LEARN.
ADACORE.COM

SPARK for the MISRA-C Developer

Release 2020-05

Yannick Moy

Sep 03, 2020

CONTENTS:

This book presents the SPARK technology — the SPARK subset of Ada and its supporting static analysis tools — through an example-driven comparison with the rules in the widely known MISRA C subset of the C language.

This document was prepared by Yannick Moy, with contributions and review from Ben Brosgol.

[1] http://creativecommons.org/licenses/by-sa/4.0

CHAPTER

ONE

PREFACE

MISRA C appeared in 1998 as a coding standard for C; it focused on avoiding error-prone programming features of the C programming language rather than on enforcing a particular programming style. A study of coding standards for C by Les Hatton[2] found that, compared to ten typical coding standards for C, MISRA C was the only one to focus exclusively on error avoidance rather than style enforcement, and by a very large margin.

The popularity of the C programming language, as well as its many traps and pitfalls, have led to the huge success of MISRA C in domains where C is used for high-integrity sofware. This success has driven tool vendors to propose many competing implementations of MISRA C checkers[3]. Tools compete in particular on the coverage of MISRA C guidelines that they help enforce, as it is impossible to enforce the 16 directives and 143 rules (collectively referred to as guidelines) of MISRA C.

The 16 directives are broad guidelines, and it is not possible to define compliance in a unique and automated way. For example, *"all code should be traceable to documented requirements"* (Directive 3.1). Thus no tool is expected to enforce directives, as the MISRA C:2012 states in introduction to the guidelines: *"different tools may place widely different interpretations on what constitutes a non-compliance."*

The 143 rules on the contrary are completely and precisely defined, and *"static analysis tools should be capable of checking compliance with rules"*. But the same sentence continues with *"subject to the limitations described in Section 6.5"*, which addresses "decidability of rules". It turns out that 27 rules out of 143 are not decidable, so no tool can always detect all violations of these rules without at the same time reporting "false alarms" on code that does not constitute a violation.

An example of an undecidable rule is rule 1.3: *"There shall be no occurrence of undefined or critical unspecified behaviour."* Appendix H of MISRA:C 2012 lists hundreds of cases of undefined and critical unspecified behavior in the C programming language standard, a majority of which are not individually decidable. For the most part, MISRA C checkers ignore undecidable rules such as rule 1.3 and instead focus on the 116 rules for which detection of violations can be automated. It is telling in that respect that the MISRA C:2012 document and its accompanying set of examples (which can be downloaded from the MISRA website[4]) does not provide any example for rule 1.3.

However, violations of undecidable rules such as rule 1.3 are known to have dramatic impact on software quality. Violations of rule 1.3 in particular are commonly amplified by compilers using the permission in the C standard to optimize aggressively without looking at the consequences for programs with undefined or critical unspecified behavior. It would be valid to ignore these rules if violations did not occur in practice, but on the contrary even experienced programmers write C code with undefined or critical unspecified behavior. An example comes from the MISRA C Committee itself in its "Appendix I: Example deviation record" of the MISRA C:2012 document, repeated in "Appendix A: Example deviation record" of the MISRA C: Compliance 2016 document[5], where the following code is proposed as a deviation of rule 10.6 *"The value of a composite expression shall not be assigned to an object with wider essential type"*:

[2] https://www.leshatton.org/Documents/MISRAC.pdf
[3] https://en.wikipedia.org/wiki/MISRA_C
[4] https://www.misra.org.uk
[5] https://www.misra.org.uk/LinkClick.aspx?fileticket=w_Syhpkf7xA%3d&tabid=57

```
uint32_t prod = qty * time_step;
```

Here, the multiplication of two unsigned 16-bit values and assignment of the result to an unsigned 32-bit variable constitutes a violation of the aforementioned rule, which gets justified for efficiency reasons. What the authors seem to have missed is that the multiplication is then performed with the signed integer type `int` instead of the target unsigned type `uint32_t`. Thus the multiplication of two unsigned 16-bit values may lead to an overflow of the 32-bit intermediate signed result, which is an occurrence of an undefined behavior. In such a case, a compiler is free to assume that the value of `prod` cannot exceed $2^{31} - 1$ (the maximal value of a signed 32-bit integer) as otherwise an undefined behavior would have been triggered. For example, the undefined behavior with values 65535 for `qty` and `time_step` is reported when running the code compiled by either the GCC or LLVM compiler with option `-fsanitize=undefined`.

The MISRA C checkers that detect violations of undecidable rules are either unsound tools that can detect only some of the violations, or sound tools that guarantee to detect all such violations at the cost of possibly many false reports of violations. This is a direct consequence of undecidability. However, static analysis technology is available that can achieve soundness without inundating users with false alarms. One example is the SPARK toolset developed by AdaCore, Altran and Inria, which is based on four principles:

- The base language Ada provides a solid foundation for static analysis through a well-defined language standard, strong typing and rich specification features.
- The SPARK subset of Ada restricts the base language in essential ways to support static analysis, by controlling sources of ambiguity such as side-effects and aliasing.
- The static analysis tools work mostly at the granularity of an individual function, making the analysis more precise and minimizing the possibility of false alarms.
- The static analysis tools are interactive, allowing users to guide the analysis if necessary or desired.

In this document, we show how SPARK can be used to achieve high code quality with guarantees that go beyond what would be feasible with MISRA C.

An on-line and interactive version of this document is available at AdaCore's learn.adacore.com site[6].

[6] https://learn.adacore.com/courses/SPARK_for_the_MISRA_C_Developer

CHAPTER

TWO

ENFORCING BASIC PROGRAM CONSISTENCY

Many consistency properties that are taken for granted in other languages are not enforced in C. The basic property that all uses of a variable or function are consistent with its type is not enforced by the language and is also very difficult to enforce by a tool. Three features of C contribute to that situation:

- the textual-based inclusion of files means that every included declaration is subject to a possibly different reinterpretation depending on context.
- the lack of consistency requirements across translation units means that type inconsistencies can only be detected at link time, something linkers are ill-equipped to do.
- the default of making a declaration externally visible means that declarations that should be local will be visible to the rest of the program, increasing the chances for inconsistencies.

MISRA C contains guidelines on all three fronts to enforce basic program consistency.

2.1 Taming Text-Based Inclusion

The text-based inclusion of files is one of the dated idiosyncracies of the C programming language that was inherited by C++ and that is known to cause quality problems, especially during maintenance. Although multiple inclusion of a file in the same translation unit can be used to emulate template programming, it is generally undesirable. Indeed, MISRA C defines Directive 4.10 precisely to forbid it for header files: *"Precautions shall be taken in order to prevent the contents of a header file being included more than once"*.

The subsequent section on "Preprocessing Directives" contains 14 rules restricting the use of text-based inclusion through preprocessing. Among other things these rules forbid the use of the `#undef` directive (which works around conflicts in macro definitions introduced by text-based inclusion) and enforces the well-known practice of enclosing macro arguments in parentheses (to avoid syntactic reinterpretations in the context of the macro use).

SPARK (and more generally Ada) does not suffer from these problems, as it relies on semantic inclusion of context instead of textual inclusion of content, using `with` clauses:

```
with Ada.Text_IO;

procedure Hello_World is
begin
   Ada.Text_IO.Put_Line ("hello, world!");
end Hello_World;
```

Note that `with` clauses are only allowed at the beginning of files; the compiler issues an error if they are used elsewhere:

```
procedure Hello_World is
   with Ada.Text_IO; -- Illegal
```

(continues on next page)

(continued from previous page)

```
begin
   Ada.Text_IO.Put_Line ("hello, world!");
end Hello_World;
```

Importing a unit (i.e., specifying it in a `with` clause) multiple times is harmless, as it is equivalent to importing it once, but a compiler warning lets us know about the redundancy:

```
with Ada.Text_IO;
with Ada.Text_IO; -- Legal but useless

procedure Hello_World is
begin
   Ada.Text_IO.Put_Line ("hello, world!");
end Hello_World;
```

The order in which units are imported is irrelevant. All orders are valid and have the same semantics.

No conflict arises from importing multiple units, even if the same name is defined in several, since each unit serves as namespace for the entities which it defines. So we can define our own version of `Put_Line` in some `Helper` unit and import it together with the standard version defined in `Ada.Text_IO`:

```
package Helper is
   procedure Put_Line (S : String);
end Helper;
```

```
with Ada.Text_IO;

package body Helper is
   procedure Put_Line (S : String) is
   begin
      Ada.Text_IO.Put_Line ("Start helper version");
      Ada.Text_IO.Put_Line (S);
      Ada.Text_IO.Put_Line ("End helper version");
   end Put_Line;
end Helper;
```

```
with Ada.Text_IO;
with Helper;

procedure Hello_World is
begin
   Ada.Text_IO.Put_Line ("hello, world!");
   Helper.Put_Line ("hello, world!");
end Hello_World;
```

The only way a conflict can arise is if we want to be able to reference `Put_Line` directly, without using the qualified name `Ada.Text_IO.Put_Line` or `Helper.Put_Line`. The `use` clause makes public declarations from a unit available directly:

```
package Helper is
   procedure Put_Line (S : String);
end Helper;
```

```
with Ada.Text_IO;

package body Helper is
   procedure Put_Line (S : String) is
```

(continues on next page)

(continued from previous page)

```
   begin
      Ada.Text_IO.Put_Line ("Start helper version");
      Ada.Text_IO.Put_Line (S);
      Ada.Text_IO.Put_Line ("End helper version");
   end Put_Line;
end Helper;
```

```
with Ada.Text_IO; use Ada.Text_IO;
with Helper; use Helper;

procedure Hello_World is
begin
   Ada.Text_IO.Put_Line ("hello, world!");
   Helper.Put_Line ("hello, world!");
   Put_Line ("hello, world!");  --  ERROR
end Hello_World;
```

Here, both units `Ada.Text_IO` and `Helper` define a procedure `Put_Line` taking a `String` as argument, so the compiler cannot disambiguate the direct call to `Put_Line` and issues an error. Here is output from AdaCore's GNAT Ada compiler:

```
1.     with Ada.Text_IO; use Ada.Text_IO;
2.     with Helper; use Helper;
3.
4.     procedure Hello_World is
5.     begin
6.        Ada.Text_IO.Put_Line ("hello, world!");
7.        Helper.Put_Line ("hello, world!");
8.        Put_Line ("hello, world!");  --  ERROR
          |
   >>> ambiguous expression (cannot resolve "Put_Line")
   >>> possible interpretation at helper.ads:2
   >>> possible interpretation at a-textio.ads:508

9.     end Hello_World;
```

Note that it helpfully points to candidate declarations, so that the user can decide which qualified name to use as in the previous two calls.

Issues arising in C as a result of text-based inclusion of files are thus completely prevented in SPARK (and Ada) thanks to semantic import of units. Note that the C++ committee identified this weakness some time ago and has approved[7] the addition of *modules* to C++20, which provide a mechanism for semantic import of units.

