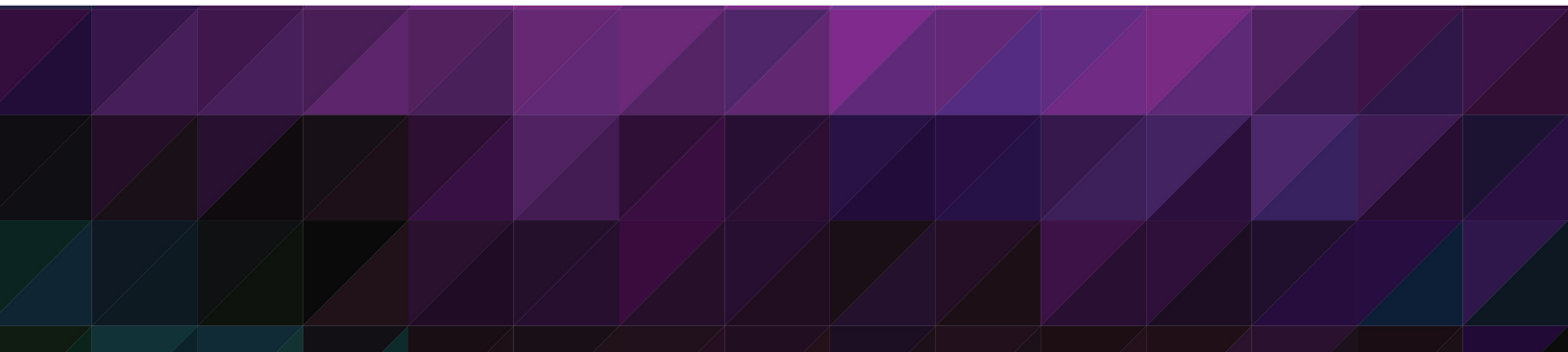
# Implementation Guidance for the Adoption of SPARK

AdaCore THALES



# Implementation Guidance for the Adoption of SPARK

***Release 1.0***

**AdaCore and Thales**

**Jan 26, 2017**

CHAPTER

ONE

# OBJECTIVES AND CONTENTS

This document was written to facilitate the adoption of SPARK. It targets team leaders and technology experts, who will find a description of the various levels of software assurance at which the technology can be used along with the associated costs and benefits. It also targets software developers (these are assumed to have some knowledge of Ada language and AdaCore technology), who will find detailed guidance of how to adopt SPARK at every assurance level.

Section *Levels of Software Assurance* presents the four assurance levels described in this document. It starts with a brief introduction of the Ada programming language and its SPARK subset and then presents the levels (Stone, Bronze, Silver and Gold) that can be achieved with the use of SPARK language and toolset, using techniques varying from merely applying the language subset up to using the most powerful analyses. The lowest levels are the simplest to adopt and already bring significant benefits. The highest levels require more effort to adopt and bring the most guarantees. This section is particularly well suited for team leaders and technology experts who want to understand how SPARK could be useful in their context.

Sections *Stone Level - Valid SPARK* to *Gold Level - Proof of Key Integrity Properties* present the details of the four levels of software assurance. Each section starts with a short description of three key aspects of adopting SPARK at that level:

- *Benefits* - What is gained from adopting SPARK?
- *Impact on Process* - How should the process be adapted to use SPARK?
- *Costs and Limitations* - What are the main costs and limitations for adopting SPARK?

The rest of each section describes how to progressively adopt SPARK at that level in an Ada project. Section *Example* shows an example of application for all four levels. These sections are particularly well suited for software developers who need to use SPARK at a given level.

Although this document is about adopting SPARK for use on existing Ada code, the same guidelines can be used for adopting SPARK from the beginning of a project. The main difference in that case is that one would not want to start at the lowest level but already take into account the final targeted level starting with the initial design phase.

This version of the document is based on the SPARK Pro 17 and GPS 17 versions. Further references are given at the end of this document.

CHAPTER

TWO

# LEVELS OF SOFTWARE ASSURANCE

## 2.1 Ada

Ada is a language for long-lived critical systems. Programming in Ada makes it easier to prevent the introduction of errors, thanks to the stronger language rules than in many comparative languages (C and C++ in particular, including their safer variants like MISRA C and MISRA C++) which make it possible for the compiler to reject erroneous programs. Programming in Ada also makes it easier to detect the presence of errors in programs, thanks to the language rules mandating run-time checking of type safety and memory safety conditions which cannot be checked at compile time so that violating these conditions during testing leads to exceptions rather than undefined behavior.

A lesser known advantage of programming in Ada is its greater number of language features for embedding program specifications inside the program, from mundane properties of data like ranges of values to rich data invariants expressed with arbitrary boolean expressions. An important addition to this list of features is the ability to provide contracts on subprograms, consisting of preconditions and postconditions. Contracts are a central part of the Ada 2012 version of the language.

Preconditions are properties that should be true when a subprogram is called. In typical software development in Ada or other languages, preconditions are either given in the program as comments accompanying subprogram declarations or as defensive code inside subprograms to detect improper calling conditions. When using the latest version of Ada, developers can express preconditions as boolean properties which should hold when a subprogram is called and the compiler can insert run-time checks to ensure that preconditions are true when the subprogram is called.

Postconditions are properties that should be true when a subprogram returns. In typical software development, postconditions are also either given in the program as comments accompanying subprogram declarations or as assertions inside subprograms to detect implementation errors, but can also be provided as defensive code to detect improper values returned at the call site. When using the latest version of Ada, developers can express postconditions as boolean properties which should be true when a subprogram returns and the compiler can insert run-time checks to ensure that postconditions are true when the subprogram returns.

The main use of preconditions and postconditions, like other language features in Ada for embedding program specifications inside the program, is to allow detecting violations of these program specifications during testing. Another increasingly important use of these language features is to facilitate the detection of errors by static analyzers, which analyze the source code of programs without actually executing them. Without such specifications in the program, static analyzers can only detect violations of language dynamic constraints (e.g. division by zero or buffer overflow). However, the presence of such specifications in the program allows static analyzers to target the verification of these higher level properties. Specifications also constrain the state space that the static analyzer has to consider during analysis, which leads to faster running time and higher precision.

## 2.2 SPARK

Static analyzers fall into two broad categories: bug finders and verifiers. Bug finders detect violations of properties. Verifiers guarantee the absence of violations of properties. Because they target opposite goals, bug finders and verifiers usually have different architectures, are based on different technologies, and require different methodologies. Typically, bug finders require little upfront work, but may generate many false alarms which need to be manually triaged and addressed, while verifiers require some upfront work, but generate fewer false alarms thanks to the use of more powerful techniques. AdaCore develops and distributes one bug finder (CodePeer) and one verifier (SPARK).

SPARK is a verifier co-developed by AdaCore and Altran and distributed by AdaCore. The main webpage for the SPARK Pro product is http://www.adacore.com/sparkpro/. SPARK analysis can give strong guarantees that a program:

- does not read uninitialized data,
- accesses global data only as intended,
- does not contain concurrency errors (deadlocks and data races),
- does not contain run-time errors (e.g. division by zero or buffer overflow),
- respects key integrity properties (e.g. interaction between components or global invariants),
- is a correct implementation of software requirements expressed as contracts.

SPARK can analyze a complete program or only parts of it, but can only be applied to parts of a program that don't explicitly use pointers (though references and addresses are allowed) and that don't catch exceptions. Pointers and exceptions are both features that make formal verification, as done by SPARK, infeasible, either because of limitations of state-of-the-art technology or because of the disproportionate effort required from users to apply formal verification in such situations. This large subset of Ada that is analyzed by SPARK is also called the SPARK language subset.

SPARK builds on the strengths of Ada to provide even more guarantees statically rather than dynamically. As summarized in the following table, Ada provides strict syntax and strong typing at compile time plus dynamic checking of run-time errors and program contracts. SPARK allows performing such checking statically. In addition, it enforces the use of a safer language subset and detects data flow errors statically.

| | Ada | SPARK |
|---|---|---|
| Contract programming | dynamic | dynamic / static |
| Run-time errors | dynamic | dynamic / static |
| Data flow errors | – | static |
| Strong typing | static | static |
| Safer language subset | – | static |
| Strict clear syntax | static | static |

The main benefit of formal program verification, as performed by SPARK (and by Frama-C or TrustInSoft Analyzer for C code) is that it allows verifying properties that are difficult or very costly to verify by other methods, such as testing or reviews. That difficulty may originate in a mix of the complexity of the software, the complexity of the requirement, and the unknown capabilities of attackers. Formal verification allows giving guarantees that some properties are always verified, however complex the context. The latest versions of international certification standards for avionics (DO-178C) and railway (CENELEC 50128:2011) have recognized these benefits by increasing the role that formal methods can play in the development of critical software.

## 2.3 Levels of SPARK Use

The scope and level of SPARK analysis depend on the objectives being pursued by the adoption of SPARK. The scope of analysis may be the totality of a project, only some units, or only parts of units. The level of analysis may range from simple guarantees provided by flow analysis to complex properties being proved. These can be divided in five easily remembered levels:

1. *Stone level* - valid SPARK
2. *Bronze level* - initialization and correct data flow
3. *Silver level* - absence of run-time errors (AoRTE)
4. *Gold level* - proof of key integrity properties
5. *Platinum level* - full functional proof of requirements

Platinum level is defined here for completeness, but is not further discussed in this document since it is not recommended during initial adoption of SPARK. Each level builds on the previous one, so that the code subject to the Gold level should be a subset of the code subject to Silver level, which itself is a subset of the code subject to Bronze level, which is in general the same as the code subject to Stone level. We advise using:

- Stone level only as an intermediate level during adoption,
- Bronze level for as large a part of the code as possible,
- Silver level as the default target for critical software (subject to costs and limitations),
- Gold level only for a subset of the code subject to specific key integrity (safety/security) properties.

Our starting point is a program in Ada, which could be thought of as the Brick level: thanks to the use of Ada programming language, this level already provides some confidence: it is the highest level in The Three Little Pigs fable! And indeed languages with weaker semantics could be thought of as Straw and Sticks levels. However, the adoption of SPARK allows us to get stronger guarantees, should the wolf in the fable adopt more aggressive means of attack than blowing.

In the following, we use "SPARK" to denote the SPARK language, and "GNATprove" to denote the formal verification tool in SPARK product.

CHAPTER

THREE

# STONE LEVEL - VALID SPARK

The goal of reaching this level is to identify as much code as possible as belonging to the SPARK subset. The user is responsible for identifying candidate SPARK code by applying the marker `SPARK_Mode` to flag SPARK code to GNATprove, which is responsible for checking that the code marked with `SPARK_Mode` is indeed valid SPARK code. Note that valid SPARK code may still be incorrect in many ways, such as raising run-time exceptions. Being valid merely means that the code respects the legality rules that define the SPARK subset in the SPARK Reference Manual (see http://docs.adacore.com/spark2014-docs/html/lrm/). The number of lines of SPARK code in a program can be computed (along with other metrics such as the total number of lines of code) by the metrics computation tool GNATmetric.

