A Comparison of SPARK with MISRA C and Frama-C

Johannes Kanig, AdaCore

November 2021

Abstract

Both SPARK and MISRA C are programming languages intended for high-assurance applications, i.e., systems where reliability is critical and safety and/or security requirements must be met. This document summarizes the two languages, compares them with respect to how they help satisfy high-assurance requirements, and compares the SPARK technology to several static analysis tools available for MISRA C with a focus on Frama-C.

1 Introduction

1.1 SPARK Overview

Ada [1] is a general-purpose programming language that has been specifically designed for safe and secure programming. Information on how Ada satisfies the requirements for high-assurance software, including the avoidance of vulnerabilities that are found in other languages, may be found in [2, 3, 4].

SPARK [5, 6] is an Ada subset that is amenable to formal analysis and thus can bring increased confidence to software requiring the highest levels of assurance. SPARK excludes or restricts features that are difficult to analyze (such as pointers and exception handling). Its restrictions guarantee the absence of unspecified behavior such as reading the value of an uninitialized variable, or depending on the evaluation order of expressions with side effects. But SPARK does include major Ada features such as generic templates and object-oriented programming, a simple but expressive set of concurrency (tasking) features known as the Ravenscar profile, and pointers that must adhere to an ownership policy. SPARK has been used in a variety of high-assurance applications, including hypervisor kernels, air traffic management, and aircraft avionics.

In fact, SPARK is more than just a subset of Ada. It also offers new features for expressing various desired program properties, or “contracts”. Some contracts reflect data or information flow/dependencies, while others express functional requirements. When the program is compiled with a standard Ada compiler, these contracts are either ignored or produce run-time checks, but they can be checked statically by the SPARK analysis tools. Some forms of contracts are explained in this document.

The SPARK tools enforce the SPARK restrictions and attempt to verify (in fact, prove) the contracts. The analysis will statically detect and prevent various classes of errors, such as buffer overflow.

A SPARK program is amenable to formal analysis using modern proof technology, bringing mathematical assurance to the software verification process. But SPARK also allows traditional testing-based verification. SPARK is the only industrially-supported formal methods technology that can combine proof and testing.

There have been various iterations of the Ada language, and SPARK also has changed over the years. Until the SPARK 2005 revision, SPARK contracts (also known as annotations) were written as specially formatted Ada comments. As of SPARK 2014, these annotations are expressed not as comments but through standard Ada 2012 constructs with semantic significance.

1.2 MISRA C Overview

The C programming language [7] has been the most widely used programming language in embedded systems. However, C is well known for having a number of features with syntactic or semantic “traps and pitfalls” that make it easy to write incorrect code. The MISRA C standard [8] attempts to address this issue and promote “best practices” for using C in safety-related applications. MISRA C specifies well over 100 rules including:

- Assignment ("=") should not be used in expressions. This avoids the common error of using an assignment, instead of the comparison operator ("=="), in a condition.
- Braces {} should always be used in connection with if-statements and loops. This avoids the common error where a statement appears to be part of a branch or loop, but in fact it is not.
- The use of dynamic memory allocation is forbidden.
- A type system, called "essential types", rules out the most surprising consequences of C’s implicit conversions.

A variety of commercially available tools check compliance with some of the MISRA C rules. However, a number of MISRA C rules cannot be fully checked by any tool.

2 SPARK compared to MISRA C

Our comparison will focus on four criteria that are important for high-assurance software:

- Syntax - is the textual representation of the program easy to read and does it prevent surprises where a construct seems to mean one thing but has a different effect?
- Typing - can the programmer specify precise information about the data it manipulates and can misuses be detected automatically (either at compile time or run time)?
- Subsetting - Can the subset restrictions be enforced, what benefits do they bring, and how general is the subset?
- Run-time verification - Does the language provide means to verify program properties at run time?

A note on terminology:

Different languages often use different terms for the same basic concept, or use the same term to mean different things. The following conventions are used in this document:

- The term “code module” is used when referring generically to a parameterized routine that can be invoked from multiple parts of the program. Code modules correspond to functions in C (and thus MISRA C) and subprograms in Ada (and thus SPARK). A subprogram can either be a *function* (if a value is to be returned) or a *procedure* (if it is to be executed for its side effect and does not return a value). Thus a SPARK procedure is like a MISRA C function that returns void.
- The description of a specific language will use that language terminology.

2.1 Syntax

The original Ada design exploited the major software engineering breakthroughs from the early to mid 1970s (most notably encapsulation and information hiding). It focused on source code readability and maintainability and SPARK has inherited these advantages. In contrast, C has many well-known pitfalls such as (missing) break statements in the branches of a switch statement, the confusion between assignment and equality (especially in boolean conditions), and the optional braces associated with if statements. Although MISRA C has removed many of these issues, SPARK still has the advantage when it comes to avoiding traps and pitfalls.

