtre, J., Marché, C., and Paskevich, A. Let's verify this with Why3. *Intern. J. on Software Tools for Technology Transfer* 17, 6 (2015), 709–727; 10.1007/s10009-014-0314-5
- Carré, B. and Garnsworthy, J. SPARK—An annotated Ada subset for safety-critical programming. In *Proceedings of the Conf. on TRIADA '90*, Association for Computing Machinery (1990), 392–402; 10.1145/255471.255563
- Chalin, P. A sound assertion semantics for the dependable systems evolution verifying compiler. In *Proceedings of the 29th Intern. Conf. on Software Engineering* (2007), 23–33; 10.1109/ICSE.2007.9
- Chalin, P., Kiniry, J.R., Leavens, G.T., and Poll, E. Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In *Formal Methods for Components and Objects*. F.S. de Boer, M.M. Bonsangue, S. Graf, and W. de Roever (Eds.). Springer Berlin Heidelberg (2006), 342–363.
- Chapman, R. *SPARKNaCl GitHub project* (2020); <https://bit.ly/48CPsgR>
- Chapman, R. and Schanda, F. Are we there yet? 20 years of industrial theorem proving with SPARK. In *Interactive Theorem Proving*. G. Klein and R. Gamboa (Eds.). Springer Intern. Publishing (2014), 17–26.
- Conchon, S., Coquereau, A., Iguernlala, M., and Mabsout, A. Alt-Ergo 2.2. In *Proceedings of the Intern. Workshop on Satisfiability Modulo Theories* (2018); <https://hal.inria.fr/hal-01960203>
- Dailler, S., Hauzar, D., Marché, C., and Moy, Y. Instrumenting a weakest precondition calculus for counterexample generation. *J. of Logical and Algebraic Methods in Programming* 99 (2018), 97–113; 10.1016/j.jlamp.2018.05.003
- de Moura, L. and Björner, N. Z3: An efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th Intern. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*. Springer-Verlag (2008), 337–340.
- Dross, C. et al. Climbing the software assurance ladder—Practical formal verification for reliable software. *Electron. Commun. Eur. Assoc. Softw. Sci. Technol.* 76 (2018).
- Dross, C. and Moy, Y. Auto-active proof of red-black trees in SPARK. In *NASA Formal Methods*. C. Barrett, M. Davies and T. Kahsal (Eds.). Springer Intern. Publishing (2017), 68–83.
- Filliâtre, J. and Paskevich, A. Why3—Where programs meet provers. In *Programming Languages and Systems*. M. Felleisen and P. Gardner (Eds.). Springer Berlin Heidelberg (2013), 125–128.
- Fonseca, P., Zhang, K., Wang, X., and Krishnamurthy, A. An empirical study on the correctness of formally verified distributed systems. In *Proceedings of the 12th European Conf. on Computer Systems*. Association for Computing Machinery (2017), 328–343; 10.1145/3064176.3064183
- Hatcliff, J. et al. Behavioral interface specification languages. *ACM Comput. Surv.* 44, 3, Article 16 (Jun. 2012), 58; 10.1145/2187671.2187678
- Hoare, C.A.R. An axiomatic basis for computer programming. *Commun. ACM* 12, 10 (Oct. 1969), 576–580; 10.1145/363235.363259
- Hoare, T. The verifying compiler: A grand challenge for computing research. *J. ACM* 50, 1 (2003), 63–69; 10.1145/602382.602403
- Jung, R., Jourdan, J., Krebbers, R., and Dreyer, D. Safe systems programming in rust. *Commun. ACM* 64, 4 (Mar. 2021), 144–152; 10.1145/3418295
- King, S., Hammond, J., Chapman, R., and Pryor, A. Is proof more cost-effective than testing? *IEEE Trans. on Software Eng.* 26, 8 (2000), 675–686; 10.1109/32.879807
- Klein, G. et al. Formally verified software in the real world. *Commun. ACM* 61, 10 (Sep. 2018), 68–77; 10.1145/3230627
- Klein, G. et al. Ten challenges for making automation a "team player" in joint human-agent activity. *IEEE Intelligent Systems* 19, 6 (2004), 91–95; 10.1109/MIS.2004.74.
- Leino, K.R.M. *Program Proofs*. The MIT Press (2023).
- Leino, K.R.M. and Moskal, M. Usable auto-active verification. In *Proceedings of the Usable Verification Workshop* (2010).
- Leino, K.R.M. Accessible software verification with Dafny. *IEEE Software* 34, 6 (2017), 94–97; 10.1109/MS.2017.4121212
- Leroy, X. A formally verified compiler back-end. *J. Autom. Reason.* 43, 4 (Dec. 2009), 363–446; 10.1007/s10817-009-9155-4
- Limonicelli, T.A. Automation should be like Iron Man, not Ultron. *ACM Queue* (2015); <https://bit.ly/3voiz9D>
- Matsushita, Y., Denis, X., Jourdan, J., and Dreyer, D. RustHornBelt: A semantic foundation for functional verification of rust programs with unsafe code. In *Proceedings of the 43rd ACM SIGPLAN Intern. Conf. on Programming Language Design and Implementation*. Association for Computing Machinery (2022), 841–856; 10.1145/3519939.3523704
- Moy, Y. How the analyzer can help the user help the analyzer. *Electronic Proceedings in Theoretical Computer Science* 338 (Aug. 2021), 97–104; 10.4204/epics.338.12
- Moy, Y. *Proving the Correctness of GNAT Light Runtime Library* (2022); <https://bit.ly/3NQCvbk>
- Rushby, J. Formal methods and the certification of critical systems. *Technical Report* (1993).
- Zhao, Y., Sanán, D., Zhang, F., and Liu, Y. High-assurance separation kernels: A survey on formal methods. *CoRR abs/1701.01535* (2017); <http://arxiv.org/abs/1701.01535>


**Roderick Chapman\*** (rodchap@amazon.co.uk) is a senior principle applied scientist at Amazon Development Centre, London, U.K.

**Claire Dross** is SPARK team lead at AdaCore, Île-de-France, Paris, France.

**Stuart Matthews** is a senior architect at Capgemini Engineering, Bath, U.K.

**Yannick Moy** is head of the Static Analysis Unit at AdaCore, Île-de-France, Paris, France.

\*The work reported in this article was completed before Chapman joined AWS.

 This work is licensed under a <http://creativecommons.org/licenses/by/4.0/>

Unlock the power of secure, reliable  
software with SPARK Pro

# AdaCore

[adacore.com](https://adacore.com)