2.2 Hardening Link-Time Checking

An issue related to text-based inclusion of files is that there is no single source for declaring the type of a variable or function. If a file `origin.c` defines a variable `var` and functions `fun` and `print`:

```
#include <stdio.h>

int var = 0;
int fun() {
   return 1;
}
```

(continues on next page)

[7] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/n4720.pdf

(continued from previous page)

```
void print() {
   printf("var = %d\n", var);
}
```

and the corresponding header file `origin.h` declares `var`, `fun` and `print` as having external linkage:

```
extern int var;
extern int fun();
extern void print();
```

then client code can include `origin.h` with declarations for `var` and `fun`:

```
#include "origin.h"

int main() {
   var = fun();
   print();
   return 0;
}
```

or, equivalently, repeat these declarations directly:

```
extern int var;
extern int fun();
extern void print();

int main() {
   var = fun();
   print();
   return 0;
}
```

Then, if an inconsistency is introduced in the type of `var` of `fun` between these alternative declarations and their actual type, the compiler cannot detect it. Only the linker, which has access to the set of object files for a program, can detect such inconsistencies. However, a linker's main task is to link, not to detect inconsistencies, and so inconsistencies in the type of variables and functions in most cases cannot be detected. For example, most linkers cannot detect if the type of `var` or the return type of `fun` is changed to `float` in the declarations above. With the declaration of `var` changed to `float`, the above program compiles and runs without errors, producing the erroneous output `var = 1065353216` instead of `var =1`. With the return type of `fun` changed to `float` instead, the program still compiles and runs without errors, producing this time the erroneous output `var = 0`.

The inconsistency just discussed is prevented by MISRA C Rule 8.3 *"All declarations of an object or function shall use the same names and type qualifiers"*. This is a decidable rule, but it must be enforced at system level, looking at all translation units of the complete program. MISRA C Rule 8.6 also requires a unique definition for a given identifier across translation units, and Rule 8.5 requires that an external declaration shared between translation units comes from the same file. There is even a specific section on "Identifiers" containing 9 rules requiring uniqueness of various categories of identifiers.

SPARK (and more generally Ada) does not suffer from these problems, as it relies on semantic inclusion of context using `with` clauses to provide a unique declaration for each entity.

2.3 Going Towards Encapsulation

Many problems in C stem from the lack of encapsulation. There is no notion of namespace that would allow a file to make its declarations available without risking a conflict with other files. Thus MISRA C has a number of guidelines that discourage the use of external declarations:

- Directive 4.8 encourages hiding the definition of structures and unions in implementation files (`.c` files) when possible: *"If a pointer to a structure or union is never dereferenced within a translation unit, then the implementation of the object should be hidden."*
- Rule 8.7 forbids the use of external declarations when not needed: *"Functions and objects should not be defined with external linkage if they are referenced in only one translation unit."*
- Rule 8.8 forces the explicit use of keyword `static` when appropriate: *"The static storage class specifier shall be used in all declarations of objects and functions that have internal linkage."*

The basic unit of modularization in SPARK, as in Ada, is the *package*. A package always has a spec (in an `.ads` file), which defines the interface to other units. It generally also has a body (in an `.adb` file), which completes the spec with an implementation. Only declarations from the package spec are visible from other units when they import (`with`) the package. In fact, only declarations from what is called the "visible part" of the spec (before the keyword `private`) are visible from units that `with` the package.

```
package Helper is
   procedure Public_Put_Line (S : String);
private
   procedure Private_Put_Line (S : String);
end Helper;
```

```
with Ada.Text_IO;

package body Helper is
   procedure Public_Put_Line (S : String) is
   begin
      Ada.Text_IO.Put_Line (S);
   end Public_Put_Line;

   procedure Private_Put_Line (S : String) is
   begin
      Ada.Text_IO.Put_Line (S);
   end Private_Put_Line;

   procedure Body_Put_Line (S : String) is
   begin
      Ada.Text_IO.Put_Line (S);
   end Body_Put_Line;
end Helper;
```

```
with Helper; use Helper;

procedure Hello_World is
begin
   Public_Put_Line ("hello, world!");
   Private_Put_Line ("hello, world!");  --  ERROR
   Body_Put_Line ("hello, world!");  --  ERROR
end Hello_World;
```

Here's the output from AdaCore's GNAT compiler:

```
1.     with Helper; use Helper;
2.
```

(continues on next page)

(continued from previous page)

```
3.     procedure Hello_World is
4.     begin
5.        Public_Put_Line ("hello, world!");
6.        Private_Put_Line ("hello, world!");  --  ERROR
          |
   >>> "Private_Put_Line" is not visible
   >>> non-visible (private) declaration at helper.ads:4

7.        Body_Put_Line ("hello, world!");  --  ERROR
          |
   >>> "Body_Put_Line" is undefined

8.     end Hello_World;
```

Note the different errors on the calls to the private and body versions of `Put_Line`. In the first case the compiler can locate the candidate procedure but it is illegal to call it from `Hello_World`, in the second case the compiler does not even know about any `Body_Put_Line` when compiling `Hello_World` since it only looks at the spec and not the body.

SPARK (and Ada) also allow defining a type in the private part of a package spec while simply declaring the type name in the public ("visible") part of the spec. This way, client code — i.e., code that `with`'s the package — can use the type, typically through a public API, but have no access to how the type is implemented:

```
package Vault is
   type Data is private;
   function Get (X : Data) return Integer;
   procedure Set (X : out Data; Value : Integer);
private
   type Data is record
      Val : Integer;
   end record;
end Vault;
```

```
package body Vault is
   function Get (X : Data) return Integer is (X.Val);
   procedure Set (X : out Data; Value : Integer) is
   begin
      X.Val := Value;
   end Set;
end Vault;
```

```
with Vault;

package Information_System is
   Archive : Vault.Data;
end Information_System;
```

```
with Information_System;
with Vault;

procedure Hacker is
   V : Integer := Vault.Get (Information_System.Archive);
begin
   Vault.Set (Information_System.Archive, V + 1);
   Information_System.Archive.Val := 0;  --  ERROR
end Hacker;
```

Note that it is possible to declare a variable of type `Vault.Data` in package `Information_System` and to get/set it through its API in procedure `Hacker`, but not to directly access its `Val` field.

CHAPTER

THREE

ENFORCING BASIC SYNTACTIC GUARANTEES

C's syntax is concise but also very permissive, which makes it easy to write programs whose effect is not what was intended. MISRA C contains guidelines to:

- clearly distinguish code from comments
- specially handle function parameters and result
- ensure that control structures are not abused

3.1 Distinguishing Code and Comments

The problem arises from block comments in C, starting with `/*` and ending with `*/`. These comments do not nest with other block comments or with line comments. For example, consider a block comment surrounding three lines that each increase variable `a` by one:

```
/*
++a;
++a;
++a; */
```

Now consider what happens if the first line is commented out using a block comment and the third line is commented out using a line comment (also known as a C++ style comment, allowed in C since C99):

```
/*
/* ++a; */
++a;
// ++a; */
```

The result of commenting out code that was already commented out is that the second line of code becomes live! Of course, the above example is simplified, but similar situations do arise in practice, which is the reason for MISRA C Directive 4.1 *"Sections of code should not be 'commented out'"*. This is reinforced with Rules 3.1 and 3.2 from the section on "Comments" that forbid in particular the use of `/*` inside a comment like we did above.

These situations cannot arise in SPARK (or in Ada), as only line comments are permitted, using `--`:

```
--  A := A + 1;
--  A := A + 1;
--  A := A + 1;
```

So commenting again the first and third lines does not change the effect:

```
--  --  A := A + 1;
--  A := A + 1;
--  --  A := A + 1;
```

3.2 Specially Handling Function Parameters and Result

3.2.1 Handling the Result of Function Calls

It is possible in C to ignore the result of a function call, either implicitly or else explicitly by converting the result to `void`:

```
f();
(void)f();
```

This is particularly dangerous when the function returns an error status, as the caller is then ignoring the possibility of errors in the callee. Thus the MISRA C Directive 4.7: *"If a function returns error information, then that error information shall be tested"*. In the general case of a function returning a result which is not an error status, MISRA C Rule 17.7 states that *"The value returned by a function having non-void return type shall be used"*, where an explicit conversion to `void` counts as a use.

In SPARK, as in Ada, the result of a function call must be used, for example by assigning it to a variable or by passing it as a parameter, in contrast with procedures (which are equivalent to void-returning functions in C). SPARK analysis also checks that the result of the function is actually used to influence an output of the calling subprogram. For example, the first two calls to F in the following are detected as unused, even though the result of the function call is assigned to a variable, which is itself used in the second case:

```
package Fun is
   function F return Integer is (1);
end Fun;
```

```
with Fun; use Fun;

procedure Use_F (Z : out Integer) is
   X, Y : Integer;
begin
   X := F;

   Y := F;
   X := Y;

   Z := F;
end Use_F;
```

Only the result of the third call is used to influence the value of an output of `Use_F`, here the output parameter Z of the procedure.

3.2.2 Handling Function Parameters

In C, function parameters are treated as local variables of the function. They can be modified, but these modifications won't be visible outside the function. This is an opportunity for mistakes. For example, the following code, which appears to swap the values of its parameters, has in reality no effect:

```
void swap (int x, int y) {
   int tmp = x;
   x = y;
   y = tmp;
}
```

MISRA C Rule 17.8 prevents such mistakes by stating that *"A function parameter should not be modified"*.

No such rule is needed in SPARK, since function parameters are only inputs so cannot be modified, and procedure parameters have a *mode* defining whether they can be modified or not. Only parameters of mode `out` or ada:*in out* can be modified — and these are prohibited from functions in SPARK — and their modification is visible at the calling site. For example, assigning to a parameter of mode `in` (the default parameter mode if omitted) results in compilation errors:

```
procedure Swap (X, Y : Integer) is
   Tmp : Integer := X;
begin
   X := Y;  --  ERROR
   Y := Tmp;  --  ERROR
end Swap;
```

Here is the output of AdaCore's GNAT compiler:

```
1.     procedure Swap (X, Y : Integer) is
2.        Tmp : Integer := X;
3.     begin
4.        X := Y;  --  ERROR
          |
   >>> assignment to "in" mode parameter not allowed

5.        Y := Tmp;  --  ERROR
          |
   >>> assignment to "in" mode parameter not allowed

6.     end Swap;
```

The correct version of `Swap` in SPARK takes parameters of mode `in out`:

```
procedure Swap (X, Y : in out Integer) is
   Tmp : constant Integer := X;
begin
   X := Y;
   Y := Tmp;
end Swap;
```

3.3 Ensuring Control Structures Are Not Abused

The previous issue (ignoring the result of a function call) is an example of a control structure being abused, due to the permissive syntax of C. There are many such examples, and MISRA C contains a number of guidelines to prevent such abuse.