- *Preprocessor*
  Ada (and thus SPARK) has no need for a preprocessor. In C, it is sometimes difficult to see what is a macro and what is C code and the interactions between C code and macros can be confusing.
- *Compound statements*
  The syntax for Ada (and thus SPARK) control flow statements such as if-statements, loops, and case statements (similar to switch statements in C), has a consistent format and prevents “dangling else” problems. Each compound statement needs to be terminated by a matching “end” (for example “end if”, “end case”, or “end loop”) so it’s easy to see where each construct begins and ends. The compiler checks for redundant or missing branches in a case statement and loop indices cannot be assigned to.
- *Parameter passing notation*
  In Ada (and thus SPARK), actual parameters to subprograms can be passed by naming the formal parameter they correspond to, which makes it easier to understand a subprogram call and the purpose of each parameter.
- *Case sensitivity in identifiers*
  A potential problem in any programming language is the confusion between similarly named variables. Identifiers in C are case-sensitive, so the variables with names "velocity", "Velocity" and "VELOCITY" are all different and may appear in the same scope. This can hinder readability, since the use of case conventions to convey semantic intent is rather subtle. In Ada (and thus SPARK), all identifiers are case-insensitive and all variations of case designate the same identifier. This may also be confusing (why did the programmer use different case conventions for the

same identifier in different places?), so Ada compilers usually warn when different casings of the same identifier are used in the program.

- *Namespace management*
  C doesn't offer any advanced namespace management; all function names live in the same namespace and are not allowed to clash. Languages like C++, Java, and Ada offer some way to structure the namespace, like packages and child units in Ada, and namespaces and classes in C++ and Java.

2.2 Type System

The type system of a language has several main purposes:

- It allows the programmer to model the logical structure of the objects from the application domain, including any constraints on the values.
- It supplies the operations for manipulating the objects and, in particular (if the language is “strongly typed”), prevents an object of one type from being treated as though it were of another type.
- It tells the compiler the memory layout of the program’s data.

The type systems in SPARK and MISRA C differ with respect to their generality and their reliability in meeting these objectives.

2.2.1 Integer Subranges

Even though MISRA-C addresses the most obvious deficiencies of the C type system, Ada (and thus SPARK) provide the more general and reliable capabilities. For example, in Ada the user can define integer types with arbitrary ranges, like this:

```
type Rating is range 0 .. 10;
```

A variable of type Rating can only be assigned values in the specified range; an attempt to assign a value outside this range will produce a run-time error (raising an exception). With SPARK tools, such attempts can be detected statically; if none is detected then the programmer is assured that no such errors will occur.

In C, only the standard integer types can be used; there is no built-in support for subranges.

2.2.2 Arrays

In C, arrays are basically pointers and no information about their bounds is automatically available. Passing an array as an argument to a function or returning an array from a function require special care and usually involve passing the length and offset of the array as extra arguments. In contrast, with Ada and SPARK, arrays are a separate type category, independent of pointers, and the bounds of an array are always known at the point of use of the array.

A common programming error is an attempt to access an array outside of its bounds. An example is the well-known "buffer overflow" that can compromise safety and security. MISRA C has several rules which disallow such accesses, but the rules are marked "undecidable", meaning that most MISRA C checkers cannot find all instances of such problems. SPARK provides two ways to completely eliminate such errors:

- In standard Ada semantics, there will be run-time checks for array accesses: whenever an array element is accessed, a check is made that the index is within the bounds of the array. If not, a run-time exception is raised.
- The SPARK proof tools can be used to prove that all array accesses are within the array's bounds, so no such error can occur.

2.2.3 Parameter Passing

In C, there is no way to specify that a parameter of a function should only be read or that it's allowed to be read and written; pointers must be used for this purpose. However, pointers are a very powerful and dangerous language feature. In Ada and SPARK, the "mode" of each parameter can be specified, indicating the direction of data flow; for example::

```
  procedure P (A : in Integer; B : in out Integer; C : out
Integer);
```

This specifies that procedure P only reads A, is allowed to read and write B, and writes C and the compiler verifies this. The difference between "in out" and "out" is that an "in out" parameter is expected to be read before being written, while an "out" parameter is expected to be written before it is read. In SPARK, analogous modes can also be specified (and checked by the SPARK tools) for accesses to global variables. No use of pointers is necessary to modify parameters. The compiler takes care of passing values by reference for efficiency when needed and ensures that this does not introduce unsafe aliasing.

2.2.4 Unions / Variant Records

MISRA C disallows the use of C unions, because they are inherently unsafe to use. Ada provides a language feature called "discriminated records", which is somewhat similar to C unions; it can be used to describe "variant" records where certain fields are present only in some cases, depending on the value of a "discriminant" field. Discriminated records in SPARK are entirely type safe and do not defeat the language's type checking even though fields with different types may be overlaid.