### Benefits

The stricter SPARK rules are enforced on a hopefully large part of the program, which leads to better quality and maintainability, as error-prone features, such as side-effects in functions and aliasing between parameters, are avoided and others, such as use of pointers, are isolated to non-SPARK parts of the program. Individual and peer review processes can be lightened on those parts of the program in SPARK, since analysis automatically eliminates some categories of defects. Parts of the program that don't respect the SPARK rules are carefully isolated so they can be more thoroughly reviewed and tested.

### Impact on Process

After the initial pass of applying SPARK rules to the program, ongoing maintenance of SPARK code is similar to ongoing maintenance of Ada code, with a few additional rules, such as the need to avoid side-effects in functions and aliasing between parameters. These additional rules are checked automatically by running GNATprove on the modified program, which can be done either by the developer before pushing changes or by an automatic system (continuous builder, regression testsuite, etc.)

### Costs and Limitations

Pointer-heavy code needs to be rewritten to remove the use of pointers or to hide pointers from SPARK analysis, which may be difficult. The initial pass may require large, but shallow, rewrites in order to transform the code, for example to rewrite functions with side-effects into procedures.

## 3.1 Initial Setup

GNATprove can only be run on the sources of a GNAT project (a file with extension 'gpr' describing source files and switches to the GNAT compiler and other tools in the GNAT tool suite). As an installation check, we should start by applying GNATprove to the project without any `SPARK_Mode` markers:

```
> gnatprove -P my_project.gpr --mode=check -j0
```

The `-j0` switch analyzes files from the project in parallel, using as many cores as available, and the `--mode=check` switch runs GNATprove in fast checking mode. GNATprove should output the following messages:

```
Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: fast partial checking of SPARK legality rules ...
```

If you installed SPARK in a different repository from GNAT, you may get errors about project files not found if your project depends on XML/Ada, GNATCOLL, or any other project distributed with GNAT. In that case, you should update the environment variable `GPR_PROJECT_PATH` as indicated in the SPARK User's Guide: http://docs.adacore.com/spark2014-docs/html/ug/en/install.html

After you successfully run GNATprove without errors, choose a simple unit in the project, preferably a leaf unit that doesn't depend on other units, and apply the `SPARK_Mode` marker to it by adding the following pragma at the start of both the spec file (typically a file with extension 'ads') and the body file (typically a file with extension 'adb' for this unit:

```
pragma SPARK_Mode;
```

Then apply GNATprove to the project again:

```
> gnatprove -P my_project.gpr --mode=check -j0
```

GNATprove should output the following messages, stating that SPARK legality rules were checked on the unit marked, possibly followed by a number of error messages pointing to locations in the code where SPARK rules were violated:

```
Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: checking of SPARK legality rules ...
```

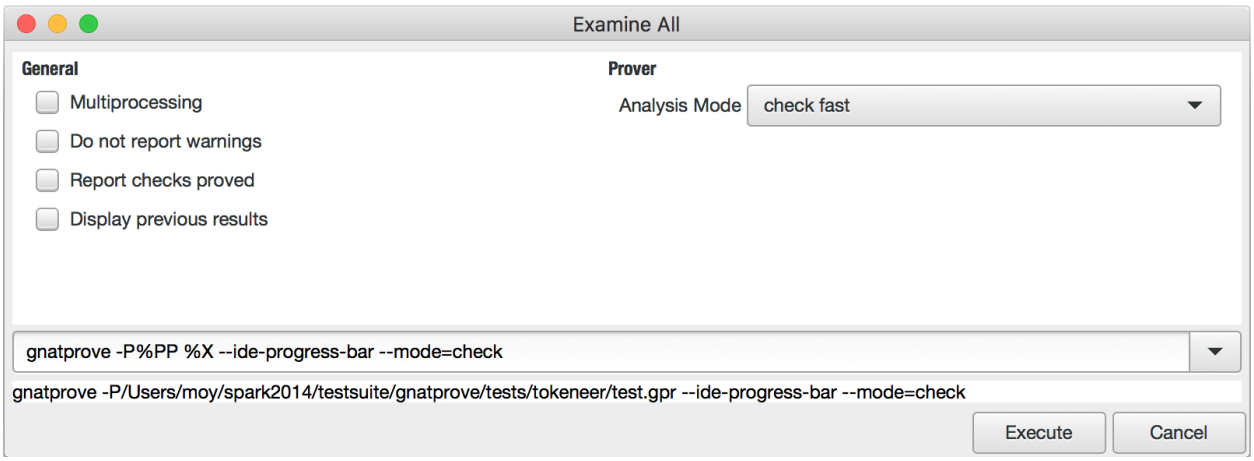
If you applied SPARK_Mode to a spec file without body (e.g. a unit defining only constants), GNATprove will notify you that no body was actually analyzed:

```
Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
warning: no bodies have been analyzed by GNATprove
enable analysis of a body using SPARK_Mode
```

At this point, you should switch to using GNAT Pro Studio (GPS), the integrated development environment provided with GNAT, in order to more easily interact with GNATprove. For example, GPS provides basic facilities for code navigation and location of errors that facilitate the adoption of SPARK. Open GPS on your project:

```
> gps -P my_project.gpr
```

There should be a SPARK menu available. Repeat the previous action within GPS by selecting the *SPARK → Examine All* menu, select the *check fast* mode in the popup window, and click *Execute*. The following snapshot shows the popup window from GPS with these settings:




GNATprove should output the same messages as before. If error messages are generated, they should now be located on the code that violates SPARK rules.

At this point, you managed to run GNATprove successfully on your project. The next step is to evaluate how much code can be identified as SPARK code. The easiest way to do that is to start by applying the marker `SPARK_Mode` to all files, using a script like the following shell script:

```
# mark.sh
for file in $@; do
   echo 'pragma SPARK_Mode;' > temp
   cat $file >> temp
   mv temp $file
done
```

or the following Python script:

```
# mark.py
import sys
for filename in sys.argv[1:]:
    with open(filename, 'r+') as f:
        content = f.read()
        f.seek(0, 0)
        f.write('pragma SPARK_Mode;\n' + content)
```

These scripts, when called on a list of files as command-line arguments, insert a line with the pragma SPARK_Mode at the beginning of each file. The list of files from a project can be obtained by calling GPRls when the project has main files (that is, it generates executables instead of libraries):

```
> gprls -P my_project.gpr --closure
```

or by calling GPRbuild with suitable arguments as follows:

```
> gprbuild -q -f -c -P my_project.gpr -gnatd.n | grep -v adainclude | sort | uniq
```

One you've obtained the list of Ada source files in the project by one of the two methods mentioned previously, you can systematically apply the `SPARK_Mode` marker to all the files with the small shell or Python script shown above:

```
> cat list_of_sources.txt | mark.sh
```

or:

```
> cat list_of_sources.txt | python mark.py
```

Then, open GPS on your project again and rerun the SPARK validity checker by again selecting menu *SPARK* → *Examine All*, select the *check fast* mode in the popup window that opens, and click *Execute*. This mode doesn't issue all possible violations of SPARK rules, but it runs much faster, so you should run in this mode in your initial runs. GNATprove should output error messages located on code that violates SPARK rules. The section *Dealing with SPARK Violations* explains how to address these violations by either modifying the code or excluding it from analysis.

After all the messages have been addressed, you should yet again rerun the SPARK validity checker, this time in a mode where all possible violations are issued. Do this by again selecting menu *SPARK* → *Examine All*, but now select the *check all* mode in the popup window that opens, and again click *Execute*. Again, GNATprove should output error messages located on code that violates SPARK rules, which should also be addressed as detailed in section *Dealing with SPARK Violations*.

A warning frequently issued by GNATprove at this stage looks like the following:

```
warning: no Global contract available for "F"
warning: assuming "F" has no effect on global items
```

This warning simply informs you that GNATprove could not compute a summary of the global variables read and written by subprogram `F`, either because it comes from an externally built library (such as the GNAT standard library, or XML/Ada) or because the implementation for `F` is not available to the analysis (for example if the code was not yet developed, the subprogram is imported, or the file with `F`'s implementation was excluded from analysis). You can provide this information to GNATprove by adding a Global contract to `F`'s declaration (see the section *Global Contract*). Alternatively, you can silence this specific warning by adding the following pragma either in the files that raise this warning or in a global configuration pragma file:

```
pragma Warnings (Off, "no Global Contract available",
                 Reason => "External subprograms have no effect on globals");
```

Note that, if required, you can silence all warnings from GNATprove with the `--warnings=off` switch.

## 3.2 Dealing with SPARK Violations

For each violation reported by GNATprove, you must decide whether to modify the code to make it respect the constraints of the SPARK subset or to exclude the code from analysis. If you make the first choice, GNATprove will be able to analyze the modified code; for the second choice, the code will be ignored during the analysis. It is thus preferable for you to modify the code whenever possible and to exclude code from analysis only as a last resort.

### 3.2.1 Excluding Code From Analysis

There are multiple methods for excluding code from analysis. Depending on the location of the violation, it may be more appropriate to exclude the enclosing subprogram or package or the complete enclosing unit.

#### Excluding a Subprogram From Analysis

When a violation occurs in a subprogram body, you can exclude that specific subprogram body from analysis by annotating it with `SPARK_Mode` aspect with value `Off` as follows:

```
procedure Proc_To_Exclude (..) with SPARK_Mode => Off is ...
function Func_To_Exclude (..) return T with SPARK_Mode => Off is ...
```

When the violation occurs in the subprogram spec, you must exclude both the spec and body from analysis by annotating both with `SPARK_Mode` aspect with value `Off`. The annotation on the subprogram body is given above and the annotation on the subprogram spec is similar:

```
procedure Proc_To_Exclude (..) with SPARK_Mode => Off;
function Func_To_Exclude (..) return T with SPARK_Mode => Off;
```

Only top-level subprograms can be excluded from analysis, i.e. subprogram units or subprograms declared inside package units, but not nested subprograms declared inside other subprograms. If a violation occurs inside a nested subprogram, you must exclude the enclosing top-level subprogram from analysis.

When only the subprogram body is excluded from analysis, the subprogram can still be called in SPARK code. When you exclude both the subprogram spec and body from analysis, you must also exclude all code that calls the subprogram.

### Excluding a Package From Analysis

Just as with subprograms, only top-level packages can be excluded from analysis, i.e. package units or packages declared inside package units, but not nested packages declared inside subprograms. If a violation occurs inside a nested package, you need to exclude the enclosing top-level subprogram from analysis. The case of local packages declared inside packages is similar to the case of subprograms, so in the following we only consider package units.