3.3.1 Preventing the Semicolon Mistake

Because a semicolon can act as a statement, and because an if-statement and a loop accept a simple statement (possibly only a semicolon) as body, inserting a single semicolon can completely change the behavior of the code:

```
int func() {
   if (0)
      return 1;
   while (1)
      return 0;
   return 0;
}
```

As written, the code above returns with status 0. If a semicolon is added after the first line (`if (0);`), then the code returns with status 1. If a semicolon is added instead after the third line (`while (1);`), then the code does not return. To prevent such surprises, MISRA C Rule 15.6 states that *"The body of an iteration-statement or a selection-statement shall be a compound statement"* so that the code above must be written:

```
int func() {
   if (0) {
      return 1;
   }
   while (1) {
      return 0;
   }
   return 0;
}
```

Note that adding a semicolon after the test of the `if` or `while` statement has the same effect as before! But doing so would violate MISRA C Rule 15.6.

In SPARK, the semicolon is not a statement by itself, but rather a marker that terminates a statement. The null statement is an explicit `null;`, and all blocks of statements have explicit `begin` and `end` markers, which prevents mistakes that are possible in C. The SPARK (also Ada) version of the above C code is as follows:

```
function Func return Integer is
begin
   if False then
      return 1;
   end if;
   while True loop
      return 0;
   end loop;
   return 0;
end Func;
```

3.3.2 Avoiding Complex Switch Statements

Switch statements are well-known for being easily misused. Control can jump to any case section in the body of the switch, which in C can be before any statement contained in the body of the switch. At the end of the sequence of statements associated with a case, execution continues with the code that follows unless a break is encountered. This is a recipe for mistakes, and MISRA C enforces a simpler *well-formed* syntax for switch statements defined in Rule 16.1: *"All switch statements shall be well-formed"*.

The other rules in the section on "Switch statements" go on detailing individual consequences of Rule 16.1. For example Rule 16.3 forbids the fall-through from one case to the next: *"An unconditional break statement shall terminate every switch-clause"*. As another example, Rule 16.4 mandates the presence of a default case to handle cases not taken into account explicitly: *"Every switch statement shall have a default label"*.

The analog of the C switch statements in SPARK (and in Ada) is the case statement. This statement has a simpler and more robust structure than the C switch, with control automatically exiting after one of the case alternatives is executed, and the compiler checking that the alternatives are disjoint (like in C) and complete (unlike in C). So the following code is rejected by the compiler:

```
package Sign_Domain is

   type Sign is (Negative, Zero, Positive);

   function Opposite (A : Sign) return Sign is
```

(continues on next page)

(continued from previous page)

```
     (case A is  --  ERROR
         when Negative => Positive,
         when Positive => Negative);

   function Multiply (A, B : Sign) return Sign is
     (case A is
         when Negative        => Opposite (B),
         when Zero | Positive => Zero,
         when Positive        => B);  --  ERROR

   procedure Get_Sign (X : Integer; S : out Sign);

end Sign_Domain;
```

```
package body Sign_Domain is

   procedure Get_Sign (X : Integer; S : out Sign) is
   begin
      case X is
         when 0 => S := Zero;
         when others => S := Negative;  --  ERROR
         when 1 .. Integer'Last => S := Positive;
      end case;
   end Get_Sign;

end Sign_Domain;
```

The error in function `Opposite` is that the `when` choices do not cover all values of the target expression. Here, `A` is of the enumeration type `Sign`, so all three values of the enumeration must be covered.

The error in function `Multiply` is that `Positive` is covered twice, in the second and the third alternatives. This is not allowed.

The error in procedure `Get_Sign` is that the `others` choice (the equivalent of C `default` case) must come last. Note that an `others` choice would be useless in `Opposite` and `Multiply`, as all `Sign` values are covered.

Here is a correct version of the same code:

```
package Sign_Domain is

   type Sign is (Negative, Zero, Positive);

   function Opposite (A : Sign) return Sign is
     (case A is
         when Negative => Positive,
         when Zero     => Zero,
         when Positive => Negative);

   function Multiply (A, B : Sign) return Sign is
     (case A is
         when Negative => Opposite (B),
         when Zero     => Zero,
         when Positive => B);

   procedure Get_Sign (X : Integer; S : out Sign);

end Sign_Domain;
```

```
package body Sign_Domain is

   procedure Get_Sign (X : Integer; S : out Sign) is
   begin
      case X is
         when 0 => S := Zero;
         when 1 .. Integer'Last => S := Positive;
         when others => S := Negative;
      end case;
   end Get_Sign;

end Sign_Domain;
```

3.3.3 Avoiding Complex Loops

Similarly to C switches, for-loops in C can become unreadable. MISRA C thus enforces a simpler *well-formed* syntax for for-loops, defined in Rule 14.2: *"A for loop shall be well-formed"*. The main effect of this simplification is that for-loops in C look like for-loops in SPARK (and in Ada), with a *loop counter* that is incremented or decremented at each iteration. Section 8.14 defines precisely what a loop counter is:

1. It has a scalar type;
2. Its value varies monotonically on each loop iteration; and
3. It is used in a decision to exit the loop.

In particular, Rule 14.2 forbids any modification of the loop counter inside the loop body. Here's the example used in MISRA C:2012 to illustrate this rule:

```
bool_t flag = false;

for ( int16_t i = 0; ( i < 5 ) && !flag; i++ )
{
  if ( C )
  {
    flag = true; /* Compliant - allows early termination of loop */
  }

  i = i + 3;     /* Non-compliant - altering the loop counter */
}
```

The equivalent SPARK (and Ada) code does not compile, because of the attempt to modify the value of the loop counter:

```
procedure Well_Formed_Loop (C : Boolean) is
   Flag : Boolean := False;
begin
   for I in 0 .. 4 loop
      exit when not Flag;

      if C then
         Flag := True;
      end if;

      I := I + 3;  --  ERROR
   end loop;
end Well_Formed_Loop;
```

Removing the problematic line leads to a valid program. Note that the additional condition being tested in the C for-loop has been moved to a separate exit statement at the start of the loop body.

SPARK (and Ada) loops can increase (or, with explicit syntax, decrease) the loop counter by 1 at each iteration.

```
for I in reverse 0 .. 4 loop
   ... -- Successive values of I are 4, 3, 2, 1, 0
end loop;
```

SPARK loops can iterate over any discrete type; i.e., integers as above or enumerations:

```
type Sign is (Negative, Zero, Positive);

for S in Sign loop
  ...
end loop;
```

3.3.4 Avoiding the Dangling Else Issue

C does not provide a closing symbol for an if-statement. This makes it possible to write the following code, which appears to try to return the absolute value of its argument, while it actually does the opposite:

```
#include <stdio.h>

int absval (int x) {
   int result = x;
   if (x >= 0)
      if (x == 0)
         result = 0;
   else
      result = -x;
   return result;
}

int main() {
   printf("absval(5) = %d\n", absval(5));
   printf("absval(0) = %d\n", absval(0));
   printf("absval(-10) = %d\n", absval(-10));
}
```

The warning issued by GCC or LLVM with option `-Wdangling-else` (implied by `-Wall`) gives a clue about the problem: although the `else` branch is written as though it completes the outer if-statement, in fact it completes the inner if-statement.

MISRA C Rule 15.6 avoids the problem: *"The body of an iteration-statement or a selection-statement shall be a compound statement"*. That's the same rule as the one shown earlier for *Preventing the Semicolon Mistake* (page 13). So the code for `absval` must be written:

```
#include <stdio.h>

int absval (int x) {
   int result = x;
   if (x >= 0) {
      if (x == 0) {
         result = 0;
      }
   } else {
      result = -x;
   }
   return result;
}
```

(continues on next page)

(continued from previous page)

```
int main() {
   printf("absval(5) = %d\n", absval(5));
   printf("absval(0) = %d\n", absval(0));
   printf("absval(-10) = %d\n", absval(-10));
}
```

which has the expected behavior.

In SPARK (as in Ada), each if-statement has a matching end marker `end if;` so the dangling-else problem cannot arise. The above C code is written as follows:

```
function Absval (X : Integer) return Integer is
   Result : Integer := X;
begin
   if X >= 0 then
      if X = 0 then
         Result := 0;
      end if;
   else
      Result := -X;
   end if;
   return Result;
end Absval;
```

Interestingly, SPARK analysis detects here that the negation operation on line 9 might overflow. That's an example of runtime error detection which will be covered in the chapter on *Detecting Undefined Behavior* (page 43).

CHAPTER

FOUR

ENFORCING STRONG TYPING

Annex C of MISRA C:2012 summarizes the problem succinctly:

> *"ISO C may be considered to exhibit poor type safety as it permits a wide range of implicit type conversions to take place. These type conversions can compromise safety as their implementation-defined aspects can cause developer confusion."*

The most severe consequences come from inappropriate conversions involving pointer types, as they can cause memory safety violations. Two sections of MISRA C are dedicated to these issues: "Pointer type conversions" (9 rules) and "Pointers and arrays" (8 rules).

Inappropriate conversions between scalar types are only slightly less severe, as they may introduce arbitrary violations of the intended functionality. MISRA C has gone to great lengths to improve the situation, by defining a stricter type system on top of the C language. This is described in Appendix D of MISRA C:2012 and in the dedicated section on "The essential type model" (8 rules).

4.1 Enforcing Strong Typing for Pointers

Pointers in C provide a low-level view of the addressable memory as a set of integer addresses. To write at address 42, just go through a pointer:

```
int main() {
   int *p = 42;
   *p = 0;
   return 0;
}
```

Running this program is likely to hit a segmentation fault on an operating system, or to cause havoc in an embedded system, both because address 42 will not be correctly aligned on a 32-bit or 64-bit machine and because this address is unlikely to correspond to valid addressable data for the application. The compiler might issue a helpful warning on the above code (with option `-Wint-conversion` implied by `-Wall` in GCC or LLVM), but note that the warning disappears when explicitly converting value 42 to the target pointer type, although the problem is still present.

Beyond their ability to denote memory addresses, pointers are also used in C to pass references as inputs or outputs to function calls, to construct complex data structures with indirection or sharing, and to denote arrays of elements. Pointers are thus at once pervasive, powerful and fragile.

4.1.1 Pointers Are Not Addresses

In an attempt to rule out issues that come from direct addressing of memory with pointers, MISRA C states in Rule 11.4 that *"A conversion should not be performed between a pointer to object and an integer type"*. As this rule is classified as only Advisory, MISRA C completes it with two Required rules:

- Rule 11.6: *"A cast shall not be performed between pointer to void and an arithmetic type"*

- Rule 11.7: *"A cast shall not be performed between pointer to object and a non-integer arithmetic type"*

In Ada, pointers are not addresses, and addresses are not integers. An opaque standard type `System.Address` is used for addresses, and conversions to/from integers are provided in a standard package `System.Storage_Elements`. The previous C code can be written as follows in Ada:

```
with System;
with System.Storage_Elements;

procedure Pointer is
   A : constant System.Address := System.Storage_Elements.To_Address (42);
   M : aliased Integer with Address => A;
   P : constant access Integer := M'Access;
begin
   P.all := 0;
end Pointer;
```

The integer value 42 is converted to a memory address `A` by calling `System.Storage_Elements.To_Address`, which is then used as the address of integer variable `M`. The pointer variable `P` is set to point to `M` (which is allowed because `M` is declared as `aliased`).