2.2.5 Pointers

As previously noted, SPARK supports pointers (or "access types" as they are known in Ada) only if they adhere to the so-called ownership policy. This policy makes it impossible for pointers to introduce problematic aliasing. As a consequence, certain code patterns such as doubly-linked lists or trees with shared subtrees cannot be implemented in SPARK. This may seem to limit the programmer's ability to define such data structures, but in practice the restriction has not proved to be a problem. If such data structures become necessary, it is possible to mix SPARK code with full Ada. The Ada data structures can use pointers without any restrictions (though their functional correctness would need to be verified separately, through testing and perhaps other analysis techniques).

2.2.6 Data Representation

As in MISRA C, one can indicate the bit-precise memory layout of records in Ada and SPARK. However, this can also be done in a way which is completely portable even across big-endian and little-endian systems.

2.3 SPARK and MISRA C as Subsets

Both SPARK and MISRA C are subsets of the larger languages: Ada and C. This raises several questions:

- Is it easy (or even possible) to determine whether a program meets the subset restrictions?
- How well does the subset manage to eliminate dangerous features of the larger language?
- How restrictive is the subset?

The SPARK tools check if an Ada program is written in the SPARK subset. This analysis is sound, so if the SPARK tools say that the program is SPARK, then there are no violations of the SPARK rules.

For MISRA C, the situation is quite different. Many MISRA checkers only check a subset of the MISRA C rules. In fact, twenty-seven rules of the MISRA standard are marked "undecidable", which means that an automatic tool cannot completely verify whether such rules are obeyed. It will either miss violations or report situations which are, upon closer inspection, not violations of the rule (false alarms). So in fact it is difficult to know if a C program really is a MISRA-C program [9].

The MISRA C standard does indeed eliminate many of C's dangerous features. However, MISRA C still includes unrestricted pointers, which are a well known source of errors and which complicate analysis because of the aliasing that they introduce. Pointers are needed in MISRA C since they are intrinsic to C semantics (for example with parameter passing). The use of goto is only discouraged. Side effects in expressions are excluded, but the rule is marked "undecidable".

Because of Ada's array type mechanism and parameter modes, many uses of pointers in C do not require pointers in SPARK; thus pointers are restricted in SPARK. Likewise goto statements are restricted to forward gotos, since other forms of goto complicate data flow analysis. Side effects in expressions are excluded, so order of evaluation is irrelevant. Thus an implementation dependence in full Ada (order of evaluation) doesn't matter; any order chosen will have the same result.

SPARK includes a number of language features that are useful in safety-critical systems: object-oriented programming, generics (similar to C++ templates), and concurrency. These are absent from MISRA C.

Finally, with its various forms of contracts (such as subprogram pre- and postconditions) SPARK offers extensive support for program verification. MISRA C lacks such support, although several tools provide some verification features. This is discussed below.

Note that an application can contain some parts that are in SPARK and others that are outside the SPARK subset (full Ada, or modules from other languages, such as C).

2.4 Run-time Verification

The SPARK support tools can statically analyze a program and prove properties ranging from absence of run-time errors to compliance with formally specified requirements, but numerous properties of a SPARK program can also be checked during testing. For example, standard Ada semantics means that array indexing, division by zero, integer overflow, and assignments outside of the declared range of a type are all checked during program execution.

One mode of operation for SPARK is to enable checks during testing to detect such error situations as early as possible and to suppress them for the production executable for more efficiency. An alternative is to selectively keep checks in the production code, for example to protect against buffer violations. In a security context, it may be safer to stop the program than to allow a security breach.

In Ada, and thus SPARK, contracts may be supplied for subprograms. Such contracts take the form of preconditions and postconditions, which are basically assertions that are checked at subprogram call and return. However, they are part of the subprogram specification and also are more flexible than normal assertions (e.g. a postcondition can refer to the value of an “in out” parameter at the point of call as well as the point of return). Contracts document the intended behavior of the program and, in effect, embed low-level requirements in the source code. They can be executed during testing, left in the production binary if desired, and are subjected to correctness proof by the SPARK tools.

MISRA C as a language has no such features, aside from assertions.

3 SPARK and Static Analysis Tools for C

In this section, the term “SPARK” will be used in a broad sense to denote the complete SPARK technology: the SPARK programming language and its supporting toolset that applies formal methods to SPARK programs.

3.1 General Classes of Static Analysis

A distinguishing characteristic of a static analysis tool is whether or not it is *sound [16]*. A tool is sound with respect to a given source code property if it will find all instances of that property (for example, an attempt to reference an uninitialized variable), and unsound if it does not. For a sound tool, another characteristic is its generality, i.e. the set of properties that it can detect or demonstrate; two examples are absence of runtime errors and full functional correctness.