When a violation occurs in a package body, either it occurs inside a subprogram or package in this package body, in which case you can exclude just that subprogram or package from analysis or you can exclude the complete package body from analysis by removing the pragma `SPARK_Mode` that was inserted at the start of the file. In that mode, you can still analyze subprograms and packages declared inside the package body by annotating them with a `SPARK_Mode` aspect with value `On` as follows:

```
--  no pragma SPARK_Mode here
package body Pack_To_Exclude is ...
   procedure Proc_To_Analyze (..) with SPARK_Mode => On is ...
   package body Pack_To_Analyze with SPARK_Mode => On is ...
end Pack_To_Exclude;
```

When the violation occurs in the package spec, there are three possibities: First, the violation can occur inside the declaration of a subprogram or package in the package spec. In that case, you can exclude just that subprogram or package from analysis by excluding both its spec and the corresponding body from analysis by annotating them with a `SPARK_Mode` aspect with value `Off` as follows:

```
pragma SPARK_Mode;
package Pack_To_Analyze is
   procedure Proc_To_Exclude (..) with SPARK_Mode => Off;
   package Pack_To_Exclude with SPARK_Mode => Off is ...
end Pack_To_Analyze;

pragma SPARK_Mode;
package body Pack_To_Analyze is ...
   procedure Proc_To_Exclude (..) with SPARK_Mode => Off is ...
   package body Pack_To_Exclude with SPARK_Mode => Off is ...
end Pack_To_Analyze;
```

Second, the violation can occur directly inside the private part of the package spec. In that case, you can exclude the private part of the package from analysis by inserting a pragma `SPARK_Mode` with value `Off` at the start of the private part and removing the pragma `SPARK_Mode` that was inserted at the start of the file containing the package body. In that mode, entities declared in the visible part of the package spec, such as types, variables, and subprograms, can still be used in SPARK code in other units, provided these declarations do not violate SPARK rules. In addition, it's

possible to analyze subprograms or packages declared inside the package by annotating them with a SPARK_Mode aspect with value On as follows:

```
pragma SPARK_Mode;
package Pack_To_Use is ...
   --  declarations that can be used in SPARK code
private
   pragma SPARK_Mode (Off);
   --  declarations that cannot be used in SPARK code
end Pack_To_Use;

--  no pragma SPARK_Mode here
package body Pack_To_Use is ...
   procedure Proc_To_Analyze (..) with SPARK_Mode => On is ...
   package body Pack_To_Analyze with SPARK_Mode => On is ...
end Pack_To_Use;
```

Finally, the violation can occur directly inside the package spec. In that case, you can exclude the complete package from analysis by removing the pragma SPARK_Mode that was inserted at the start of both the files for the package spec and the package body. In that mode, entities declared in the package spec, such as types, variables, and subprograms, can still be used in SPARK code in other units, provided these declarations do not violate SPARK rules. In addition, it's also possible to analyze subprograms or packages declared inside the package, by annotating them with a SPARK_Mode aspect with value On as follows:

```
--  no pragma SPARK_Mode here
package Pack_To_Exclude is ...
   procedure Proc_To_Analyze (..) with SPARK_Mode => On;
   package Pack_To_Analyze with SPARK_Mode => On is ...
end Pack_To_Exclude;

--  no pragma SPARK_Mode here
package body Pack_To_Exclude is ...
   procedure Proc_To_Analyze (..) with SPARK_Mode => On is ...
   package body Pack_To_Analyze with SPARK_Mode => On is ...
end Pack_To_Exclude;
```

Note that cases 2 and 3 above are not exclusive: the violations of case 2 are in fact included in those of case 3. In case 2, all declarations in the visible part of the package are analyzed as well as their bodies when explicitly marked with a SPARK_Mode aspect. In case 3, only those declarations and bodies explicitly marked with a SPARK_Mode aspect are analyzed.

### 3.2.2 Modifying Code To Remove SPARK Violations

In many cases, code can be modified so that either SPARK violations are removed completely or can be moved to some part of the code that does not prevent most of the code from being analyzed. In general, this is good because SPARK violations point to features that can easily lead to code that is more difficult to maintain (such as side effects in functions) or to understand (such as pointers). Below, we consider typical SPARK violations found in Ada code and how to address each by modifying the code. When code modification is not possible or too complex/costly, the code with the violation should be excluded from analysis by following the recommendations of the previous section. The following table lists the main restrictions of SPARK that lead to violations in Ada code and how they are typically addressed, as detailed in the rest of this section.

| | How to remove the violation? | How to hide the violation? |
|---|---|---|
| Use of access type | Use references, addresses, or indexes in an array or a collection | Use a private type, defined as access type in a private section marked `SPARK_Mode Off` |
| Side-effect in function | Transform function in procedure with additional parameter for result | Mark function body with `SPARK_Mode Off` and function spec with `Global => null` to hide side-effect |
| Exception handler | Use result value to notify caller of error when recovery is required | Split subprogram into functionality without exception handler, and wrapper with exception handler marked with `SPARK_Mode Off` |

In the following, we consider the error messages that are issued in each case.

### access to "T" is not allowed in SPARK

See 'access type is not allowed in SPARK'

### access type is not allowed in SPARK

These errors are issued on uses of access types ('pointers'). For example:

```
Data1 : Integer;
Data2 : Boolean;
Data3 : access Integer;  --<<--  VIOLATION

procedure Operate is
begin
   Data1 := 42;
   Data2 := False;
   Data3.all := 42;  --<<--  VIOLATION
end Operate;
```

In some cases, the uses of access types can be removed from the subprogram into a helper subprogram, which is then excluded from analysis. For example, we can modify the code above as follows, where both the declaration of global variable `Data3` of access type and the assignment to `Data3.all` are grouped in a package body `Memory_Accesses` that is excluded from analysis, while the package spec for `Memory_Accesses` can be used in SPARK code:

```
Data1 : Integer;
Data2 : Boolean;

package Memory_Accesses is
   procedure Write_Data3 (V : Integer);
end Memory_Accesses;

package body Memory_Accesses
  with SPARK_Mode => Off
is
   Data3 : access Integer;

   procedure Write_Data3 (V : Integer) is
   begin
          Data3.all := V;
   end Write_Data3;
end Memory_Accesses;
```

```
procedure Operate is
begin
   Data1 := 42;
   Data2 := False;
   Memory_Accesses.Write_Data3 (42);
end Operate;
```

In other cases, the access type needs to be visible from client code, but the fact that it's implemented as an access type need not be visible to client code. Here's an example of such a case:

```
type Ptr is access Integer;  --<<--  VIOLATION

procedure Operate (Data1, Data2, Data3 : Ptr) is
begin
   Data1.all := Data2.all;
   Data2.all := Data2.all + Data3.all;
   Data3.all := 42;
end Operate;
```

In that case, the access type can be made a private type of either a local package or of package defined in a different unit, whose private part (and possibly also its package body) is excluded from analysis. For example, we can modify the code above as follows, where the type `Ptr` together with accessors to query and update objects of type `Ptr` are grouped in package `Ptr_Accesses`:

```
package Ptr_Accesses is
   type Ptr is private;
   function Get (X : Ptr) return Integer;
   procedure Set (X : Ptr; V : Integer);
private
   pragma SPARK_Mode (Off);
   type Ptr is access Integer;
end Ptr_Accesses;

package body Ptr_Accesses
  with SPARK_Mode => Off
is
   function Get (X : Ptr) return Integer is (X.all);
   procedure Set (X : Ptr; V : Integer) is
   begin
          X.all := V;
   end Set;
end Ptr_Accesses;

procedure Operate (Data1, Data2, Data3 : Ptr_Accesses.Ptr) is
   use Ptr_Accesses;
begin
   Set (Data1, Get (Data2));
   Set (Data2, Get (Data2) + Get (Data3));
   Set (Data3, 42);
end Operate;
```

### explicit dereference is not allowed in SPARK

See 'access type is not allowed in SPARK'

### function with "in out" parameter is not allowed in SPARK

This error is issued on a function with an 'in out' parameter. For example:

```
function Increment_And_Add (X, Y : in out Integer) return Integer is
   --<<--  VIOLATION
begin
   X := X + 1;
   Y := Y + 1;
   return X + Y;
end Increment_And_Add;
```

The function can be transformed into a procedure by adding an 'out' parameter for the returned value, as follows:

```
procedure Increment_And_Add (X, Y : in out Integer; Result : out Integer) is
begin
   X := X + 1;
   Y := Y + 1;
   Result := X + Y;
end Increment_And_Add;
```

### function with output global "X" is not allowed in SPARK

This error is issued on a function with a side-effect on variables in scope. For example:

```
Count : Integer := 0;

function Increment return Integer is
begin
   Count := Count + 1;  --<<--  VIOLATION
   return Count;
end Increment;
```

The function can be transformed into a procedure by adding an 'out' parameter for the returned value, as follows:

```
procedure Increment (Result : out Integer) is
begin
   Count := Count + 1;
   Result := Count;
end Increment;
```

Alternatively, when the side-effects have no influence on the properties to verify, they can be masked to the analysis. For example, consider a procedure `Log` that writes global data, causing all of its callers to have side-effects:

```
Last : Integer := 0;

procedure Log (X : Integer) is
begin
   Last := X;
end Log;

function Increment_And_Log (X : Integer) return Integer is
begin
   Log (X);  --<<--  VIOLATION
   return X + 1;
end Increment_And_Log;
```

A legitimate solution here is to mask the side-effects in procedure `Log` for the analysis, by annotating the spec of `Log` with an aspect `Global` with value `null` and by excluding the body of `Log` from analysis:

```
procedure Log (X : Integer)
  with Global => null;

Last : Integer := 0;

procedure Log (X : Integer)
  with SPARK_Mode => Off
is
begin
   Last := X;
end Log;

function Increment_And_Log (X : Integer) return Integer is
begin
   Log (X);
   return X + 1;
end Increment_And_Log;
```

### handler is not allowed in SPARK

This error is issued on exception handlers. For example, on the following code:

```
Not_Found : exception;

procedure Find_Before_Delim
  (S        : String;
   C, Delim : Character;
   Found    : out Boolean;
   Position : out Positive)
is
begin
   for J in S'Range loop
          if S(J) = Delim then
         raise Not_Found;
      elsif S(J) = C then
             Position := J;
             Found := True;
                 Return;
      end if;
   end loop;
   raise Not_Found;
exception              --<<--  VIOLATION
   when Not_Found =>
          Position := 1;
      Found := False;
end Find_Before_Delim;
```

The subprogram with an exception handler can usually be split between core functionality, which may raise exceptions but does not contain an exception handler and thus can be analyzed, and a wrapper calling the core functionality, which contains the exception handler and is excluded from analysis. For example, we can modify the code above to perform the search for a character in function `Find_Before_Delim`, which raises an exception if the desired character is not found before either the delimiter or the end of the string, and a procedure `Find_Before_Delim`, which wraps the call to function `Find_Before_Delim`, as follows:

```
Not_Found : exception;

function Find_Before_Delim (S : String; C, Delim : Character) return Positive is
begin
   for J in S'Range loop
      if S(J) = Delim then
         raise Not_Found;
      elsif S(J) = C then
         return J;
      end if;
   end loop;
   raise Not_Found;
end Find_Before_Delim;

procedure Find_Before_Delim
  (S         : String;
   C, Delim : Character;
   Found     : out Boolean;
   Position : out Positive)
  with SPARK_Mode => Off
is
begin
   Position := Find_Before_Delim (S, C, Delim);
   Found := True;
exception
   when Not_Found =>
      Position := 1;
      Found := False;
end Find_Before_Delim;
```

### side effects of function "F" are not modeled in SPARK

This error is issued on a call to a function with side-effects on variables in scope. Note that a corresponding error 'function with output global "X" is not allowed in SPARK' will also be issued on function `F` if it's marked `SPARK_Mode` with value `On` (either directly or in a region of code marked as such). For example, on the following code, calling the function `Increment_And_Log` seen previously:

```
procedure Call_Increment_And_Log is
   X : Integer;
begin
   X := Increment_And_Log (10);  --<<--  VIOLATION
end Call_Increment_And_Log;
```

The called function can be transformed into a procedure as seen previously. If it's not marked `SPARK_Mode` with value `On`, a legitimate solution might be to mask its side-effects for the analysis, by annotating its spec with a `Global` aspect with value `null`.