Ada requires more verbiage than C:

- The integer value 42 must be explicitly converted to type `Address`
- To get a pointer to a declared variable such as `M`, the declaration must be marked as `aliased`

The added syntax helps first in making clear what is happening and, second, in ensuring that a potentially dangerous feature (assigning to a value at a specific machine address) is not used inadvertently.

The above example is legal Ada but not SPARK, since SPARK does not support pointers (they significantly complicate formal analysis). SPARK does allow addresses, however.

4.1.2 Pointers Are Not References

Passing parameters by reference is critical for efficient programs, but the absence of references distinct from pointers in C incurs a serious risk. Any parameter of a pointer type can be copied freely to a variable whose lifetime is longer than the object pointed to, a problem known as "dangling pointers". MISRA C forbids such uses in Rule 18.6: *"The address of an object with automatic storage shall not be copied to another object that persists after the first object has ceased to exist"*. Unfortunately, enforcing this rule is difficult, as it is undecidable.

In SPARK, parameters can be passed by reference, but no pointer to the parameter can be stored, which completely solves this issue. In fact, the decision to pass a parameter by copy or by reference rests in many cases with the compiler, but such compiler dependency has no effect on the functional behavior of a SPARK program. In the example below, the compiler may decide to pass parameter `P` of procedure `Rotate_X` either by copy or by reference, but regardless of the choice the postcondition of `Rotate_X` will hold: the final value of `P` will be modified by rotation around the `X` axis.

```
package Geometry is

   type Point_3D is record
      X, Y, Z : Float;
   end record;

   procedure Rotate_X (P : in out Point_3D) with
     Post => P = P'Old'Update (Y => P.Z'Old, Z => -P.Y'Old);

end Geometry;
```

```
package body Geometry is

   procedure Rotate_X (P : in out Point_3D) is
      Tmp : constant Float := P.Y;
   begin
      P.Y := P.Z;
      P.Z := -Tmp;
   end Rotate_X;

end Geometry;
```

SPARK's analysis tool can mathematically prove that the postcondition is true.

4.1.3 Pointers Are Not Arrays

The greatest source of vulnerabilities regarding pointers is their use as substitutes for arrays. Although the C language has a syntax for declaring and accessing arrays, this is just a thin syntactic layer on top of pointers. Thus:

- Array access is just pointer arithmetic;
- If a function is to manipulate an array then the array's length must be separately passed as a parameter; and
- The program is susceptible to the various vulnerabilities originating from the confusion of pointers and arrays, such as buffer overflow.

Consider a function that counts the number of times a value is present in an array. In C, this could be written:

```
#include <stdio.h>

int count(int *p, int len, int v) {
   int count = 0;
   while (len--) {
      if (*p++ == v) {
         count++;
      }
   }
   return count;
}

int main() {
   int p[5] = {0, 3, 9, 3, 3};
   int c = count(p, 5, 3);
   printf("value 3 is seen %d times in p\n", c);
   return 0;
}
```

Function `count` has no control over the range of addresses accessed from pointer `p`. The critical property that the `len` parameter is a valid length for an array of integers pointed to by parameter `p` rests completely with the caller of `count`, and `count` has no way to check that this is true.

To mitigate the risks associated with pointers being used for arrays, MISRA C contains eight rules in a section on "Pointers and arrays". These rules forbid pointer arithmetic (Rule 18.4) or, if this Advisory rule is not followed, require pointer arithmetic to stay within bounds (Rule 18.1). But, even if we rewrite the loop in `count` to respect all decidable MISRA C rules, the program's correctness still depends on the caller of `count` passing a correct value of `len`:

```
#include <stdio.h>
```

(continues on next page)

(continued from previous page)

```
int count(int *p, int len, int v) {
   int count = 0;
   for (int i = 0; i < len; i++) {
      if (p[i] == v) {
         count++;
      }
   }
   return count;
}

int main() {
   int p[5] = {0, 3, 9, 3, 3};
   int c = count(p, 5, 3);
   printf("value 3 is seen %d times in p\n", c);
   return 0;
}
```

The resulting code is more readable, but still vulnerable to incorrect values of parameter `len` passed by the caller of `count`, which violates undecidable MISRA C Rules 18.1 (pointer arithmetic should stay within bounds) and 1.3 (no undefined behavior). Contrast this with the same function in SPARK (and Ada):

```
package Types is
   type Int_Array is array (Positive range <>) of Integer;
end Types;
```

```
with Types; use Types;

function Count (P : Int_Array; V : Integer) return Natural is
   Count : Natural := 0;
begin
   for I in P'Range loop
      if P (I) = V then
         Count := Count + 1;
      end if;
   end loop;
   return Count;
end Count;
```

```
with Ada.Text_IO; use Ada.Text_IO;
with Types; use Types;
with Count;

procedure Test_Count is
   P : constant Int_Array := (0, 3, 9, 3, 3);
   C : constant Integer := Count (P, 3);
begin
   Put_Line ("value 3 is seen" & C'Img & " times in p");
end Test_Count;
```

The array parameter `P` is not simply a homogeneous sequence of Integer values. The compiler must represent `P` so that its lower and upper bounds (`P'First` and `P'Last`) and thus also its length (`P'Length`) can be retrieved. Function `Count` can simply loop over the range of valid array indexes `P'First .. P'Last` (or `P'Range` for short). As a result, function `Count` can be verified in isolation to be free of vulnerabilities such as buffer overflow, as it does not depend on the values of parameters passed by its callers. In fact, we can go further in SPARK and show that the value returned by `Count` is no greater than the length of parameter `P` by stating this property in the postcondition of `Count` and asking the SPARK analysis tool to prove it:

```
package Types is
   type Int_Array is array (Positive range <>) of Integer;
end Types;
```

```
with Types; use Types;

function Count (P : Int_Array; V : Integer) return Natural with
  Post => Count'Result <= P'Length
is
   Count : Natural := 0;
begin
   for I in P'Range loop
      pragma Loop_Invariant (Count <= I - P'First);
      if P (I) = V then
         Count := Count + 1;
      end if;
   end loop;
   return Count;
end Count;
```

The only help that SPARK analysis required from the programmer, in order to prove the postcondition, is a loop invariant (a special kind of assertion) that reflects the value of `Count` at each iteration.

4.1.4 Pointers Should Be Typed

The C language defines a special pointer type `void*` that corresponds to an untyped pointer. It is legal to convert any pointer type to and from `void*`, which makes it a convenient way to simulate C++ style templates. Consider the following code which indirectly applies `assign_int` to integer `i` and `assign_float` to floating-point `f` by calling `assign` on both:

```
#include <stdio.h>

void assign_int (int *p) {
   *p = 42;
}

void assign_float (float *p) {
   *p = 42.0;
}

typedef void (*assign_fun)(void *p);

void assign(assign_fun fun, void *p) {
   fun(p);
}

int main() {
   int i;
   float f;
   assign((assign_fun)&assign_int, &i);
   assign((assign_fun)&assign_float, &f);
   printf("i = %d; f = %f\n", i, f);
}
```

The references to the variables `i` and `f` are implicitly converted to the `void*` type as a way to apply `assign` to any second parameter `p` whose type matches the argument type of its first argument `fun`. The use of an untyped argument means that the responsibility for the correct typing rests completely with the programmer. Swap `i` and `f` in the calls to `assign` and you still get a compilable program without warnings, that runs and produces completely bogus output:

```
i = 1109917696; f = 0.000000
```

instead of the expected:

```
i = 42; f = 42.000000
```

Generics in SPARK (and Ada) can implement the desired functionality in a fully typed way, with any errors caught at compile time, where procedure `Assign` applies its parameter procedure `Initialize` to its parameter V:

```
generic
   type T is private;
   with procedure Initialize (V : out T);
procedure Assign (V : out T);
```

```
procedure Assign (V : out T) is
begin
   Initialize (V);
end Assign;
```

```
with Ada.Text_IO; use Ada.Text_IO;
with Assign;

procedure Apply_Assign is
   procedure Assign_Int (V : out Integer) is
   begin
      V := 42;
   end Assign_Int;

   procedure Assign_Float (V : out Float) is
   begin
      V := 42.0;
   end Assign_Float;

   procedure Assign_I is new Assign (Integer, Assign_Int);
   procedure Assign_F is new Assign (Float, Assign_Float);

   I : Integer;
   F : Float;
begin
   Assign_I (I);
   Assign_F (F);
   Put_Line ("I =" & I'Img & "; F =" & F'Img);
end Apply_Assign;
```

The generic procedure `Assign` must be instantiated with a specific type for T and a specific procedure (taking a single `out` parameter of this type) for `Initialize`. The procedure resulting from the instantiation applies to a variable of this type. So switching `I` and F above would result in an error detected by the compiler. Likewise, an instantiation such as the following would also be a compile-time error:

```
procedure Assign_I is new Assign (Integer, Assign_Float);
```

4.2 Enforcing Strong Typing for Scalars

In C, all scalar types can be converted both implicitly and explicitly to any other scalar type. The semantics is defined by rules of *promotion* and *conversion*, which can confuse even experts. One

example was noted earlier, in the *Preface* (page 3). Another example appears in an article introducing a safe library for manipulating scalars[8] by Microsoft expert David LeBlanc. In its conclusion, the author acknowledges the inherent difficulty in understanding scalar type conversions in C, by showing an early buggy version of the code to produce the minimum signed integer:

```
return (T)(1 << (BitCount()-1));
```

The issue here is that the literal 1 on the left-hand side of the shift is an `int`, so on a 64-bit machine with 32-bit `int` and 64-bit type T, the above is shifting 32-bit value 1 by 63 bits. This is a case of undefined behavior, producing an unexpected output with the Microsoft compiler. The correction is to convert the first literal 1 to T before the shift:

```
return (T)((T)1 << (BitCount()-1));
```

Although he'd asked some expert programmers to review the code, no one found this problem.

To avoid these issues as much as possible, MISRA C defines its own type system on top of C types, in the section on "The essential type model" (eight rules). These can be seen as additional typing rules, since all rules in this section are decidable, and can be enforced at the level of a single translation unit. These rules forbid in particular the confusing cases mentioned above. They can be divided into three sets of rules:

- restricting operations on types
- restricting explicit conversions
- restricting implicit conversions

4.2.1 Restricting Operations on Types

Apart from the application of some operations to floating-point arguments (the bitwise, mod and array access operations) which are invalid and reported by the compiler, all operations apply to all scalar types in C. MISRA C Rule 10.1 constrains the types on which each operation is possible as follows.