Unsound tools can be useful for finding defects in code. Examples of such tools, also called “bug finders”, are Coverity and CodeSonar. These tools usually employ static analysis techniques to find bugs or suspicious code, with a relatively low percentage of false alarms (also known as false positives), but they do not guarantee finding all bugs. Such tools cannot be used to obtain any guarantees concerning the analyzed code, because nothing can be said if the tool returns without reporting an error.

The second category, sound tools that can prove absence of run-time errors, includes Polyspace and CodePeer (and also SPARK, but that is discussed below). Sound analysis tools find all potential problems of a certain kind. However, they also report false alarms, so the user needs to review the reported problems and find (and fix) the ones that represent real issues.

The third category, sound tools that prove functional properties, includes Frama-C and again SPARK. The user has to identify, via annotations or contracts, what the program is supposed to do (its specification). Besides demonstrating an absence of run-time errors, these tools can also check the annotations/contracts to verify whether the code corresponds to its specification.

3.2 The Frama-C framework

Frama-C is not a monolithic tool, but rather a platform for static analysis, that contains various plug-ins to achieve different functionality. In this document, we will mention the Eva plug-in, which corresponds to the second category of sound tools, and the WP plug-in, which can be used to prove functional properties.

The company TrustInSoft [17] commercializes a modified version of Frama-C under the name TIS Analyzer, which currently consists in an analysis based on abstract interpretation with the interval domain, similar to what the *Value* plug-in (predecessor of the *Eva* plug-in) provided in Frama-C. TrustInSoft has recently started an effort to develop the *J3* plug-in, with similar functionalities as the *Jessie* and *WP* plug-ins of Frama-C.

3.3 Static Analysis in SPARK

SPARK performs sound static analysis, so it can detect and report certain classes of errors completely, allowing the programmer to repair the problem and eliminate the possibility of such errors at runtime. These include:

- Logical errors such as dead stores and unused variables
- Access to uninitialized data
- Division by zero
- Arithmetic overflow
- Assigning an out-of-range value to an integer or floating-point variable
- "Buffer overflow", such as accessing an array outside of its bounds
- Misuse of "variant records" (a language feature somewhat similar to C union records)
- Null pointer dereference
- Memory leaks

Other errors, such as assigning an out-of-range value, are related to the ability to declare types with constrained ranges. In C, such errors would need to be detected by user-written code.

In addition, SPARK allows verification of user-provided contracts. Some examples:

- Global annotations allow specifying the effect of a subprogram on global variables.
- Assertions can be inserted to specify a condition which should be true at some execution point.
- Preconditions and postconditions allow specifying conditions that should be true at subprogram entry and exit, respectively. In effect, preconditions and postconditions

can express the specification of a function: what it is supposed to do. Ada's quantification expressions are useful in these contexts, providing a general notation for capturing such conditions.

SPARK can statically check all these conditions. For example, it will detect and report the following situations:

- A subprogram writes to a global variable which is not mentioned in its Global annotation.
- At the point of call for a subprogram, its precondition may evaluate to False.
- At the point of return from a subprogram, its postcondition may evaluate to False.

As a sound tool, SPARK allows establishing additional code properties:

- Global variables are manipulated in a manner consistent with their Global contracts.
- All subprograms implement their specification (if a subprogram is called and its precondition is True, then its postcondition will be True when it returns).
- Subprograms are only called when their specification allows it (i.e., at each point of call, the subprogram's precondition is True).

3.4 Comparison Summary

The following table summarizes the characteristics of the tools in the three categories described above. Tools in all three categories can find bugs, but only sound tools can prove that there are no other bugs of certain classes. Only tools which allow extra annotations/contracts can prove that a code module implements its specification.

	Bug Finders	**Sound Static Analysis**	**Extra Annotations**
Tools	***Coverity, CodeSonar***	***Polyspace, CodePeer, Frama-C (Eva)***	***SPARK, Frama-C (WP)***
Find bugs?	**Yes**	**Yes**	**Yes**
Ensure absence of run-time errors?	**No**	**Yes**	**Yes**
Prove compliance with specification?	**No**	**No**	**Yes**

4 SPARK Compared with Frama-C

4.1 Similarities between SPARK and Frama-C

SPARK and Frama-C are quite similar in their core functionality. The basic idea is that the tools can prove that the analyzed code complies with its specification. This specification must be written by the user as part of the source code and is used by the static analysis tool. It takes the form of a stylized comment in Frama-C (and is thus ignored by the compiler) but in SPARK it is part of the standard Ada syntax (see also the section "The Annotation Language" below).

4.1.1 Assertions

One form of annotation available both in SPARK and Frama-C is the assertion. In Frama-C, it looks like this:

```
x++;
/*@ assert x > 0 ; */
```

In SPARK like this:

```
X := X + 1;
pragma Assert (X > 0);
```

The idea is that at this point in the program the expression in the assertion should be true.