CHAPTER

FOUR

# BRONZE LEVEL - INITIALIZATION AND CORRECT DATA FLOW

The goal of reaching this level is making sure that no uninitialized data can ever be read and, optionally, preventing unintended access to global variables. This also ensures no possible interference between parameters and global variables, meaning that the same variable isn't passed multiple times to a subprogram, either as a parameter or global variable.

### Benefits

The SPARK code is guaranteed to be free from a number of defects: no reads of uninitialized variables, no possible interference between parameters and global variables, no unintended access to global variables.

When `Global` contracts are used to specify which global variables are read and/or written by subprograms, maintenance is facilitated by a clear documentation of intent, which is checked automatically by running GNATprove, so that any mismatch between the implementation and the specification is reported.

### Impact on Process

An initial pass is required where flow analysis is turned on and the resulting messages are resolved either by rewriting code or justifying any false alarms. Once this is complete, ongoing maintenance can maintain the same guarantees at a low cost. A few simple idioms can be used to avoid most false alarms and the remaining false alarms can be easily justified.

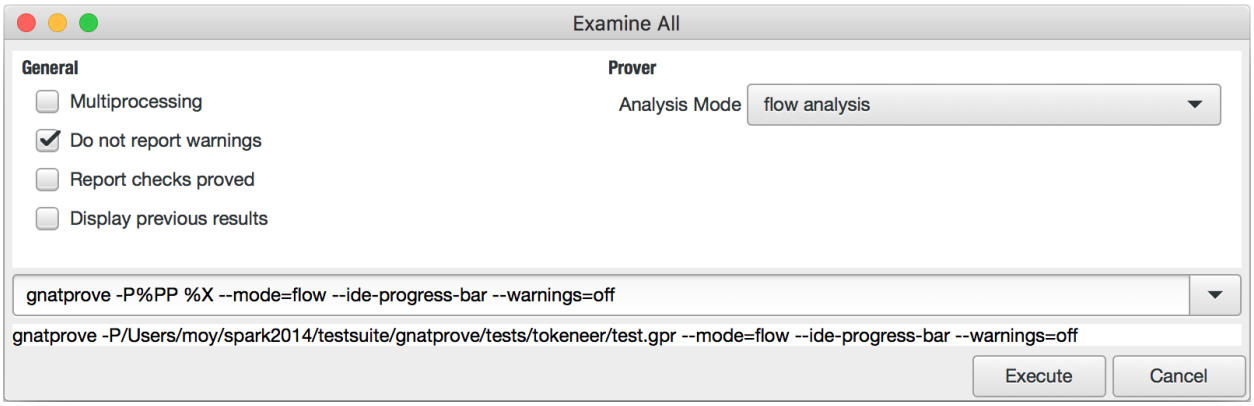
### Costs and Limitations

The initial pass may require a substantial effort to get rid of all false alarms, depending on the coding style adopted up to that point. The analysis may take a long time, up to an hour, on large programs but it is guaranteed to terminate. Flow analysis is, by construction, limited to local understanding of the code, with no knowledge of values (only code paths) and handling of composite variables is only through calls, rather than component by component, which may lead to false alarms.

## 4.1 Running GNATprove in Flow Analysis Mode

Two distinct static analyses are performed by GNATprove. Flow analysis is the fastest and requires no user-supplied annotations. It tracks the flow of information between variables on a per subprogram basis. In particular, it allows finding every potential use of uninitialized data. The second analysis, proof, will be described in the sections on Silver and Gold levels.

To run GNATprove in flow analysis mode on your project, select the *SPARK → Examine All* menu. In the GPS panel, select the *flow analysis* mode, check the *Do not report warnings* box, uncheck the *Report checks proved* box, and click *Execute*. The following snapshot shows the popup window from GPS with these settings:




GNATprove should output the following messages, possibly followed by a number of messages pointing to potential problems in your program:

```
Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: analysis of data and information flow ...
```

The following messages output by GNATprove are check messages and should have the form:

```
medium: "V" might not be initialized
```

Listed first is the severity of the check, which is one of *low*, *medium*, or *high*. It reflects both the likelihood that the reported problem is indeed a bug and the criticality if it is a bug. Following the colon is the type of check message, here a potential read of an uninitialized variable. They'll be located at the point in your code where the error can occur. The corresponding line in GPS will be highlighted in red.

Flow analysis can issue several types of check messages. In this document, we concentrate on the two most common ones. Initialization checks relate to uses of uninitialized data and are described in section *Initialization Checks*. Section *Aliasing* discusses check messages related to aliasing of subprogram parameters and global variables. Other check messages can also be issued when volatile variables or tasking constructs are used. You can find more information about these additional checks in http://docs.adacore.com/spark2014-docs/html/ug/en/source/how_to_view_gnatprove_output.html#description-of-messages.

Once you have addressed each check message, you can re-reun flow analysis with the *Report checks proved* box checked to see the verifications successfully performed by GNATprove. This time, it should only issue 'info' messages, highlighted in green in GPS, like the following:

```
info: initialization of "V" proved
```

Flow analysis can also generate useful warnings about dead code, unused variables or incorrect parameter modes. To achieve this level, it may be interesting to look at these warnings. We explain how this can be done in section *Flow Analysis Warnings*.

As further optional steps in this level, critical parts of the program can be annotated to make sure they don't make unintended accesses to global data. This activity is explained in section *Global Annotations*.

## 4.2 Initialization Checks

Initialization checks are the most common check messages issued by GNATprove in flow analysis mode. Indeed, each time a variable is read or returned by a subprogram, GNATprove performs a check to make sure it has been initialized. A failed initialization check message can have one of the two forms:

```
high: "V" is not initialized
```

or:

```
medium: "V" might not be initialized
```

Choose a unit in which GNATprove reports an unproved initialization check and open it in GPS. You can launch flow analysis on only this unit by opening the *SPARK → Examine File* menu, selecting the *flow analysis* mode in the GPS panel, checking the *Do not report warnings* box, unchecking the *Report checks proved* box, and clicking *Execute*. To investigate an unproved initialization check, click on the corresponding check message in the GPS *Locations* tab. The editor should move to the corresponding location in your program.

Not all unproved initialization checks denote actual reads of uninitialized variables: SPARK features a stronger initialization policy than Ada and the verification of initialization of variables in GNATprove suffers from shortcomings. Determining whether an initialization check issued by GNATprove is a real error is done by code review and is usually straightforward. While actual reads of uninitialized data must be corrected, check messages that don't correspond to actual errors (called 'false alarms' or 'false positives') can be either 'justified', that is, annotated with a proper justification (see section on *Justifying Unproved Check Messages*), or worked around. In the rest of this section, we review the most common cases where GNATprove may produce unproved initialization checks. We then describe how the code can be changed to avoid false alarms or, alternately, explain how they can be justified.

### 4.2.1 SPARK Strong Data Initialization Policy

GNATprove verifies data initialization modularly on a per subprogram basis. To allow this verification, the SPARK language requires a stronger data initialization policy than standard Ada: you should initialize every global variable that is read by a subprogram and every parameter of mode 'in' or 'in out' on entry to the subprogram.

```
procedure P (X : in out Integer) is
begin
    X := X + 1;  --<<--  ok
end P;
X : Integer;
P (X);  --<<--  high: "X" is not initialized
```

Parameters of mode 'out' are considered to always be uninitialized on subprogram entry so their value should not be read prior to initialization:

```
procedure P (X : out Integer) is
begin
    X := X + 1;  --<<--  high: "X" is not initialized
end P;
X : Integer;
P (X);  --<<--  ok
```

The expression returned from a function and the parameters of mode 'out' of a procedure should be initialized on the subprogram's return:

```
procedure P (X : out Integer) is  --<<--  high: "X" is not initialized in P
begin
```

```
   null;
end P;
```

If a global variable is completely initialized by a subprogram, it's considered as an output of the subprogram and SPARK does not require it to be initialized at subprogram entry:

```
G : Integer;
procedure P is  --<<--  info: initialization of "G" proved
begin
   G := 0;
end P;
```

You can find more information about SPARK's data initialization policy in the SPARK User's Guide: http://docs.adacore.com/spark2014-docs/html/ug/en/source/language_restrictions.html#data-initialization-policy.

In some cases, this initialization policy may be too constraining. For example, consider the following `Search` procedure:

```
procedure Search (A      : Nat_Array;
                 E      : Natural;
                 Found  : out Boolean;
                 Result : out Positive)
is
begin
   for I in A'Range loop
      if A (I) = E then
         Found := True;
         Result := I;
         return;
      end if;
   end loop;
   Found := False;
end Search;
```

This code is perfectly safe as long as the value of `Result` is only read when `Found` is `True`. Nevertheless, flow analysis issues an unproved check on `Result`'s declaration:

```
medium: "Result" might not be initialized in "Search"
```

You can consider this check message as a false alarm and can easily either justify it (see section on *Justifying Unproved Check Messages*) or work around it, depending on what is more appropriate. A safer alternative, however, is to always initialize `Result` on all paths through `Search`.