4.2.1.1 Arithmetic Operations on Arithmetic Types

Adding two Boolean values, or an Apple and an Orange, might sound like a bad idea, but it is easily done in C:

```
#include <stdbool.h>
#include <stdio.h>

int main() {
   bool b1 = true;
   bool b2 = false;
   bool b3 = b1 + b2;

   typedef enum {Apple, Orange} fruit;
   fruit f1 = Apple;
   fruit f2 = Orange;
   fruit f3 = f1 + f2;

   printf("b3 = %d; f3 = %d\n", b3, f3);

   return 0;
}
```

[8] https://msdn.microsoft.com/en-us/library/ms972705.aspx

No error from the compiler here. In fact, there is no undefined behavior in the above code. Variables `b3` and `f3` both end up with value 1. Of course it makes no sense to add Boolean or enumerated values, and thus MISRA C Rule 18.1 forbids the use of all arithmetic operations on Boolean and enumerated values, while also forbidding most arithmetic operations on characters. That leaves the use of arithmetic operations for signed or unsigned integers as well as floating-point types and the use of modulo operation `%` for signed or unsigned integers.

Here's an attempt to simulate the above C code in SPARK (and Ada):

```
package Bad_Arith is

   B1 : constant Boolean := True;
   B2 : constant Boolean := False;
   B3 : constant Boolean := B1 + B2;

   type Fruit is (Apple, Orange);
   F1 : constant Fruit := Apple;
   F2 : constant Fruit := Orange;
   F3 : constant Fruit := F1 + F2;

end Bad_Arith;
```

Here is the output from AdaCore's GNAT compiler:

```
 1.     package Bad_Arith is
 2.
 3.        B1 : constant Boolean := True;
 4.        B2 : constant Boolean := False;
 5.        B3 : constant Boolean := B1 + B2;
                                       |
    >>> there is no applicable operator "+" for type "Standard.Boolean"

 6.
 7.        type Fruit is (Apple, Orange);
 8.        F1 : constant Fruit := Apple;
 9.        F2 : constant Fruit := Orange;
10.        F3 : constant Fruit := F1 + F2;
                                      |
    >>> there is no applicable operator "+" for type "Fruit" defined at line 7

11.
12.     end Bad_Arith;
```

It is possible, however, to get the predecessor of a Boolean or enumerated value with `Value'Pred` and its successor with `Value'Succ`, as well as to iterate over all values of the type:

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Ok_Arith is

   B1 : constant Boolean := False;
   B2 : constant Boolean := Boolean'Succ (B1);
   B3 : constant Boolean := Boolean'Pred (B2);

   type Fruit is (Apple, Orange);
   F1 : constant Fruit := Apple;
   F2 : constant Fruit := Fruit'Succ (F1);
   F3 : constant Fruit := Fruit'Pred (F2);

begin
   pragma Assert (B1 = B3);
   pragma Assert (F1 = F3);
```

(continues on next page)

(continued from previous page)

```
   for B in Boolean loop
      Put_Line (B'Img);
   end loop;

   for F in Fruit loop
      Put_Line (F'Img);
   end loop;
end Ok_Arith;
```

4.2.1.2 Boolean Operations on Boolean

"Two bee or not two bee? Let's C":

```
#include <stdbool.h>
#include <stdio.h>

int main() {
   typedef enum {Ape, Bee, Cat} Animal;
   bool answer = (2 * Bee) || ! (2 * Bee);
   printf("two bee or not two bee? %d\n", answer);
   return 0;
}
```

The answer to the question posed by Shakespeare's Hamlet is 1, since it reduces to `A or not A` and this is true in classical logic.

As previously noted, MISRA C forbids the use of the multiplication operator with an operand of an enumerated type. Rule 18.1 also forbids the use of Boolean operations "and", "or", and "not" (&&, ||, !, respectively, in C) on anything other than Boolean operands. It would thus prohibit the Shakespearian code above.

Below is an attempt to express the same code in SPARK (and Ada), where the Boolean operators are `and`, `or`, and `not`. The `and` and `or` operators evaluate both operands, and the language also supplies short-circuit forms that evaluate the left operand and only evaluate the right operand when its value may affect the result.

```
package Bad_Hamlet is
   type Animal is (Ape, Bee, Cat);
   Answer : Boolean := 2 * Bee or not 2 * Bee; -- Illegal
end Bad_Hamlet;
```

As expected, the compiler rejects this code. There is no available * operation that works on an enumeration type, and likewise no available `or` or `not` operation.

4.2.1.3 Bitwise Operations on Unsigned Integers

Here's a genetic engineering example that combines a Bee with a Dog to produce a Cat, by manipulating the atomic structure (the bits in its representation):

```
#include <stdbool.h>
#include <assert.h>

int main() {
   typedef enum {Ape, Bee, Cat, Dog} Animal;
   Animal mutant = Bee ^ Dog;
   assert (mutant == Cat);
```

(continues on next page)

(continued from previous page)

```
   return 0;
}
```

This algorithm works by accessing the underlying bitwise representation of Bee and Dog (0x01 and 0x03, respectively) and, by applying the exclusive-or operator ^, transforming it into the underlying bitwise representation of a Cat (0x02). While powerful, manipulating the bits in the representation of values is best reserved for unsigned integers as illustrated in the book Hacker's Delight[9]. MISRA C Rule 18.1 thus forbids the use of all bitwise operations on anything but unsigned integers.

Below is an attempt to do the same in SPARK (and Ada). The bitwise operators are `and`, `or`, `xor`, and `not`, and the related bitwise functions are `Shift_Left`, `Shift_Right`, `Shift_Right_Arithmetic`, `Rotate_Left` and `Rotate_Right`:

```
package Bad_Genetics is
   type Animal is (Ape, Bee, Cat, Dog);
   Mutant : Animal := Bee xor Dog;  --  ERROR
   pragma Assert (Mutant = Cat);
end Bad_Genetics;
```

The declaration of `Mutant` is illegal, since the `xor` operator is only available for Boolean and unsigned integer (modular) values; it is not available for `Animal`. The same restriction applies to the other bitwise operators listed above. If we really wanted to achieve the effect of the above code in legal SPARK (or Ada), then the following approach will work (the type `Unsigned_8` is an 8-bit modular type declared in the predefined package `Interfaces`).

```
with Interfaces; use Interfaces;
package Unethical_Genetics is
   type Animal is (Ape, Bee, Cat, Dog);
   A      : constant array (Animal) of Unsigned_8 :=
              (Animal'Pos (Ape), Animal'Pos (Bee),
               Animal'Pos (Cat), Animal'Pos (Dog));
   Mutant : Animal := Animal'Val (A (Bee) xor A (Dog));
   pragma Assert (Mutant = Cat);
end Unethical_Genetics;
```

Note that `and`, `or`, `not` and `xor` are used both as logical operators and as bitwise operators, but there is no possible confusion between these two uses. Indeed the use of such operators on values from modular types is a natural generalization of their uses on Boolean, since values from modular types are often interpreted as arrays of Booleans.

4.2.2 Restricting Explicit Conversions

A simple way to bypass the restrictions of Rule 10.1 is to explicitly convert the arguments of an operation to a type that the rule allows. While it can often be useful to cast a value from one type to another, many casts that are allowed in C are either downright errors or poor replacements for clearer syntax.

One example is to cast from a scalar type to Boolean. A better way to express `(bool)x` is to compare x to the zero value of its type: `x != 0` for integers, `x != 0.0` for floats, `x != '\0'` for characters, `x != Enum` where `Enum` is the first enumerated value of the type. Thus, MISRA C Rule 10.5 advises avoiding casting non-Boolean values to Boolean.

Rule 10.5 also advises avoiding other casts that are, at best, obscure:

- from a Boolean to any other scalar type
- from a floating-point value to an enumeration or a character
- from any scalar type to an enumeration

[9] http://www.hackersdelight.org/

The rules are not symmetric, so although a float should not be cast to an enum, casting an enum to a float is allowed. Similarly, although it is advised to not cast a character to an enum, casting an enum to a character is allowed.

The rules in SPARK are simpler. There are no conversions between numeric types (integers, fixed-point and floating-point) and non-numeric types (such as Boolean, Character, and other enumeration types). Conversions between different non-numeric types are limited to those that make semantic sense, for example between a derived type and its parent type. Any numeric type can be converted to any other numeric type, with precise rules for rounding/truncating values when needed and run-time checking that the converted value is in the range associated with the target type.

4.2.3 Restricting Implicit Conversions

Rules 10.1 and 10.5 restrict operations on types and explicit conversions. That's not enough to avoid problematic C programs; a program violating one of these rules can be expressed using only implicit type conversions. For example, the Shakespearian code in section *Boolean Operations on Boolean* (page 27) can be reformulated to satisfy both Rules 10.1 and 10.5:

```
#include <stdbool.h>
#include <stdio.h>

int main() {
   typedef enum {Ape, Bee, Cat} Animal;
   int b = Bee;
   bool t = 2 * b;
   bool answer = t || ! t;
   printf("two bee or not two bee? %d\n", answer);
   return 0;
}
```

Here, we're implicitly converting the enumerated value Bee to an int, and then implicitly converting the integer value 2 * b to a Boolean. This does not violate 10.1 or 10.5, but it is prohibited by MISRA C Rule 10.3: *"The value of an expression shall not be assigned to an object with a narrower essential type or of a different essential type category"*.

Rule 10.1 also does not prevent arguments of an operation from being inconsistent, for example comparing a floating-point value and an enumerated value. But MISRA C Rule 10.4 handles this situation: *"Both operands of an operator in which the usual arithmetic conversions are performed shall have the same essential type category"*.

In addition, three rules in the "Composite operators and expressions" section avoid common mistakes related to the combination of explicit/implicit conversions and operations.

The rules in SPARK (and Ada) are far simpler: there are no implicit conversions! This applies both between types of a different *essential type category* as MISRA C puts it, as well as between types that are structurally the same but declared as different types.