4.1.2 Contracts

In both Frama-C and SPARK, one can specify the expected behavior of a code module using *preconditions* and *postconditions*. A precondition is a property that should be true when the code module is called. Similarly, a postcondition is a property that should be true when the code module returns. Together, the precondition and postcondition form a *contract* between the caller and the callee, where the caller guarantees the precondition, and in exchange the callee guarantees the postcondition.

As a first approximation, pre- and postconditions are simply assertions at the beginning and the end of a code module. However, both Frama-C and SPARK allow extra syntax in postconditions to refer to function results and also to values of variables *at the beginning of the call.* A simple example in Frama-C is the contract for a function incrementing an integer:

```
/*@
requires \valid(x);
requires *x < 2147483647;
assigns *x;
ensures *x == \old(*x) + 1;
*/
void increment_guarded (int *x) {
  (*x)++;
}
```

And the analogous SPARK code would look like this:

```
procedure Increment_Guarded (X : in out Integer) with
  Pre => X < Integer'Last,
  Post => X = X'Old + 1
is begin
   X := X + 1;
end Increment_Guarded;
```

Note the use of the special syntax `\old(*x)` in Frama-C and `X'Old` in SPARK to refer to the "old" value of x, that is on entry to the code module.

4.1.3 Side effects

In both SPARK and in Frama-C, the user specifies the side effects of a code module. The side effects declare:

- which global variables and parameters are read by the code module;
- which global variables and parameters are written by the code module.

This information is essential for the correct analysis of both the code module and all of its invocations. In Frama-C, this is known as an *assigns clause.* In SPARK, each parameter's mode indicates whether the parameter is read, written, or both, and for global variables a *Global data dependency* contract is used.

An important difference between Frama-C and SPARK is that assigns clauses must be written by users in Frama-C (for both functions and loops) while Global contracts are generated in SPARK when not provided by users.

4.1.4 Formal Verification

Formal verification is the process that determines whether the source code is consistent with all its annotations/contracts. Both SPARK and Frama-C will check that:

- a code module only reads and writes the variables that it is allowed to read and write, based on its parameter declarations and its assigns clause or Global contract;
- if the code module is called in a context where its precondition is true, then it will not raise any run-time errors or incur unspecified behavior during execution, for example arithmetic overflow, buffer overflow, or division by zero;
- if the code module is called in a context where its precondition is true, then its postcondition will be true on any path that leads to a return. One can also say that the code module "implements its contract".

SPARK and Frama-C achieve these results only by examining the source code (thus "static analysis"), and if they report no error, none of the above-mentioned errors can ever occur in the analyzed software.

Note that neither SPARK nor Frama-C guarantee the absence of so-called non-functional errors, such as stack overflow and violation of timing properties.

In this sense, SPARK and Frama-C are very similar, because they have the same objectives and achieve them with similar means: formal methods implemented though modern proof technology.

4.1.5 Ghost code

Ghost code is extra code that does not contribute to the functionality of the program, but helps its formal verification. Ghost code is either invisible to the compiler (Frama-C) or can be enabled or disabled using a compiler switch, similar to assertions (SPARK). A separate paper [15] discusses differences in the handling of ghost code between Frama-C and SPARK. For the purposes of this high-level comparison, it is enough to say that both Frama-C and SPARK support ghost code.

4.2 Differences between SPARK and Frama-C

4.2.1 Precise subtypes for integers and floating-point types

In SPARK, in addition to the predefined types which correspond to 8-bit, 16-bit, 32-bit and 64-bit signed and unsigned integers, subtypes of any range can be defined:

```
  subtype Test_Score is Integer range 0 .. 100;
```

SPARK's static analysis will not only verify that variables of this subtype never take values outside of the specified range, but it can also *use* that information, for example to prove that no overflow can occur in computations with values of that type. The same is true for floating-point types.

In C, basically only the predefined integer and floating point types can be used, and no more precise bounds can be specified directly when variables are defined. If a program requires data to have more precise bounds, these bounds have to be specified in the contracts of all functions which manipulate this data.

In an internship project at AdaCore [10], where the C firmware of a small drone was replaced by SPARK code and proved to be free of run-time errors, constrained subtypes were used to deal with overflow of floating-point computations. In Frama-C, these precise bounds would have to be added to all functions manipulating the corresponding variables.

Researchers from the University of Bristol [11] have had a similar experience, stating that "[they] captured most of the required information using types that constrain the ranges of numeric variables".

The Frama-C annotation language has the ability to add properties such as ranges to variables and types as “data invariants” and “type invariants”. But this syntax is apparently not supported by the Frama-C tool at the time of this writing (Vanadium

release of Frama-C [12]).

4.2.2 Pointers and the Memory Model

The aliasing introduced by pointers makes formal verification difficult, and thus SPARK encourages the use of other language features instead, and only allows a form of pointers that do not introduce problematic aliasing. In C, pointers have many uses, for example to pass parameters by reference and to access arrays (in SPARK no pointers are necessary for these), so restricting pointers is unrealistic.