### 4.2.2 Handling of Composite Objects as a Whole

It follows from the SPARK initialization policy that out parameters of a composite type must be completely defined by the subprogram. One side-effect of this is that it makes it impossible to fully initialize a record object by successively initializing each component through procedure calls:

```
type R is record
   F1 : Integer;
   F2 : Integer;
end record;

procedure Init_F1 (X : out R) is
  --<<--  high: "X.F2" is not initialized in "Init_F1"
```

```
begin
   X.F1 := 0;
end Init_F1;

procedure Init_F2 (X : in out R) is
begin
   X.F2 := 0;
end Init_F2;

X : R;
Init_F1 (X);
Init_F2 (X);
```

### 4.2.3 Imprecise Handling of Arrays

Though record objects are treated as composites for inter-procedural data initialization policy, the initialization status of each record component is tracked independently inside a single subprogram. For example, a record can be initialized by successive assignments into each of its components:

```
X : R;
X.F1 := 0;
X.F2 := 0;
P (X);  --<<--  info: initialization of "Y.F1" proved
        --<<--  info: initialization of "Y.F2" proved
```

The same isn't true for arrays because checking that each index of an array has been initialized in general requires dynamic evaluation of expressions (to compute which indexes have been assigned to). As a consequence, GNATprove considers an update of an array variable as a read of this variable and issues an unproved initialization check every time an assignment is done into a potentially uninitialized array. It then assumes that the whole array has been initialized for the rest of the analysis. Specifically, initializing an array element-by-element will result in an unproved initialization check:

```
A : Nat_Array (1 .. 3);
A (1) := 1;  --<<--  medium: "A" might not be initialized
A (2) := 2;  --<<--  info: initialization of "A" proved
```

### 4.2.4 Value Dependency

Flow analysis is not value dependent, meaning that it is not influenced by the actual value of expressions. As a consequence, it's not able to determine that some paths of a program are impossible, so it may issue unproved checks on such a path. For example, in the following program, GNATprove cannot verify that `X1` is initialized in the assignment to `X2` even though the two if statements share the same condition:

```
X1 : Integer;
X2 : Integer;
if X < C then
   X1 := 0;
end if;
if X < C then
   X2 := X1;  --<<--  medium: "X1" might not be initialized
end if;
```

### 4.2.5 Rewriting the Code to Avoid False Alarms

In cases where the code can be modified, it may be a good idea to rewrite it so that GNATprove can successfully verify data initialization. In the following, we list these modifications, starting from the least intrusive and ending with the most intrusive. It's best to initialize variables at declaration and this is the recommended work-around whenever possible since it only requires modifying the variable declaration and is not very error-prone. However, it is impossible for variables of a private type and may be difficult for complex data and inefficient for large structures.

```
A : Nat_Array (1 .. 3) := (others => 0);
A (1) := 1;  --<<--  info: initialization of "A" proved
A (2) := 2;  --<<--  info: initialization of "A" proved
```

Another option is to add a default to the variable's type, though this is more intrusive as it impacts every variable of that type with default initialization. For example, if the initializing expression takes time to execute and there are thousands of variables of this type which are initialized by default, this may impact the overall running time of the application. On the other hand, it's especially interesting for private types, for which the previous work-around is not applicable. A default initial value can be defined for scalar types using `Default_Value`, for array types using `Default_Component_Value`, and for record types by introducing a default for each record component:

```
type My_Int is new Integer with Default_Value => 0;
type Nat_Array is array (Positive range <>) of Natural with
   Default_Component_Value => 0;
type R is record
   F1 : Integer := 0;
   F2 : My_Int;
end record;
```

You can also annotate private types with the `Default_Initial_Condition` aspect, which allows defining a property which should hold whenever a variable of this type is initialized by default. When no property is provided, it defaults to `True` and implies that the type can be safely initialized by default. If the full view of the type is in SPARK, a single initialization check will be issued for such a type at the type's declaration:

```
type Stack is private with Default_Initial_Condition;
type Stack is record
   Size    : Natural := 0;
   Content : Nat_Array (1 .. Max);
end record;  --<<--  medium: type "Stack" is not fully initialized

S : Stack;
P (S);  --<<--  info: initialization of "S.Size" proved
        --<<--  info: initialization of "S.Content" proved
```

Yet another option is to refactor code to respect the SPARK data initialization policy. Specifically, initialize every components of a record object in a single procedure and always initialize subprogram outputs. Alternatively, partial initialization (only on some program paths) can be represented by a variant record:

```
type Optional_Result (Found : Boolean) is record
   case Found is
      when False => null;
      when True  => Content : Positive;
   end case;
end record;

procedure Search (A      : Nat_Array;
                  E      : Natural;
                  Result : out Optional_Result)
is
```

```
begin
   for I in A'Range loop
      if A (I) = E then
         Result := (Found => True, Content => I);
         return;
      end if;
   end loop;
   Result := (Found => False);
end Search;
```

### 4.2.6 Justifying Unproved Check Messages

You can selectively accept check messages, like those emitted for data initialization, by supplying an appropriate justification. When you do that, the tool silently assumes the data affected by the justified check has been initialized and won't warn again about its uses. To annotate a check, add a `pragma Annotate` in the source code on the line following the failed initialization check:

```
pragma Annotate (GNATprove, Category, Pattern, Reason);
```

A `pragma Annotate` expects exactly 4 arguments. The first is fixed and should always be `GNATprove`. The second argument, named `Category`, can be either `False_Positive` or `Intentional`. `False_Positive` should be used when the data is initialized by the program but GNATprove is unable to verify it, while `Intentional` should be used when the variable is not initialized, but for some reason this is not a problem; some examples will be given later. The third argument, named `Pattern`, should be a part of the check message. For initialization checks, "'X" might not be initialized' or "'X" is not initialized', depending on the message, is appropriate. Finally, the last argument is the most important. It stores an explanation of why the check was accepted. It should allow reviewing the justification easily. A rule that's often applied in practice is that the reason should identify the author of the justification, using the format '<initials> <reason>', for example 'YM variable cannot be zero here'.

You can find a complete description of how checks can be justified in the SPARK User's Guide: http://docs.adacore.com/spark2014-docs/html/ug/en/source/how_to_use_gnatprove_in_a_team.html#justifying-check-messages.

On the code below, GNATprove is unable to verify that the array A is initialized by successive initialization of its elements:

```
A : Nat_Array (1 .. 3);
A (1) := 1;
pragma Annotate
  (GNATprove, False_Positive, """A"" might not be initialized",
   String'("A is properly initialized by these three successive"
           & " assignments"));
A (2) := 2;
A (3) := 3;
```

Since the array `A` is correctly initialized by the code above, the annotation falls in the category `False_Positive`. Note that the `pragma Annotate` must be located just after the line for which the check message is issued.

Because SPARK enforces a stronger initialization policy than Ada, you may want to justify a check message for a variable that may not be completely initialized. In this case, you should be especially careful to precisely state the reasons why the check message is acceptable since the code may change later and SPARK would not spot any invalid usage of the annotated variable. For example, when we accept the check message stating that `Result` may not be initialized by `Search`, we must explain precisely what is required both from the implementation and from the callers to make the review valid:

```
procedure Search (A      : Nat_Array;
                  E      : Natural;
```

```
                  Found  : out Boolean;
                  Result : out Positive);
pragma Annotate
  (GNATprove, Intentional, """Result"" might not be initialized",
   String'("Result is always initialized when Found is True and never"
           & " read otherwise");
```

As another example, we can assume every instance of a stack is initialized by default only under some assumptions that should be recorded in the justification message:

```
type Stack is private with Default_Initial_Condition;
type Stack is record
   Size    : Natural := 0;
   Content : Nat_Array (1 .. Max);
end record;
pragma Annotate
  (GNATprove, Intentional, """Stack"" is not fully initialized",
   String'("The only indexes that can be accessed in a stack are"
           & " those smaller than Size. These indexes will always"
           & " have been initialized when Size is increased."));
```

On existing, thoroughly tested code, unconditional reads of uninitialized data are rather unlikely. Neverthless, there may be a path through the program where an uninitialized variable can be read. Before justifying an unproved initialization check, it's important to understand why it's not proved and what are the assumptions conveyed to the tool when justifying it. The result of this analysis should then be stored inside the reason field of the `pragma Annotate` to simplify later reviews.

## 4.3 Aliasing

### 4.3.1 Detecting Possible Aliasing

In SPARK, an assignment to a variable cannot change the value of another variable. This is enforced by forbidding the use of access types ('pointers') and by restricting aliasing between parameters and global variables so that only benign aliasing is accepted (i.e. aliasing that does not cause interference).

A check message concerning a possible aliasing has the form:

```
high: formal parameter "X" and global "Y" are aliased (SPARK RM 6.4.2)
```

This message is warning that, for the call at the given location, the variable `Y` supplied for the formal parameter `X` of the subprogram was already visible in the subprogram. As a result, assignments to `Y` in the subprogram will affect the value of `X` and the converse holds too. This is detected as an error by GNATprove, which always assumes variables to be distinct.

As stated in the check message, the precise rules for aliasing are detailed in SPARK Reference Manual section 6.4.2. They can be summarized as follows:

Two out parameters should never be aliased. Notice that the trivial cases of parameter aliasing are already forbidden by Ada and reported as errors by the compiler, such as in the following subprogram:

```
procedure Swap (X, Y : in out Integer);

Swap (Z, Z);  --<<--  writable actual for "X" overlaps with actual for "Y"
```

An 'in' and 'an' out parameter should not be aliased:

```
procedure Move_X_To_Y (X : in T; Y : out T);

Move_X_To_Y (Z, Z);
   --<<--  high: formal parameters "X" and "Y" are aliased (SPARK RM 6.4.2)
```

As an exception, SPARK allows aliasing between an 'in' and an 'out' parameter if the 'in' parameter is always passed by copy. For example, if we change `T` to `Integer` in the previous example (so that the arguments are always passed by copy), GNATprove no longer outputs any unproved check message:

```
procedure Move_X_To_Y (X : in Integer; Y : out Integer);

Move_X_To_Y (Z, Z);  --<<--  ok
```

However, an 'out' parameter should never be aliased with a global variable referenced by the subprogram. This is really the same as aliasing between output parameters, but it cannot be reported by the compiler because it doesn't track uses of global variables:

```
procedure Swap_With_Y (X : in out Integer);

Swap_With_Y (Y);
   --<<--  high: formal parameter "X" and global "Y" are aliased (SPARK RM 6.4.2)
```

Note that aliasing between an 'out' parameter and a global variable is also forbidden even if the global variable is never written:

```
procedure Move_X_To_Y (Y : out Integer);

Move_X_To_Y (X);
   --<<--  high: formal parameter "Y" and global "X" are aliased (SPARK RM 6.4.2)
```

An 'in' parameter should not be aliased with a global variable written by the subprogram:

```
procedure Move_X_To_Y (X : in T);

Move_X_To_Y (Y);
   --<<--  high: formal parameter "X" and global "Y" are aliased (SPARK RM 6.4.2)
```

Just like aliasing between parameters, aliasing is allowed if the 'in' parameter is always passed by copy:

```
procedure Move_X_To_Y (X : in Integer);

Move_X_To_Y (Y);  --<<--  ok
```

Note that aliasing can also occur between parts of composite variables such as components of records or elements of arrays. You can find more information about aliasing in the SPARK User's Guide: http://docs.adacore.com/spark2014-docs/html/ug/en/source/language_restrictions.html#absence-of-interferences.