```
procedure Bad_Conversions is
   pragma Warnings (Off);
   F : Float := 0.0;
   I : Integer := 0;
   type Animal is (Ape, Bee, Cat);
   type My_Animal is new Animal; -- derived type
   A : Animal := Cat;
   M : My_Animal := Bee;
   B : Boolean := True;
   C : Character := 'a';
begin
   F := I;  --  ERROR
```

(continues on next page)

(continued from previous page)

```
   I := A;  --  ERROR
   A := B;  --  ERROR
   M := A;  --  ERROR
   B := C;  --  ERROR
   C := F;  --  ERROR
end Bad_Conversions;
```

The compiler reports a mismatch on every statement in the above procedure (the declarations are all legal).

Adding explicit conversions makes the assignments to `F` and `M` valid, since SPARK (and Ada) allow conversions between numeric types and between a derived type and its parent type, but all other conversions are illegal:

```
procedure Bad_Conversions is
   pragma Warnings (Off);
   F : Float := 0.0;
   I : Integer := 0;
   type Animal is (Ape, Bee, Cat);
   type My_Animal is new Animal; -- derived type
   A : Animal := Cat;
   M : My_Animal := Bee;
   B : Boolean := True;
   C : Character := 'a';
begin
   F := Float (I);       --  OK
   I := Integer (A);     --  ERROR
   A := Animal (B);      --  ERROR
   M := My_Animal (A);   --  OK
   B := Boolean (C);     --  ERROR
   C := Character (F);   --  ERROR
end Bad_Conversions;
```

Although an enumeration value cannot be converted to an integer (or *vice versa*) either implicitly or explicitly, SPARK (and Ada) provide functions to obtain the effect of a type conversion. For any enumeration type `T`, the function `T'Pos(e)` takes an enumeration value from type `T` and returns its relative position as an integer, starting at `0`. For example, `Animal'Pos(Bee)` is `1`, and `Boolean'Pos(False)` is `0`. In the other direction, `T'Val(n)`, where `n` is an integer, returns the enumeration value in type `T` at relative position `n`. If `n` is negative or greater then `T'Pos(T'Last)` then a run-time exception is raised.

Hence, the following is valid SPARK (and Ada) code; `Character` is defined as an enumeration type:

```
procedure Ok_Conversions is
   pragma Warnings (Off);
   F : Float := 0.0;
   I : Integer := 0;
   type Animal is (Ape, Bee, Cat);
   type My_Animal is new Animal;
   A : Animal := Cat;
   M : My_Animal := Bee;
   B : Boolean := True;
   C : Character := 'a';
begin
   F := Float (I);
   I := Animal'Pos (A);
   I := My_Animal'Pos (M);
   I := Boolean'Pos (B);
   I := Character'Pos (C);
   I := Integer (F);
   A := Animal'Val(2);
```

(continues on next page)

(continued from previous page)

```
end Ok_Conversions;
```

CHAPTER

FIVE

INITIALIZING DATA BEFORE USE

As with most programming languages, C does not require that variables be initialized at their declaration, which makes it possible to unintentionally read uninitialized data. This is a case of undefined behavior, which can sometimes be used to attack the program.

5.1 Detecting Reads of Uninitialized Data

MISRA C attempts to prevent reads of uninitialized data in a specific section on "Initialization", containing five rules. The most important is Rule 9.1: "*The value of an object with automatic storage duration shall not be read before it has been set*". The first example in the rule is interesting, as it shows a non-trivial (and common) case of conditional initialization, where a function `f` initializes an output parameter `p` only in some cases, and the caller `g` of `f` ends up reading the value of the variable `u` passed in argument to `f` in cases where it has not been initialized:

```
static void f ( bool_t b, uint16_t *p )
{
  if ( b )
  {
     *p = 3U;
  }
}

static void g (void)
{
   uint16_t u;

   f ( false, &u );

   if ( u == 3U )
   {
      /* Non-compliant use - "u" has not been assigned a value. */
   }
}
```

Detecting the violation of Rule 9.1 can be arbitrarily complex, as the program points corresponding to a variable's initialization and read can be separated by many calls and conditions. This is one of the undecidable rules, for which most MISRA C checkers won't detect all violations.

In SPARK, the guarantee that all reads are to initialized data is enforced by the SPARK analysis tool, GNATprove, through what is referred to as *flow analysis*. Every subprogram is analyzed separately to check that it cannot read uninitialized data. To make this modular analysis possible, SPARK programs need to respect the following constraints:

- all inputs of a subprogram should be initialized on subprogram entry
- all outputs of a subprogram should be initialized on subprogram return

Hence, given the following code translated from C, GNATprove reports that function F might not always initialize output parameter P:

```
with Interfaces; use Interfaces;

package Init is
   procedure F (B : Boolean; P : out Unsigned_16);
   procedure G;
end Init;
```

```
package body Init is

   procedure F (B : Boolean; P : out Unsigned_16) is
   begin
      if B then
         P := 3;
      end if;
   end F;

   procedure G is
      U : Unsigned_16;
   begin
      F (False, U);

      if U = 3 then
         null;
      end if;
   end G;

end Init;
```

We can correct the program by initializing P to value 0 when condition B is not satisfied:

```
with Interfaces; use Interfaces;

package Init is
   procedure F (B : Boolean; P : out Unsigned_16);
   procedure G;
end Init;
```

```
package body Init is

   procedure F (B : Boolean; P : out Unsigned_16) is
   begin
      if B then
         P := 3;
      else
         P := 0;
      end if;
   end F;

   procedure G is
      U : Unsigned_16;
   begin
      F (False, U);

      if U = 3 then
         null;
      end if;
   end G;

end Init;
```

GNATprove now does not report any possible reads of uninitialized data. On the contrary, it confirms that all reads are made from initialized data.

In contrast with C, SPARK does not guarantee that global data (called *library-level* data in SPARK and Ada) is zero-initialized at program startup. Instead, GNATprove checks that all global data is explicitly initialized (at declaration or elsewhere) before it is read. Hence it goes beyond the MISRA C Rule 9.1, which considers global data as always initialized even if the default value of all-zeros might not be valid data for the application. Here's a variation of the above code where variable U is now global:

```
with Interfaces; use Interfaces;

package Init is
   U : Unsigned_16;
   procedure F (B : Boolean);
   procedure G;
end Init;
```

```
package body Init is

   procedure F (B : Boolean) is
   begin
      if B then
         U := 3;
      end if;
   end F;

   procedure G is
   begin
      F (False);

      if U = 3 then
         null;
      end if;
   end G;

end Init;
```

```
with Init;

procedure Call_Init is
begin
   Init.G;
end Call_Init;
```

GNATprove reports here that variable U might not be initialized at program startup, which is indeed the case here. It reports this issue on the main program `Call_Init` because its analysis showed that F needs to take U as an initialized input (since F is not initializing U on all paths, U keeps its value on the other path, which needs to be an initialized value), which means that G which calls F also needs to take U as an initialized input, which in turn means that `Call_Init` which calls G also needs to take U as an initialized input. At this point, we've reached the main program, so the initialization phase (referred to as *elaboration* in SPARK and Ada) should have taken care of initializing U. This is not the case here, hence the message from GNATprove.

It is possible in SPARK to specify that G should initialize variable U; this is done with a *data dependency* contract introduced with aspect `Global` following the declaration of procedure G:

```
with Interfaces; use Interfaces;

package Init is
   U : Unsigned_16;
```

(continues on next page)

(continued from previous page)

```
   procedure F (B : Boolean);
   procedure G with Global => (Output => U);
end Init;
```

```
package body Init is

   procedure F (B : Boolean) is
   begin
      if B then
         U := 3;
      end if;
   end F;

   procedure G is
   begin
      F (False);

      if U = 3 then
         null;
      end if;
   end G;

end Init;
```

```
with Init;

procedure Call_Init is
begin
   Init.G;
end Call_Init;
```

GNATprove reports the error on the call to F in G, as it knows at this point that F needs U to be initialized but the calling context in G cannot provide that guarantee. If we provide the same data dependency contract for F, then GNATprove reports the error on F itself, similarly to what we saw for an output parameter U.

5.2 Detecting Partial or Redundant Initialization of Arrays and Structures

The other rules in the section on "Initialization" deal with common errors in initializing aggregates and *designated initializers* in C99 to initialize a structure or array at declaration. These rules attempt to patch holes created by the lax syntax and rules in C standard. For example, here are five valid initializations of an array of 10 elements in C:

```
int main() {
   int a[10] = {0};
   int b[10] = {0, 0};
   int c[10] = {0, [8] = 0};
   int d[10] = {0, [8] = 0, 0};
   int e[10] = {0, [8] = 0, 0, [8] = 1};
   return 0;
}
```

Only a is fully initialized to all-zeros in the above code snippet. MISRA C Rule 9.3 thus forbids all other declarations by stating that *"Arrays shall not be partially initialized"*. In addition, MISRA C Rule 9.4 forbids the declaration of e by stating that *"An element of an object shall not be initialised more than once"* (in e's declaration, the element at index 8 is initialized twice).

The same holds for initialization of structures. Here is an equivalent set of declarations with the same potential issues:

```
int main() {
   typedef struct { int x; int y; int z; } rec;
   rec a = {0};
   rec b = {0, 0};
   rec c = {0, .y = 0};
   rec d = {0, .y = 0, 0};
   rec e = {0, .y = 0, 0, .y = 1};
   return 0;
}
```

Here only a, d and e are fully initialized. MISRA C Rule 9.3 thus forbids the declarations of b and c. In addition, MISRA C Rule 9.4 forbids the declaration of e.

In SPARK and Ada, the aggregate used to initialize an array or a record must fully cover the components of the array or record. Violations lead to compilation errors, both for records:

```
package Init_Record is
   type Rec is record
      X, Y, Z : Integer;
   end record;
   R : Rec := (X => 1);  --  ERROR, Y and Z not specified
end Init_Record;
```

and for arrays:

```
package Init_Array is
   type Arr is array (1 .. 10) of Integer;
   A : Arr := (1 => 1);  --  ERROR, elements 2..10 not specified
end Init_Array;
```

Similarly, redundant initialization leads to compilation errors for records:

```
package Init_Record is
   type Rec is record
      X, Y, Z : Integer;
   end record;
   R : Rec := (X => 1, Y => 1, Z => 1, X => 2);  --  ERROR, X duplicated
end Init_Record;
```

and for arrays:

```
package Init_Array is
   type Arr is array (1 .. 10) of Integer;
   A : Arr := (1 .. 8 => 1, 9 .. 10 => 2, 7 => 3);  --  ERROR, A(7) duplicated
end Init_Array;
```

Finally, while it is legal in Ada to leave uninitialized parts in a record or array aggregate by using the box notation (meaning that the default initialization of the type is used, which may be no initialization at all), SPARK analysis rejects such use when it leads to components not being initialized, both for records:

```
package Init_Record is
   type Rec is record
      X, Y, Z : Integer;
   end record;
   R : Rec := (X => 1, others => <>);  --  ERROR, Y and Z not specified
end Init_Record;
```

and for arrays:

```
package Init_Array is
   type Arr is array (1 .. 10) of Integer;
   A : Arr := (1 .. 8 => 1, 9 .. 10 => <>);  --  ERROR, A(9..10) not specified
end Init_Array;
```

CHAPTER

SIX

CONTROLLING SIDE EFFECTS

As with most programming languages, C allows side effects in expressions. This leads to subtle issues about conflicting side effects, when subexpressions of the same expression read/write the same variable.

6.1 Preventing Undefined Behavior

Conflicting side effects are a kind of undefined behavior; the C Standard (section 6.5) defines the concept as follows:

> *"Between two sequence points, an object is modified more than once, or is modified and the prior value is read other than to determine the value to be stored"*

This legalistic wording is somewhat opaque, but the notion of sequence points is summarized in Annex C of the C90 and C99 standards. MISRA C repeats these conditions in the Amplification of Rule 13.2, including the read of a volatile variable as a side effect similar to writing a variable.

This rule is undecidable, so MISRA C completes it with two rules that provide simpler restrictions preventing some side effects in expressions, thus reducing the potential for undefined behavior:

- Rule 13.3: *"A full expression containing an increment (++) or decrement (–) operator should have no other potential side effects other than that caused by the increment or decrement operator"*.
- Rule 13.4: *"The result of an assignment operator should not be used"*.

In practice, conflicting side effects usually manifest themselves as portability issues, since the result of the evaluation of an expression depends on the order in which a compiler decides to evaluate its subexpressions. So changing the compiler version or the target platform might lead to a different behavior of the application.

To reduce the dependency on evaluation order, MISRA C Rule 13.1 states: *"Initializer lists shall not contain persistent side effects"*. This case is theoretically different from the previously mentioned conflicting side effects, because initializers that comprise an initializer list are separated by sequence points, so there is no risk of undefined behavior if two initializers have conflicting side effects. But given that initializers are executed in an unspecified order, the result of a conflict is potentially as damaging for the application.

6.2 Reducing Programmer Confusion

Even in cases with no undefined or unspecified behavior, expressions with multiple side effects can be confusing to programmers reading or maintaining the code. This problem arises in particular with C's increment and decrement operators that can be applied prior to or after the expression evaluation, and with the assignment operator = in C since it can easily be mistaken for equality. Thus MISRA C forbids the use of the increment / decrement (Rule 13.3) and assignment (Rule 13.4) operators in expressions that have other potential side effects.

In other cases, the presence of expressions with side effects might be confusing, if the programmer wrongly thinks that the side effects are guaranteed to occur. Consider the function `decrease_until_one_is_null` below, which decreases both arguments until one is null:

```
#include <stdio.h>

void decrease_until_one_is_null (int *x, int *y) {
   if (x == 0 || y == 0) {
      return;
   }
   while (--*x != 0 && --*y != 0) {
      // nothing
   }
}

int main() {
   int x = 42, y = 42;
   decrease_until_one_is_null (&x, &y);
   printf("x = %d, y = %d\n", x, y);
   return 0;
}
```

The program produces the following output:

```
x = 0, y = 1
```

I.e., starting from the same value 42 for both x and y, only x has reached the value zero after `decrease_until_one_is_null` returns. The reason is that the side effect on y is performed only conditionally. To avoid such surprises, MISRA C Rule 13.5 states: *"The right hand operand of a logical && or || operator shall not contain persistent side effects"*; this rule forbids the code above.

MISRA C Rule 13.6 similarly states: *"The operand of the sizeof operator shall not contain any expression which has potential side effects"*. Indeed, the operand of `sizeof` is evaluated only in rare situations, and only according to C99 rules, which makes any side effect in such an operand a likely mistake.

6.3 Side Effects and SPARK

In SPARK, expressions cannot have side effects; only statements can. In particular, there are no increment/decrement operators, and no assignment operator. There is instead an assignment statement, whose syntax using `:=` clearly distinguishes it from equality (using =). And in any event an expression is not allowed as a statement and this a construct such as `X = Y;` would be illegal. Here is how a variable X can be assigned, incremented and decremented:

```
X := 1;
X := X + 1;
X := X - 1;
```

There are two possible side effects when evaluating an expression:

- a read of a volatile variable
- a side effect occurring inside a function that the expression calls

Reads of volatile variables in SPARK are restricted to appear immediately at statement level, so the following is not allowed:

```
package Volatile_Read is
   X : Integer with Volatile;
   procedure P (Y : out Integer);
end Volatile_Read;
```

```
package body Volatile_Read is
   procedure P (Y : out Integer) is
   begin
      Y := X - X;  --  ERROR
   end P;
end Volatile_Read;
```

Instead, every read of a volatile variable must occur immediately before being assigned to another variable, as follows:

```
package Volatile_Read is
   X : Integer with Volatile;
   procedure P (Y : out Integer);
end Volatile_Read;
```

```
package body Volatile_Read is
   procedure P (Y : out Integer) is
      X1 : constant Integer := X;
      X2 : constant Integer := X;
   begin
      Y := X1 - X2;
   end P;
end Volatile_Read;
```

Note here that the order of capture of the volatile value of X might be significant. For example, X might denote a quantity which only increases, like clock time, so that the above expression X1 - X2 would always be negative or zero.

Even more significantly, functions in SPARK cannot have side effects; only procedures can. The only effect of a SPARK function is the computation of a result from its inputs, which may be passed as parameters or as global variables. In particular, SPARK functions cannot have `out` or `in out` parameters:

```
function Bad_Function (X, Y : Integer; Sum, Max : out Integer) return Boolean;
--  ERROR, since "out" parameters are not allowed
```

More generally, SPARK does not allow functions that have a side effect in addition to returning their result, as is typical of many idioms in other languages, for example when setting a new value and returning the previous one:

```
package Bad_Functions is
   function Set (V : Integer) return Integer;
   function Get return Integer;
end Bad_Functions;
```

```
package body Bad_Functions is

   Value : Integer := 0;

   function Set (V : Integer) return Integer is
      Previous : constant Integer := Value;
   begin
      Value := V;  --  ERROR
      return Previous;
   end Set;

   function Get return Integer is (Value);

end Bad_Functions;
```

GNATprove detects that function `Set` has a side effect on global variable `Value` and issues an error.

The correct idiom in SPARK for such a case is to use a procedure with an `out` parameter to return the desired result:

```
package Ok_Subprograms is
   procedure Set (V : Integer; Prev : out Integer);
   function Get return Integer;
end Ok_Subprograms;
```

```
package body Ok_Subprograms is

   Value : Integer := 0;

   procedure Set (V : Integer; Prev : out Integer) is
   begin
      Prev := Value;
      Value := V;
   end Set;

   function Get return Integer is (Value);

end Ok_Subprograms;
```

With the above restrictions in SPARK, none of the conflicts of side effects that can occur in C can occur in SPARK, and this is guaranteed by flow analysis.

CHAPTER

SEVEN

DETECTING UNDEFINED BEHAVIOR

Undefined behavior (and critical unspecified behavior, which we'll treat as undefined behavior) are the plague of C programs. Many rules in MISRA C are designed to avoid undefined behavior, as evidenced by the twenty occurrences of "undefined" in the MISRA C:2012 document.

MISRA C Rule 1.3 is the overarching rule, stating very simply:

> *"There shall be no occurrence of undefined or critical unspecified behaviour."*

The deceptive simplicity of this rule rests on the definition of *undefined or critical unspecified behaviour*. Appendix H of MISRA:C 2012 lists hundreds of cases of undefined and critical unspecified behavior in the C programming language standard, a majority of which are not individually decidable.

It is therefore not surprising that a majority of MISRA C checkers do not make a serious attempt to verify compliance with MISRA C Rule 1.3.

7.1 Preventing Undefined Behavior in SPARK

Since SPARK is a subset of the Ada programming language, SPARK programs may exhibit two types of undefined behaviors that can occur in Ada:

- *bounded error*: when the program enters a state not defined by the language semantics, but the consequences are bounded in various ways. For example, reading uninitialized data can lead to a bounded error, when the value read does not correspond to a valid value for the type of the object. In this specific case, the Ada Reference Manual states that either a predefined exception is raised or execution continues using the invalid representation.
- *erroneous execution*: when when the program enters a state not defined by the language semantics, but the consequences are not bounded by the Ada Reference Manual. This is the closest to an undefined behavior in C. For example, concurrently writing through different tasks to the same unprotected variable is a case of erroneous execution.

Many cases of undefined behavior in C would in fact raise exceptions in SPARK. For example, accessing an array beyond its bounds raises the exception `Constraint_Error` while reaching the end of a function without returning a value raises the exception `Program_Error`.

The SPARK Reference Manual defines the SPARK subset through a combination of *legality rules* (checked by the compiler, or the compiler-like phase preceding analysis) and *verification rules* (checked by the formal analysis tool GNATprove). Bounded errors and erroneous execution are prevented by a combination of legality rules and the *flow analysis* part of GNATprove, which in particular detects potential reads of uninitialized data, as described in *Detecting Reads of Uninitialized Data* (page 33). The following discussion focuses on how SPARK can verify that no exceptions can be raised.

7.2 Proof of Absence of Run-Time Errors in SPARK

The most common run-time errors are related to misuse of arithmetic (division by zero, overflows, exceeding the range of allowed values), arrays (accessing beyond an array bounds, assigning between arrays of different lengths), and structures (accessing components that are not defined for a given variant).

Arithmetic run-time errors can occur with signed integers, unsigned integers, fixed-point and floating-point (although with IEEE 754 floating-point arithmetic, errors are manifest as special run-time values such as NaN and infinities rather than as exceptions that are raised). These errors can occur when applying arithmetic operations or when converting between numeric types (if the value of the expression being converted is outside the range of the type to which it is being converted).

Operations on enumeration values can also lead to run-time errors; e.g., `T'Pred(T'First)` or `T'Succ(T'Last)` for an enumeration type `T`, or `T'Val(N)` where `N` is an integer value that is outside the range `0 .. T'Pos(T'Last)`.

The `Update` procedure below contains what appears to be a simple assignment statement, which sets the value of array element `A(I+J)` to `P/Q`.

```
package Show_Runtime_Errors is

   type Nat_Array is array (Integer range <>) of Natural;
   --  The values in subtype Natural are 0 , 1, ... Integer'Last

   procedure Update (A : in out Nat_Array; I, J, P, Q : Integer);

end Show_Runtime_Errors;
```

```
package body Show_Runtime_Errors is

   procedure Update (A : in out Nat_Array; I, J, P, Q : Integer) is
   begin
      A (I + J) := P / Q;
   end Update;

end Show_Runtime_Errors;
```

However, for an arbitrary invocation of this procedure, say `Update(A, I, J, P, Q)`, an exception can be raised in a variety of circumstances:

- The computation `I+J` may overflow, for example if `I` is `Integer'Last` and `J` is positive.

  ```
  A (Integer'Last + 1) := P / Q;
  ```

- The value of `I+J` may be outside the range of the array `A`.

  ```
  A (A'Last + 1) := P / Q;
  ```

- The division `P / Q` may overflow in the special case where `P` is `Integer'First` and `Q` is `-1`, because of the asymmetric range of signed integer types.

  ```
  A (I + J) := Integer'First / -1;
  ```

- Since the array can only contain non-negative numbers (the element subtype is `Natural`), it is also an error to store a negative value in it.

  ```
  A (I + J) := 1 / -1;
  ```

- Finally, if `Q` is 0 then a divide by zero error will occur.

```
A (I + J) := P / 0;
```

For each of these potential run-time errors, the compiler will generate checks in the executable code, raising an exception if any of the checks fail:

```
A (Integer'Last + 1) := P / Q;
--  raised CONSTRAINT_ERROR : overflow check failed

A (A'Last + 1) := P / Q;
--  raised CONSTRAINT_ERROR : index check failed

A (I + J) := Integer'First / (-1);
--  raised CONSTRAINT_ERROR : overflow check failed

A (I + J) := 1 / (-1);
--  raised CONSTRAINT_ERROR : range check failed

A (I + J) := P / 0;
--  raised CONSTRAINT_ERROR : divide by zero
```

These run-time checks incur an overhead in program size and execution time. Therefore it may be appropriate to remove them if we are confident that they are not needed.

The traditional way to obtain the needed confidence is through testing, but it is well known that this can never be complete, at least for non-trivial programs. Much better is to guarantee the absence of run-time errors through sound static analysis, and that's where SPARK and GNATprove can help.

More precisely, GNATprove logically interprets the meaning of every instruction in the program, taking into account both control flow and data/information dependencies. It uses this analysis to generate a logical formula called a *verification condition* for each possible check.