The developers of Frama-C had no choice but to support formal verification for programs containing pointers in their general form. This makes Frama-C a very powerful tool for reasoning about data structures such as linked lists implemented with pointers, or the common array-manipulating programs that are usually pointer-based. However, this power comes at a high price in complexity.

Variables such as integers, records (structs) and arrays, as they are used by a programmer in C or SPARK, ultimately are only abstractions of the memory of the machine. In most cases the programmer need not be concerned with the details. But when different program variables correspond to the same memory locations, then this abstract view breaks down and surprising things can happen. For example, in C, modifying the data pointed to by a pointer variable can in theory modify almost any other variable, unless it is known where the pointer points to.

When distinct program variables correspond to the same or overlapping memory regions, this is called *aliasing*. In C, because of the presence of pointers, aliasing is unavoidable and has to be taken into account by the programmer and by tools such as Frama-C. For example, a function to copy one array into another can be annotated like this in Frama-C (taken from the “ACSL by example” document [13]):

```
/*@
  requires valid_a: \valid_read(a + (0..n-1));
  requires valid_b: \valid(b + (0..n-1));
  requires sep: \separated(a + (0..n-1), b + (0..n-1));
  assigns b[0..n-1];
  ensures equal: EqualRanges{Here,Here}(a, n, b);
*/
void copy(const value_type* a, const size_type n, value_type* b);
```

Because of the typical C use of pointers to represent arrays, the programmer has to specify the length of each array and has to specify that the memory regions of source and destination do not overlap (using the `\separated` notation). The result is a very heavy specification of this simple function.

In SPARK, pointers are not needed to represent arrays, and parameters of a function or procedure cannot be aliased. So the abstraction of the memory in the form of program variables is meaningful, as different variables always denote different objects. This allows the array copy procedure to have a much simpler specification than in Frama-C:

```
procedure Copy (A : in Array_Type; B : out Array_Type)
  with Pre  => A’Length = B’Length,
       Post => A = B;
```

In actuality, the language features of SPARK make copying arrays so simple that one would not write a function for it. But this example illustrates clearly the main differences in the memory models of Frama-C and SPARK. On the one hand, Frama-C allows the analysis of programs which manipulate pointers in arbitrary ways, which SPARK does not allow. On the other hand, the associated complexity makes reasoning about simple code such as basic array manipulations much more difficult than it needs to be. As a result, not only is it costly to specify the contracts of such code, but automatic provers are less likely to succeed in proving that the code implements their contracts.

4.2.3 The Annotation Language

Although the annotation languages of Frama-C and SPARK have many similarities, there are also many differences.

4.2.3.1 Syntax

Syntactically, Frama-C annotations are specially marked comments, namely comments that start with "`/*@`". SPARK contracts take the form of Ada 2012 *aspects*, a language feature for attaching extra information to declared entities. The important aspects `Pre` and `Post` are defined by the Ada 2012 language standard.

The advantage of using comments is that the compiler completely ignores them, so any compiler can be used. Using Ada 2012 aspects instead of comments for annotations requires that an Ada 2012 compiler be used for compiling SPARK programs.

Writing annotations is an additional effort programmers have to provide, and they will be more motivated to do so if many if not all tools of the development process support annotations. From a purely technical point of view, there is no reason why annotations in comments should be less supported than annotations that are part of the language. However, tool developers see it as their obligation to support all language-defined features, while they see other things (e.g. parsing comments) as an extra effort. As a result, many tools such as code-browsing and refactoring tools read and modify the SPARK contracts. The Ada compiler itself can insert assertions for the pre- and postconditions, unit testing tools also use the pre- and postconditions as oracles and to better interpret test failures. On the other hand, to our knowledge, no other tool reads Frama-C annotations.

4.2.3.2 Semantics

The actual assertions, pre- and postconditions in Frama-C, are written in a language that is called ACSL (ANSI/ISO C Specification Language) [12, 13]. ACSL expressions are similar to C expressions, and all the common operators such as arithmetic and pointer operations are present. But there are important subtle differences:

- Integers in ACSL are mathematical integers and floating point values are real variables of infinite precision;
- Regular C functions can not be called inside ACSL expressions; instead extra so-called logic functions must be defined.

Overall, programmers are faced with two different languages, the programming language C and the annotation language ACSL.

In contrast, in SPARK, pre- and postconditions are just boolean expressions: any boolean expression which is allowed in a SPARK program is also allowed in a SPARK contract and has the same meaning. This has been a very important choice early on in the design of the SPARK language and makes the SPARK language easier to learn than other formal methods.

4.2.3.3 Executable Contracts

A very important benefit of SPARK contracts is that they are executable. If the programmer desires (for example for testing), SPARK pre- and postconditions can be checked during the execution of the program; a runtime error is raised when a pre- or postcondition is False. In such a situation, it is immediately clear there is a problem either in the code or in the specification (contract) of the function. Thus contracts are not only useful for formal verification of the SPARK program, but they also increase the usefulness of testing.