### 4.3.2 Dealing with Unproved Aliasing Checks

Complying with SPARK rules concerning aliasing usually requires refactoring the code. This is, in general, a good idea because aliasing can be the source of errors that are difficult to find since they only occur in some cases. When calling a subprogram with aliased parameters, there's a good chance of failing in a case the implementer of the subprogram has not considered and thus of triggering an inappropriate result. Furthermore, the behavior of a subprogram call when its parameter are aliased depends on how parameter are passed (by copy or by reference) and on the order in which the by-copy parameters, if any, are copied back. Since these are not specified by the Ada language, it may introduce either compiler or platform dependences in the behavior of the program.

It can be the case that GNATprove's analysis is not precise enough and that it issues an unproved check message in cases in which there really is no possible aliasing. This can be the case, for example, for aliasing between a subprogram input parameter and an output global variable referenced by the subprogram if the parameter is not of a by-copy type (a type mandated to be passed by value by the Ada Reference Manual) but for which the developer knows that, in her environment, the compiler indeed passes it by copy. In this case, the check message can be justified similarly to initialization checks:

```
type T is record
   F : Integer;
end record with
   Convention => C_Pass_By_Copy;

procedure Move_X_To_Y (X : in T);

Move_X_To_Y (Y);
pragma Annotate
  (GNATprove, False_Positive,
   "formal parameter ""X"" and global ""Y"" are aliased",
   String'("My compiler follows Ada RM-B-3 68 implementation advice"
           & " and passes variables of type T by copy as it uses the"
           & " C_Pass_By_Copy convention"));
```

GNATprove restrictions explained in the section about initialization checks can also lead to false alarms, in particular for aliasing between parts of composite objects. In the following example, `Only_Read_F2_Of_X` only references the component `F2` in `X`. But, since GNATprove handles composite global variables as a whole, it still emits an unproved aliasing check in this case, which can be justified as follows:

```
procedure Only_Read_F2_Of_X (Y : out Integer);

Only_Read_F2_Of_X (X.F1);
pragma Annotate
  (GNATprove, False_Positive,
   "formal parameter ""Y"" and global ""X"" are aliased",
   String'("Only_Read_F2_Of_X only references the component F2 in X"
           & " so no aliasing can be introduced with X.F1"));
```

In the same way, because it is not value dependent, flow analysis emits an unproved aliasing check when two (distinct) indices of an array are given as output parameters to a subprogram, which can be justified as follows:

```
pragma Assert (I = 2);
Swap (A (1), A (I));
pragma Annotate
  (GNATprove, False_Positive,
   "formal parameters ""X"" and ""Y"" might be aliased",
   String'("As I is equal to 2 prior to the call, A (1) and A (I) are"
           & " never aliased."));
```

## 4.4 Flow Analysis Warnings

Other than check messages, flow analysis can also issue warnings, which usually flag suspicious code that may be the sign of an error in the program. They should be inspected, but can be suppressed when they're deemed spurious, without risk of missing a critical issue for the soundness of the analysis. To see these warnings, run the tool in flow analysis mode with warnings enabled. Select *SPARK* → *Examine All* menu, in the GPS panel, select the *flow* mode, uncheck the *Do not report warnings* and *Report checks proved* boxes, and click *Execute*.

GNATprove warnings, like the compiler warnings, are associated with a source location and prefixed with the word 'warning':

```
warning: subprogram "Test" has no effect
```

You can suppress GNATprove warnings globally by using the switch `--warnings=off`, which is equivalent to checking the *Do not report warnings* box in GPS, or locally by using `pragma Warnings`. For example, the above warning can be suppressed by switching off local warnings with the above message around the declaration of the procedure `Test` as follows:

```
pragma Warnings
  (Off, "subprogram ""Test"" has no effect",
   Reason => "Written to demonstrate GNATprove's capabilities");

procedure Test;

pragma Warnings (On, "subprogram ""Test"" has no effect");
```

A common rule applied in practice is that the reason should identify the author of the pragma, using the format '<initials> <reason>', for example 'CD subprogram is only a test'.

How warnings can be suppressed in GNATprove is described in the SPARK User's Guide: http://docs.adacore.com/spark2014-docs/html/ug/en/source/how_to_use_gnatprove_in_a_team.html#suppressing-warnings.

The rest of this section lists warnings that may be issued by GNATprove and explains the meaning of each.

### initialization of X has no effect

Flow analysis tracks flow of information between variables. While doing so, it can detect cases where the initial value of a variable is never used to compute the value of any object. It reports it with a warning:

```
function Init_Result_Twice return Integer is
   Result : Integer := 0;
      --<<--  warning initialization of Result has no effect
begin
   Result := 1;
   return Result;
end Init_Result_Twice;
```

### unused assignment

Flow analysis also detects assignments which store into a variable a value that will never be read:

```
procedure Write_X_Twice (X : out Integer) is
begin
   X := 1;  --<<--  warning: unused assignment
   X := 2;
end Write_X_Twice;
```

Note that flow analysis is not value dependent. As a consequence, it cannot detect cases when an assignment is useless because it stores the same value that was previously stored in the variable:

```
procedure Write_X_To_Same (X : in out Integer) is
   Y : Integer;
begin
```

```
   Y := X;
   X := Y;  --<<--  no warning
end Write_X_To_Same;
```

### "X" is not modified, could be IN

Flow analysis also checks the modes of subprogram parameters. It warns on 'in out' parameters whose value is never modified:

```
procedure Do_Not_Modify_X (X, Y : in out Integer) is
    --<<--  warning: "X" is not modified, could be IN
begin
   Y := Y + X;
end Do_Not_Modify_X;
```

### unused initial value of "X"

Flow analysis also detects an 'in' or 'in out' parameter whose initial value is never read by the program:

```
procedure Initialize_X (X : in out Integer) is
   --<<--  warning: unused initial value of "X"
begin
   X := 1;
end Initialize_X;
```

### statement has no effect

Flow analysis can detect a statement which has no effect on any output of the subprogram:

```
procedure Initialize_X (X : out Integer) is
   Y : Integer;
begin
   Set_To_One (Y);  --<<--  statement has no effect
   X := 1;
end Initialize_X;
```

### subprogram "S" has no effect

When a subprogram as a whole has no output or effect, it's also reported by GNATprove:

```
procedure Do_Nothing is
   --<<--  warning: subprogram "Do_Nothing" has no effect
begin
   null;
end Do_Nothing;
```

## 4.5 Global Annotations

### 4.5.1 Global Contract

In addition to what's been presented so far, you may want to use flow analysis to verify specific data-dependency relations. This can be done by providing the tool with additional `Global` contracts stating the set of global variables accessed by a subprogram. You need to only supply those contracts that you want to verify. Other contracts will be automatically inferred by the tool. The simplest form of data dependency contract states that a subprogram is not allowed to either read or modify global variables:

```
procedure Increment (X : in out Integer) with
   Global => null;
```

This construction uses the Ada 2012 aspect syntax. You must place it on the subprogram declaration if any, otherwise on the subprogram body. You can use an alternative notation based on pragmas if compatibility with older versions of Ada is required:

```
procedure Increment (X : in out Integer);
pragma Global (null);
```

This annotation is the most common one as most subprograms don't use global state. In its more complete form, the `Global` contract allows specifing precisely the set of variables that are read, updated, and initialized by the subprogram:

```
procedure P with
   Global =>
      (Input  => (X1, X2, X3),
      --  variables read but not written by P (same as 'in' parameters)
       In_Out => (Y1, Y2, Y3),
      --  variables read and written by P (same as 'in out' parameters)
       Output => (Z1, Z2, Z3));
      --  variables initialized by P (same as 'out' parameters)
```

The use of `Global` contracts is not mandatory. However, whenever a contract is provided, it must be correct and complete: that is, it must mention every global variable accessed by the subprogram with the correct mode. Similarly to subprogram parameter modes, global contracts are checked by the tool in flow analysis mode and checks and warnings are issued in case of non-conformance. To verify manually supplied global contracts, run GNATprove in flow analysis mode by selecting the *SPARK → Examine File* menu, selecting the *flow* mode in the GPS panel, checking the *Do not report warnings* box, uncheck the *Report checks proved* box, and clicking *Execute*.

Global contracts are described more completely in the SPARK User's Guide: http://docs.adacore.com/spark2014-docs/html/ug/en/source/subprogram_contracts.html#data-dependencies.

### 4.5.2 Constants with Variable Inputs

When a subprogram accesses a constant whose value depends on variable inputs (that is, on the value of variables or of other constants with variable inputs), it must be listed in the `Global` contract of the subprogram, if any. This may come as a surprise to users. However, this is required to properly verify every flow of information between variables of the program. As an example, on the following program, the dependency of `Set_X_To_C` on the value of `Y` is expressed by the constant with the variable input `C` appearing in its `Global` contract:

```
Y : Integer := 0;
procedure Set_X_To_Y (X : out Integer) with
   Global => (Input => Y)  --<<--  Y is an input of Set_X_To_Y
is
```

```
   C : constant Integer := Y;
   procedure Set_X_To_C with
      Global => (Input => C, Output => X)
      --<<--  the dependency on Y is visible through the dependency on C
   is
   begin
      X := C;
   end Set_X_To_C;
begin
   Set_X_To_C;
end Set_X_To_Y;
```

You can find more information about constants with variable inputs in the SPARK User's Guide: http://docs.adacore.com/spark2014-docs/html/ug/en/source/package_contracts.html#special-cases-of-state-abstraction.