```
A (Integer'Last + 1) := P / Q;
--  medium: overflow check might fail

A (A'Last + 1) := P / Q;
--  medium: array index check might fail

A (I + J) := Integer'First / (-1);
--  medium: overflow check might fail

A (I + J) := 1 / (-1);
--  medium: range check might fail

A (I + J) := P / 0;
--  medium: divide by zero might fail
```

The verification conditions are then given to an automatic prover. If every verification condition can be proved, then no run-time errors will occur.

GNATprove's analysis is sound — it will detect all possible instances of run-time exceptions being raised — while also having high precision (i.e., not producing a cascade of "false alarms").

The way to program in SPARK so that GNATprove can guarantee the absence of run-time errors entails:

- declaring variables with precise constraints, and in particular to specify precise ranges for scalars; and
- defining preconditions and postconditions on subprograms, to specify respectively the constraints that callers should respect and the guarantees that the subprogram should provide on exit.

For example, here is a revised version of the previous example, which can guarantee through proof that no possible run-time error can be raised:

```
package No_Runtime_Errors is

   subtype Index_Range is Integer range 0 .. 100;

   type Nat_Array is array (Index_Range range <>) of Natural;

   procedure Update (A : in out Nat_Array; I, J : Index_Range; P, Q : Positive)
   with
     Pre => I + J in A'Range;

end No_Runtime_Errors;
```

```
package body No_Runtime_Errors is

   procedure Update (A : in out Nat_Array; I, J : Index_Range; P, Q : Positive) is
   begin
      A (I + J) := P / Q;
   end Update;

end No_Runtime_Errors;
```

CHAPTER

EIGHT

DETECTING UNREACHABLE CODE AND DEAD CODE

MISRA C defines *unreachable code* as code that cannot be executed, and it defines *dead code* as code that can be executed but has no effect on the functional behavior of the program. (These definitions differ from traditional terminology, which refers to the first category as "dead code" and the second category as "useless code".) Regardless of the terminology, however, both types are actively harmful, as they might confuse programmers and lead to errors during maintenance.

The "Unused code" section of MISRA C contains seven rules that deal with detecting both unreachable code and dead code. The two most important rules are:

- Rule 2.1: *"A project shall not contain unreachable code"*, and
- Rule 2.2: *"There shall not be dead code"*.

Other rules in the same section prohibit unused entities of various kinds (type declarations, tag declarations, macro declarations, label declarations, function parameters).

While some simple cases of unreachable code can be detected by static analysis (typically if a condition in an `if` statement can be determined to be always true or false), most cases of unreachable code can only be detected by performing coverage analysis in testing, with the caveat that code reported as not being executed is not necessarily unreachable (it could simply reflect gaps in the test suite). Note that *statement coverage*, rather than the more comprehensive *decision coverage* or *modified condition / decision coverage* (MC/DC) as defined in the DO-178C standard for airborne software, is sufficient to detect potential unreachable statements, corresponding to code that is not covered during the testing campaign.

The presence of dead code is much harder to detect, both statically and dynamically, as it requires creating a complete dependency graph linking statements in the code and their effect on visible behavior of the program.

SPARK can detect some cases of both unreachable and dead code through its precise construction of a dependency graph linking a subprogram's statements to all its inputs and outputs. This analysis might not be able to detect complex cases, but it goes well beyond what other analyses do in general.

```
procedure Much_Ado_About_Little (X, Y, Z : Integer; Success : out Boolean) is

   procedure Ok is
   begin
      Success := True;
   end Ok;

   procedure NOk is
   begin
      Success := False;
   end NOk;

begin
   Success := False;
```

(continues on next page)

(continued from previous page)

```
   for K in Y .. Z loop
      if K < X and not Success then
         Ok;
      end if;
   end loop;

   if X > Y then
      Ok;
   else
      NOk;
   end if;

   if Z > Y then
      NOk;
      return;
   else
      Ok;
      return;
   end if;

   if Success then
      Success := not Success;
   end if;
end Much_Ado_About_Little;
```

The only code in the body of `Much_Ado_About_Little` that affects the result of the procedure's execution is the `if Z > Y...` statement, since this statement sets `Success` to either True or False regardless of what the previous statements did. I.e., the statements preceding this `if` are dead code in the MISRA C sense. Since both branches of the `if Z > Y...` statement return from the procedure, the subsequent `if Success...` statement is unreachable. GNATprove detects and issues warnings about both the dead code and the unreachable code.

CHAPTER

NINE

CONCLUSION

The C programming language is ”close to the metal” and has emerged as a *lingua franca* for the majority of embedded platforms of all sizes. However, its software engineering deficiencies (such as the absence of data encapsulation) and its many traps and pitfalls present major obstacles to those developing critical applications. To some extent, it is possible to put the blame for programming errors on programmers themselves, as Linus Torvalds admonished:

> *”Learn C, instead of just stringing random characters together until it compiles (with warnings).”*

But programmers are human, and even the best would be hard pressed to be 100% correct about the myriad of semantic details such as those discussed in this document. Programming language abstractions have been invented precisely to help developers focus on the ”big picture” (thinking in terms of problem-oriented concepts) rather than low-level machine-oriented details, but C lacks these abstractions. As Kees Cook from the Kernel Self Protection Project puts it (during the Linux Security Summit North America 2018):

> *”Talking about C as a language, and how it’s really just a fancy assembler”*

Even experts sometimes have problems with the C programming language rules, as illustrated by Microsoft expert David LeBlanc (see *Enforcing Strong Typing for Scalars* (page 24)) or the MISRA C Committee itself (see the *Preface* (page 3)).

The rules in MISRA C represent an impressive collective effort to improve the reliability of C code in critical applications, with a focus on avoiding error-prone features rather than enforcing a particular programming style. The Rationale provided with each rule is a clear and unobjectionable justification of the rule’s benefit.

At a fundamental level, however, MISRA C is still built on a base language that was not really designed with the goal of supporting large high-assurance applications. As shown in this document, there are limits to what static analysis can enforce with respect to the MISRA C rules. It’s hard to retrofit reliability, safety and security into a language that did not have these as goals from the start.

The SPARK language took a different approach, starting from a base language (Ada) that was designed from the outset to support solid software engineering, and eliminating features that were implementation dependent or otherwise hard to formally analyze. In this document we have shown how the SPARK programming language and its associated formal verification tools can contribute usefully to the goal of producing error-free software, going beyond the guarantees that can be achieved in MISRA C.

CHAPTER

TEN

REFERENCES

10.1 About MISRA C

The official website of the MISRA association https://www.misra.org.uk/ has many freely available resources about MISRA C, some of which can be downloaded after registering on the MISRA Bulletin Board at https://www.misra.org.uk/forum/ (such as the examples from the MISRA C:2012 standard, which includes a one-line description of each guideline).

The following documents are freely available:

- *MISRA Compliance 2016: Achieving compliance with MISRA coding guidelines*, 2016, which explains the rationale and process for compliance, including a thorough discussions of acceptable deviations
- *MISRA C:2012 - Amendment 1: Additional security guidelines for MISRA C:2012*, 2016, which contains 14 additional guidelines focusing on security. This is a minor addition to MISRA C.

The main MISRA C:2012 document can be purchased from the MISRA webstore.

PRQA is the company that first developed MISRA C, and they have been heavily involved in every version since then. Their webpage http://www.prqa.com/coding-standards/misra/ contains many resources about MISRA C: product datasheets, white papers, webinars, professional courses.

The PRQA Resources Library at http://info.prqa.com/resources-library?filter=white_paper has some freely available white papers on MISRA C and the use of static analyzers:

- An introduction to MISRA C:2012 at http://info.prqa.com/MISRA C-2012-whitepaper-evaluation-lp
- *The Myth of Perfect MISRA Compliance* at http://info.prqa.com/myth-of-perfect-MISRA Compliance-evaluation-lp, providing background information on the use and limitations of static analyzers for checking MISRA C compliance

In 2013 ISO standardized a set of 45 rules focused on security, available in the *C Secure Coding Rules*. A draft is freely available at http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1624.pdf

In 2018 MISRA published *MISRA C:2012 - Addendum 2: Coverage of MISRA C:2012 against ISO/IEC TS 17961:2013 "C Secure"*, mapping ISO rules to MISRA C:2012 guidelines. This document is freely available from https://www.misra.org.uk/.

10.2 About SPARK

The e-learning website https://learn.adacore.com/ contains a freely available interactive course on SPARK.

The SPARK User's Guide is available at http://docs.adacore.com/spark2014-docs/html/ug/.

The SPARK Reference Manual is available at http://docs.adacore.com/spark2014-docs/html/lrm/.

A student-oriented textbook on SPARK is *Building High Integrity Applications with SPARK* by John McCormick and Peter Chapin, published by Cambridge University Press. It covers the latest version of the language, SPARK 2014.

A historical account of the evolution of SPARK technology and its use in industry is covered in the article *Are We There Yet? 20 Years of Industrial Theorem Proving with SPARK* by Roderick Chapman and Florian Schanda, at http://proteancode.com/keynote.pdf

The website https://www.adacore.com/sparkpro is a portal for up-to-date information and resources on SPARK. AdaCore blog's site https://blog.adacore.com/ contains a number of SPARK-related posts.

The booklet *AdaCore Technologies for Cyber Security* shows how AdaCore's technology can be used to prevent or mitigate the most common security vulnerabilities in software. See https://www.adacore.com/books/adacore-tech-for-cyber-security/.

The booklet *AdaCore Technologies for CENELEC EN 50128:2011* shows how AdaCore's technology can be used in conjunction with the CENELEC EN 50128:2011 software standard for railway control and protection systems. It describes in particular where the SPARK technology fits best and how it can be used to meet various requirements of the standard. See: https://www.adacore.com/books/cenelec-en-50128-2011/.

The booklet *AdaCore Technologies for DO-178C/ED-12C* similarly shows how AdaCore's technology can be used in conjunction with the DO-178C/ED-12C standard for airborne software, and describes in particular how SPARK can be used in conjunction with the Formal Methods supplement DO-333/ED-216. See https://www.adacore.com/books/do-178c-tech/.

10.3 About MISRA C and SPARK

The blog post at https://blog.adacore.com/MISRA-C-2012-vs-spark-2014-the-subset-matching-game reviews the 27 undecidable rules in MISRA C:2012 and describes how SPARK addresses them.

The white paper *A Comparison of SPARK with MISRA C and Frama-C* at https://www.adacore.com/papers/compare-spark-MISRA-C-frama-c compares SPARK to MISRA C and to the formal verification tool Frama-C for C programs.