Executable contracts also are a possible solution for another problem, namely proved code with code that has not been proved. For example, one can annotate a code module with a precondition and prove that, if it is called when the precondition is true, then no run-time error occurs within the subprogram. But what happens if the code module is called, but the precondition is False? Of course this cannot happen in code that has itself been proved, but it can happen otherwise.

One classical solution, which is also used when no formal verification is applied, is called *defensive code*. Instead of (or in addition to) writing the precondition, code can be added at the beginning of the code module which checks the values of parameters and global variables, for example like this:

```
/*@
requires b != 0;
*/
int divide (int a, int b) {
   if (b == 0) {
     // do some action here to abort the program or report an error
   }
   return a / b;
}
```

Such code is useful even if the function is proved, because other, unproved functions may call it incorrectly with the parameter `b` set to `0`. So there are situations where such defensive code would echo the Frama-C precondition.

In SPARK, the precondition is executable and can act as the defensive code directly:

```
function Divide (A, B : Integer) return Integer
  with Pre => B /= 0
is begin
  return A / B;
end Divide;
```

If compiled with the appropriate compiler switch, the precondition is compiled into an assertion, and the resulting program has the same effect as the defensive code in the C program.

More recently [14], (part of) the ACSL specification language has been made executable, this is called E-ACSL (for Executable ACSL). It requires generating new C-files, which contain extra code for the executable contracts. While E-ACSL provides many of the benefits of executable contracts, some often used parts of ACSL are not part of E-ACSL, making this feature less used in Frama-C than it is in SPARK.

4.2.3.4 Mathematical specifications

SPARK's focus on executable contracts makes it less suitable for tasks when mathematical, inherently non-executable data or concepts are required. This is the case in particular for concepts that users could axiomatize (in SPARK one is generally required to provide an implementation for any helper function), although this can be emulated in SPARK with imported ghost code modules.

While Frama-C manipulates natively mathematical (unbounded) integers and real (infinite precision) numbers in ACSL annotations, SPARK provides mathematical integers and rationals in the standard library of Ada, with all necessary arithmetic operations and conversions to and from machine integers.4.2.3.5 Advanced annotations

SPARK provides some features in the annotation language that are not present in Frama-C:

- Subtype predicates allow users to specify extra constraints on objects of a type. The SPARK tools will check that these constraints are verified at all times. This is useful for validity constraints on data.
- Type invariants allow a package to maintain some data invariant, while allowing the package to break the invariant temporarily. This is useful for internal properties that some package must maintain, that are usually not important to the outside of the package.

4.2.3 Library support

In both Frama-C and SPARK, a wide range of libraries can be used via the underlying programming languages Ada and C. However, if the libraries are not annotated, one cannot generally prove code that uses these libraries. Both in SPARK and Frama-C, few annotated libraries exist.

In Frama-C, part of the C standard library is available in annotated form to be used in Frama-C programs.

In SPARK, part of the standard library of Ada is annotated with contracts (most notably units providing generic containers, I/O, character and string manipulation), and an extra lemma library is provided which helps with common properties that are difficult to prove without such a library.

4.2.4 The SPARK and Frama-C tools

The previous discussion focused on the comparison of the ACSL language with the SPARK language. Another axis of comparison is the tool support for these languages, the features and ease of use of the tools.

Both tools are open source, and both tools are commercially available. Both tools have roughly the same structure and way of working when it comes to proof of program properties. There are some differences though:

- SPARK can unroll loops with a small number of loop iterations, and inline calls to local subprograms to simplify annotation of programs;
- SPARK can generate some parts of loop invariants (the most difficult to master form of annotations, which exists in both Frama-C and SPARK); this greatly helps handling of data in loops;
- SPARK can show counterexamples for some failed proofs, which helps understand why a property is not correct, or why the tool failed to prove it;
- SPARK issues very detailed messages on failed proofs to help users both locate the origin of the problem and correct it;
- SPARK is very well-integrated into the AdaCore development environment GNAT Studio.
- Frama-C is provided in source form. The user has to compile Frama-C, and install supporting tools such as the automated provers Z3 and CVC4, and the Why3 library. This differs from the SPARK tools, that are provided as a complete binary package, ready to be installed on Linux and Windows, containing all the tools that are needed.

4.3 Comparison Summary

SPARK and Frama-C are similar tools for their respective languages. However, the SPARK language brings a number of benefits:

- SPARK contracts are part of the standard Ada 2012 language. There is no need for programmers to learn a separate notation and syntactic errors will be detected by the compiler. Frama-C annotations are a different language that must be learned. Since they are simply comments, errors are only detected by the Frama-C analyzer.
- SPARK contracts for functional correctness are just assertions and can be executed and tested. Thus they can be used by testing tools, not only during static analysis.
- SPARK does not have problematic aliasing, even via pointers, and it imposes other restrictions which make writing contracts much simpler. In particular, two mutable variables can always be considered to denote distinct objects. In Frama-C, one has to explicitly state that two pointers do not point to overlapping memory regions. This makes its annotations rather heavy.