### 4.5.3 Abstract State

Sometimes, you may want to annotate a subprogram that accesses a variable that isn't visible from the subprogram declaration because it's declared inside some package private part or body. In such a case, you must give a name to the variable through an abstract state declaration. This name can then be used to refer to the variable from within `Global` contracts (but not from within regular code or assertions). More precisely, an abstract state can be declared for the hidden state of a package using an `Abstract_State` aspect (or the equivalent pragma):

```
package P with
   Abstract_State => State
is
   V : Integer;  --  V is visible in P so cannot be part of State

   procedure Update_All with
     Global => (Output => (V, State));
   --  The Global contract mentions V explicitly but uses State to
   --  refer to H and B.

private
   H : Integer with  --  H is hidden in P, it must be part of State
     Part_Of => State;
end P;

package body P with
   Refined_State => (State => (H, B))
is
   B : Integer; --  B is hidden in P, it must be part of State

   procedure Update_All is
   begin
      V := 0;
      H := 0;
      B := 0;
   end Update_All;
end P;
```

An `Abstract_State` annotation is not required, though it may be necessary to annotate some subprograms with `Global` contracts. However, when it's provided, it must be correct and complete: it must list precisely all the hidden variable declared in the package. Several abstract states can be defined for the same package to allow more precise `Global` contracts on subprograms accessing only subsets of the package's hidden variables:

```
package P with
   Abstract_State => (State1, State2)
is
   procedure Update_Only_H with
     Global => (Output => (State1));
   --  If only one abstract state was defined for B and H, the Global
   --  contract would be less precise.

private
   H : Integer with
     Part_Of => State1;
end P;

package body P with
   Refined_State => (State1 => H, State2 => B)
is
   B : Integer := 0;

   procedure Update_Only_H is
   begin
      H := 0;
   end Update_Only_H;
end P;
```

When you provide an abstract state, you must refine it into its constituents in the package body using the `Refined_State` aspect or pragma. Furthermore, to be able to analyze the package specification independently, every private variable must be linked to an abstract state using the `Part_Of` aspect. You can find information about state abstraction in the SPARK User's Guide: http://docs.adacore.com/spark2014-docs/html/ug/en/source/package_contracts.html#state-abstraction.

CHAPTER

FIVE

# SILVER LEVEL - ABSENCE OF RUN-TIME ERRORS (AORTE)

The goal of this level is ensuring that the program does not raise an exception at run time. Among other things, this ensures that the control flow of the program cannot be circumvented by exploiting a buffer overflow, possibly as a consequence of an integer overflow. This also ensures that the program cannot crash or behave erratically when compiled without support for run-time exceptions (compiler switch `-gnatp`) because of operation that would have triggered a run-time exception.

GNATprove can be used to prove the complete absence of possible run-time errors corresponding to all possible explicit raising of exceptions in the program, raising exception `Constraint_Error` at run time, and all possible failures of assertions (corresponding to raising exception `Assert_Error` at run time).

A special kind of run-time errors that can be proved at this level is the absence of exceptions from defensive code. This requires users to add subprogram preconditions (see section *Preconditions* for details) that correspond to the conditions checked in defensive code. For example, defensive code that checks the range of inputs will translate into preconditions of the form `Input_X in Low_Bound .. High_Bound`. These conditions are then checked by GNATprove at each call.

## Benefits

The SPARK code is guaranteed to be free from run-time errors (Absence of Run Time Errors - AoRTE) plus all the defects already detected at Bronze level: no reads of uninitialized variables, no possible interference between parameters and/or global variables, and no unintended access to global variables. Thus, the quality of the program can be guaranteed to achieve higher levels of integrity than would be possible in another programming language.

All the messages about possible run-time errors can be carefully reviewed and justified (for example by relying on external system constraints such as the maximum time between resets) and these justifications can be later reviewed as part of quality inspections.

The proof of AoRTE can be used to compile the final executable without run-time exceptions (compiler switch `-gnatp`), which allows having a very efficient code comparable to what can be achieved in C or assembly.

The proof of AoRTE can be used to comply with the objectives of certification standards in various domains (DO-178 in avionics, EN 50128 in railway, IEC 61508 in many safety related industries, ECSS-Q-ST-80C in space, IEC 60880 in nuclear, IEC 62304 in medical, ISO 26262 in automotive). To date, the use of SPARK has been qualified in EN 50128 context. Qualification material for DO-178 contexts should be available in 2018. Qualification material in any context can be developed by AdaCore as part of a contract.

## Impact on Process

An initial pass is required where proof of AoRTE is applied to the program and the resulting messages are resolved by either rewriting code or justifying any false alarms. Once this is complete, like for the Bronze level, ongoing maintenance can maintain the same guarantees at reasonable cost. Using precise types and simple subprogram contracts

(preconditions and postconditions) is sufficient to avoid most false alarms and any remaining false alarms can be easily justified.

Special treatment is required for loops, which may need the addition of loop invariants to prove AoRTE inside and after the loop. The detailed process for adding them is described in the SPARK User's Guide, as well as examples of common patterns of loops and their corresponding loop invariants.

### Costs and Limitations

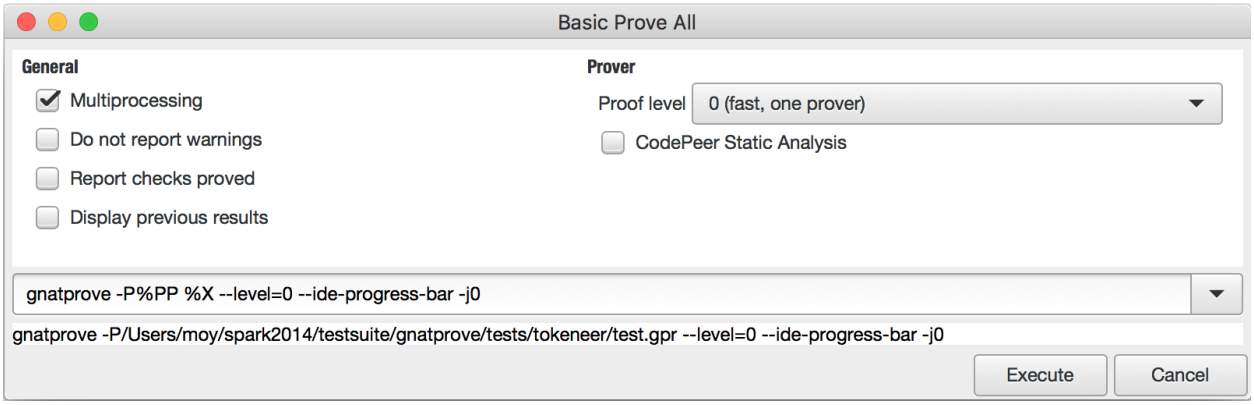
The initial pass may require a substantial effort to get rid of all false alarms, depending on the coding style adopted previously. The analysis may take a long time, up to a few hours, on large programs but is guaranteed to terminate. Proof is, by construction, limited to local understanding of the code, which requires using sufficiently precise types of variables, and some preconditions and postconditions on subprograms to communicate relevant properties to their callers.

Even if a property is provable, automatic provers may nevertheless not be able to prove it, due to limitations of the heuristic techniques used in automatic provers. In practice, these limitations are mostly visible on non-linear integer arithmetic (such as division and modulo) and floating-point arithmetic.

## 5.1 Running GNATprove in Proof Mode

Proof is the second static analysis performed by GNATprove, after the flow analysis seen at Bronze level. Unlike flow analysis, proof may take more or less time to run, depending on the selected proof level. The higher the proof level, the more precise the results and the longer the analysis.

Launch GNATprove in proof mode on your project by selecting the *SPARK → Prove All* menu. In the GPS panel, select *0* as the value of *Proof level*, check the *Multiprocessing* box, uncheck the *Report checks proved* box, and click *Execute*. The following snapshot shows the popup window from GPS with these settings:




GNATprove should output the following messages, possibly followed by a number of messages pointing to potential problems in your program:

```
Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ..
```

The following messages output by GNATprove are check messages and should have the form:

```
medium: overflow check might fail
```

Similarly to the messages previously described, the severity of the check is shown first. It is one of *low*, *medium*, or *high* and reflects both the likelihood of the reported problem being a bug and the criticality of the bug, if it exists.

Following the colon is the type of the check message, here a potential arithmetic overflow. Each message is located in your code at the point where the error can occur and the corresponding line in GPS editor is highlighted in red.

GNATprove can issue several kinds of check messages. In this document, we concentrate on the five most common: division by zero, array index, arithmetic overflow, value in range, and correct discriminant. They are described in section *Run-time Checks*. Other specific check messages can also be issued when tagged types or tasking constructs are used. You can find more information about these additional checks in the SPARK User's Guide: http://docs.adacore.com/spark2014-docs/html/ug/en/source/how_to_view_gnatprove_output.html#description-of-messages.

Proving AoRTE requires interacting with GNATprove inside GPS to either fix the code, add annotations, succeed in proving the check, or to justify the innocuity of the message. This process is explained in section *Investigating Unproved Run-time Checks*.

Once each unproved check message has been addressed in some way, you can run proof mode again with the box *Report checks proved* checked to see the verifications successfully performed by GNATprove. It should only issue 'info' messages, highlighted in green in GPS, like the following:

```
info: overflow check proved
```

## 5.2 Run-time Checks

### divide by zero

This checks that the second operand of a division, mod or rem operation is not equal to zero. It's applicable to all integer and real types for division and to all integer types for mod and rem. Here's an example of such checks:

```
type Oper is (D, M, R);
type Unsigned is mod 2**32;
Small : constant := 1.0 / 2.0**7;
type Fixed is delta Small range -1.0 .. 1.0 - Small
  with Size => 8;

procedure Oper_Integer (Op : Oper; X, Y : Integer; U : out Integer) is
begin
   case Op is
      when D => U := X / Y;     --<<-- medium: divide by zero might fail
      when M => U := X mod Y;   --<<-- medium: divide by zero might fail
      when R => U := X rem Y;   --<<-- medium: divide by zero might fail
   end case;
end Oper_Integer;

procedure Oper_Unsigned (Op : Oper; X, Y : Unsigned; U : out Unsigned) is
begin
   case Op is
      when D => U := X / Y;     --<<-- medium: divide by zero might fail
      when M => U := X mod Y;   --<<-- medium: divide by zero might fail
      when R => U := X rem Y;   --<<-- medium: divide by zero might fail
   end case;
end Oper_Unsigned;

procedure Div_Float (X, Y : Float; U : out Float) is
begin
   U := X / Y;  --<<-- medium: divide by zero might fail
end Div_Float;

procedure Div_Fixed (X, Y : Fixed; U : out Fixed) is
```

```
begin
   U := X / Y;  --<<-- medium: divide by zero might fail
end Div_Fixed;
```

A special case of possible division by zero is the exponentiation of a float value of 0.0 by a negative exponent since the result of this operation is defined as the inverse of the exponentiation of its argument (hence 0.0) by the absolute value of the exponent. Here's an example of such checks:

```
procedure Exp_Float (X : Float; Y : Integer; U : out Float) is
begin
   U := X ** Y;  --<<-- medium: divide by zero might fail
end Exp_Float;
```

### index check

This checks that a given index used to access an array is within the bounds of the array. This applies to both reads and writes to an array. Here's an example of such checks:

```
function Get (S : String; J : Positive) return Character is
begin
   return S(J);  --<<-- medium: array index check might fail
end Get;

procedure Set (S : in out String; J : Positive; C : Character) is
begin
   S(J) := C;  --<<-- medium: array index check might fail
end Set;
```