The following table summarizes the comparison between the two technologies:

	Frama-C	**SPARK**

Pre/Postconditions?	**Yes**	**Yes**
Global annotations?	**Yes**	**Yes (Generated)**
Pointers?	**Yes**	**Yes (Ownership)**
Aliasing excluded?	**No**	**Yes**
Precise subtypes?	**No**	**Yes**
Executable annotations?	**Via E-ACSL**	**Yes**
Unbounded integers, reals	**Yes**	**Yes (Library)**
Specification language same as programming language?	**No**	**Yes**
Ghost code?	**Yes**	**Yes**

4.4 Other comparisons

For a comparison of Frama-C and SPARK on a non-trivial codebase, see the presentation of researchers Christophe Garion and Jérôme Hugues at Frama-C & SPARK Day 2017 [18]. They took a fairly large piece of critical software (10,000 sloc in Ada and 15,000 sloc in C), the PolyORB-Hi runtime for distributed software generated from an AADL description, and applied Frama-C on the C version and SPARK on the Ada version to achieve absence of run-time errors and proof of properties. Their conclusion, which matches the initial assessment they did in 2014, is that it is much easier to achieve high assurance through formal program verification in SPARK than in C, mostly because the language really supports it.

Another source of comparison between SPARK and Frama-C is the ACSL by Example by Fraunhofer [13] which presents dozens of examples of C code and how they can be verified with Frama-C/WP. Garion and Hugues have produced an equivalent SPARK by Example [19] which presents the SPARK version of the C algorithms from ACSL by Example.

For a more in-depth discussion of the differences between SPARK and Frama-C, see an article written between AdaCore, CEA and Inria [15], in which we describe the most important differences in terms of specification language, ghost code and counterexamples. See in particular figure 4 on page 15 which summarizes all these differences.

5 Conclusions

This paper compared the SPARK and MISRA C languages with respect to support for high-assurance software development and also compared the SPARK verification technology with Frama-C. In summary, SPARK provides a variety of benefits over MISRA C, in part because Ada was designed from the start to support good software engineering principles whereas “safe” C subsets such as MISRA C are an *a posteriori* attempt to work around deficiencies that are intrinsic to C. Further, the concept of compliance with MISRA C is not well defined, since some of the rules are unenforceable, while the SPARK restrictions in terms of Ada features are well-defined and easily checked.

Notwithstanding the methodological issues with C and MISRA C, the use of C in critical embedded systems has motivated the development of technologies for formal analysis and proofs of correctness of C programs, most notably Frama-C. Again, we feel that the SPARK technology is the more appropriate choice in practice: it does not require a special annotation language, it supports contracts that can be checked either statically or at run-time, and the SPARK restrictions (in particular the ownership policy for pointers) eliminate the possibility of problematic aliasing and make the verification process much simpler than with Frama-C.

References

[1] Ada Reference Manual

[2] NASA Software Guidebook

[3] ISO PDTR 15942 *Programming languages - guide for the use of the Ada Programming language in High-Integrity systems*

[4] ISO/IEC Technical report 24772, 2013 - *Information Technology - Programming Languages - Guidance to avoiding vulnerabilities in programming languages through language selection and use*

[5] SPARK Reference Manual

[6] SPARK User’s Guide

[7] C11 Standard

[8] MISRA C:2012 standard

[9] MISRA C 2012 vs SPARK 2014, the Subset Matching Game

[10] Blog Post by Anthony Leonardo Gracio

[11] P. Trojanek, K. Eder: Verification and testing of mobile robot navigation algorithms: A case study in SPARK, *Robots and Systems* (IROS 2014)

[12] Patrick Baudin et. al.:, *ACSL: ANSI/ISO C Specification Language Version 1.17*

[13] Jochen Burghardt, Jens Gerlach and Timon Lapawczyk: *ACSL By Example* version 22.0.0

[14] E-ACSL

[15] Nikolai Kosmatov, Claude Marché, Yannick Moy, Julien Signoles: Static versus Dynamic Verification in Why3, Frama-C and SPARK 2014 7th International Symposium on Leveraging Applications, Oct 2016.

[16] What's the Difference Between Sound and Unsound Static Analysis?

[17] TrustInSoft Website

[18] Christophe Garion and Jérôme Hugues at Frama-C & SPARK Day 2017 - https://frama-c.com/download/framaCDay/FCSD17/talk/02_Garion_Hugues.pdf

[19] Christophe Garion and Jérôme Hugues, SPARK by Example - https://github.com/tofgarion/spark-by-example