### overflow check

This checks that the result of a given arithmetic operation is within the bounds of its base type, which corresponds to the bounds of the underlying machine type. It's applicable to all signed integer types (but not modular integer types) and real types, for most arithmetic operations (unary negation, absolute value, addition, subtraction, multiplication, division, exponential). Here's an example of such checks:

```
type Oper is (Minus, AbsVal, Add, Sub, Mult, Div, Exp);
type Unsigned is mod 2**32;
Small : constant := 1.0 / 2.0**7;
type Fixed is delta Small range -1.0 .. 1.0 - Small
  with Size => 8;

procedure Oper_Integer (Op : Oper; X, Y : Integer; E : Natural; U : out Integer) is
begin
   case Op is
      when Minus  => U := -X;  --<<-- medium: overflow check might fail
      when AbsVal => U := abs X;  --<<-- medium: overflow check might fail
      when Add    => U := X + Y;  --<<-- medium: overflow check might fail
      when Sub    => U := X - Y;  --<<-- medium: overflow check might fail
      when Mult   => U := X * Y;  --<<-- medium: overflow check might fail
      when Div    => U := X / Y;  --<<-- medium: overflow check might fail
      when Exp    => U := X ** E;  --<<-- medium: overflow check might fail
   end case;
end Oper_Integer;
```

```
procedure Oper_Float (Op : Oper; X, Y : Float; E : Natural; U : out Float) is
begin
   case Op is
      when Minus  => U := -X;
      when AbsVal => U := abs X;
      when Add    => U := X + Y;  --<<-- medium: overflow check might fail
      when Sub    => U := X - Y;  --<<-- medium: overflow check might fail
      when Mult   => U := X * Y;  --<<-- medium: overflow check might fail
      when Div    => U := X / Y;  --<<-- medium: overflow check might fail
      when Exp    => U := X ** E;  --<<-- medium: overflow check might fail
   end case;
end Oper_Float;

procedure Oper_Fixed (Op : Oper; X, Y : Fixed; E : Natural; U : out Fixed) is
begin
   case Op is
      when Minus  => U := -X;  --<<-- medium: overflow check might fail
      when AbsVal => U := abs X;  --<<-- medium: overflow check might fail
      when Add    => U := X + Y;  --<<-- medium: overflow check might fail
      when Sub    => U := X - Y;  --<<-- medium: overflow check might fail
      when Mult   => U := X * Y;  --<<-- medium: overflow check might fail
      when Div    => U := X / Y;  --<<-- medium: overflow check might fail
      when Exp    => null;
   end case;
end Oper_Fixed;
```

Note that there is no overflow check when negating a floating-point value or taking its absolute value since floating-point base types (32 bits or 64 bits) have symmetric ranges. On the other hand, negating a signed integer or taking its absolute value may result in an overflow if the argument value is the minimal machine integer for this type because signed machine integers are don't have symmetric ranges (they have one less positive value than to negative values). Fixed-point types are based in an machine integer representation, so they can also overflow on negation and absolute value.

### range check

This checks that a given value is within the bounds of its expected scalar subtype. It's applicable to all scalar types, including signed and modulo integers, enumerations and real types. Here's an example of such checks:

```
type Enum is (A, B, C, D, E);
subtype BCD is Enum range B .. D;

type Unsigned is mod 2**32;
subtype Small_Unsigned is Unsigned range 0 .. 10;

Small : constant := 1.0 / 2.0**7;
type Fixed is delta Small range -1.0 .. 1.0 - Small
  with Size => 8;
subtype Natural_Fixed is Fixed range 0.0 .. Fixed'Last;

subtype Natural_Float is Float range 0.0 .. Float'Last;

procedure Convert_Enum (X : Enum; U : out BCD) is
begin
   U := X;  --<<-- medium: range check might fail
end Convert_Enum;
```

```
procedure Convert_Integer (X : Integer; U : out Natural) is
begin
   U := X;  --<<-- medium: range check might fail
end Convert_Integer;

procedure Convert_Unsigned (X : Unsigned; U : out Small_Unsigned) is
begin
   U := X;  --<<-- medium: range check might fail
end Convert_Unsigned;

procedure Convert_Float (X : Float; U : out Natural_Float) is
begin
   U := X;  --<<-- medium: range check might fail
end Convert_Float;

procedure Convert_Fixed (X : Fixed; U : out Natural_Fixed) is
begin
   U := X;  --<<-- medium: range check might fail
end Convert_Fixed;
```

#### discriminant check

This checks that the discriminant of the given discriminated record has the expected value. For variant records, this check is performed for a simple access, either read or write, to a record component. Here's an example of such checks:

```
type Rec (B : Boolean) is record
   case B is
      when True =>
         X : Integer;
      when False =>
         Y : Float;
   end case;
end record;

function Get_X (R : Rec) return Integer is
begin
   return R.X;  --<<-- medium: discriminant check might fail
end Get_X;

procedure Set_X (R : in out Rec; V : Integer) is
begin
   R.X := V;  --<<-- medium: discriminant check might fail
end Set_X;
```

## 5.3 Investigating Unproved Run-time Checks

You should expect many messages about possible run-time errors to be issued the first time you analyze a program, for two main reasons: First, the analysis done by GNATprove relies on the information provided in the program to compute all possible values of variables. This information lies chiefly in the types and contracts added by programmers. If types are not precise enough and/or necessary contracts are not inserted, GNATprove cannot prove AoRTE. Second, the initial analysis performed at proof level 0 is the fastest but also the least precise. Nevertheless, you should start at this level because many checks are not initially provable due to imprecise types and missing contracts. As you add

precise types and contracts to the program, it becomes profitable for you to perform analyses at higher proof levels 1 and 2 to get more precise results.

Proving AoRTE requires interacting with GNATprove inside GPS. Thus, we suggest that you select a unit (preferably one with few dependences over other unproved units, ideally a leaf unit not depending on other unproved units) with some unproved checks. Open GPS on your project, display this unit inside GPS, and put the focus on this unit. Inside this unit, select a subprogram (preferably one with few calls to other unproved subprograms, ideally a leaf subprogram not calling other unproved subprograms) with some unproved checks. This is the first subprogram you will analyze at Silver level.

For each unproved run-time check in this subprogram, you should follow the following steps:

1. Understand why the run-time check can't fail. If you don't understand why a run-time check can never fail, GNATprove can't either. You may discover at this stage that the run-time check can indeed fail, in which case you must first correct the program so that this isn't possible.
2. Determine if the reason(s) that the check always succeeds are known locally. GNATprove analysis is modular, meaning it only looks at locally available information to determine whether a check succeeds or not. This information consists mostly of the types of parameters and global variables, the precondition of the subprogram, and the postconditions of the subprogram it calls. If the information is not locally available, you should change types and/or add contracts to make it locally available to the analysis. See the paragraphs below on 'More Precise Types' and 'Useful Contracts'.
3. If the run-time check depends on the value of a variable being modified in a loop, you may need to add a loop invariant, i.e. a specific annotation in the form of a `pragma Loop_Invariant` inside the loop, which summarizes the effect of the loop on the variable value. See the specific section of the SPARK User's Guide on that topic: http://docs.adacore.com/spark2014-docs/html/ug/en/source/how_to_write_loop_invariants.html.
4. Once you're confident this check should be provable, run SPARK in proof mode on the specific line with the check by right-clicking on the line in the editor panel inside GPS, selecting *SPARK → Prove Line* from the contextual menu, selecting *2* as value for *Proof level* and checking the *Report checks proved* box, both in the GPS panel, and clicking *Execute*. GNATprove should either output a message confirming that the check is proved or the same message as before. In the latter case, you will need to interact with GNATprove to investigate why the check still isn't proved.
5. It may sometimes be difficult to distinguish cases where some information is missing for the provers to prove the check from cases where the provers are incapable of proving the check even with the necessary information. There are multiple actions you can take that may help distinguishing those cases, as documented in a specific section of the SPARK User's Guide on that topic (see subsections on 'Investigating Unprovable Properties' and 'Investigating Prover Shortcomings'): http://docs.adacore.com/spark2014-docs/html/ug/en/source/how_to_investigate_unproved_checks.html. Usually, the best action to narrow down the issue to its core is to insert assertions in the code that test whether the check can be proved at some specific point in the program. For example, if a check message is issued about a possible division by zero on expression `X/Y`, and the implementation contains many branches and paths before this point, try adding assertions that `Y /= 0` in the various branches. This may point to a specific path in the program which causes the issue or it may help provers to manage to prove both the assertion and the check. In such a case, it's good practice to retain in the code only those essential assertions that help produce the automatic proof and to remove other intermediate assertions that you inserted during your interaction with the prover.
6. If the check turns out to be unprovable due to limitations in the proving technology, you will have to justify its presence by inserting a `pragma Annotate` after the line where the check message is reported so that future runs of GNATprove will not report it again . See SPARK User's Guide at http://docs.adacore.com/spark2014-docs/html/ug/en/source/how_to_investigate_unproved_checks.html.

Below we describe how you can change types to be more precise for analysis and how you can add contracts that will make it possible to prove AoRTE.

### More Precise Types

GNATprove's analysis crucially depends on the ranges of scalar types. If the program uses standard scalar types such as `Integer` and `Float`, nothing is known about the range of the data manipulated and result most arithmetic operations will lead to an overflow check message. In particular, data that is used to index arrays or as the right-hand-side of division operations (which includes mod and rem operators) should be known to be respectively in range of the array and not null, generally just by looking at their type.

When standard types such as `Integer` and `Float` are used, you will need to introduce more specific types like `Temperature` or `Length`, with suitable ranges. These may be either new types like:

```
type Temperature is digits 6 range -100.0 .. 300.0;
type Length is range 0 .. 65_535;
```

derived types like:

```
type Temperature is new Float range -100.0 .. 300.0;
type Length is new Integer range 0 .. 65_535;
```

or subtypes like:

```
subtype Temperature is Float range -100.0 .. 300.0;
subtype Length is Integer range 0 .. 65_535;
```

When user types are introduced, you may either add a suitable range to these types or introduce derived types or subtypes with suitable range as above.

### Useful Contracts

Aside from types, it might be important to specify in which context a subprogram may be called. This is known as the precondition of the subprogram. All the examples of check messages seen in section *Run-time Checks* could be proved if suitable preconditions are added to the enclosing subprogram. For example, consider procedure `Convert_Integer`, which assigns an integer `X` to a natural `U`:

```
procedure Convert_Integer (X : Integer; U : out Natural) is
begin
   U := X;  --<<--  medium: range check might fail
end Convert_Integer;
```

In order for GNATprove to prove that the conversion cannot lead to a range check failure, it needs to know that `X` is non-negative when calling `Convert_Integer`, which can be expressed as a precondition as follows:

```
procedure Convert_Integer (X : Integer; U : out Natural)
  with Pre => X >= 0
is
begin
   U := X;
end Convert_Integer;
```

With such a precondition, the range check inside `Convert_Integer` is proved by GNATprove. As a result of inserting preconditions for subprograms, GNATprove checks that the corresponding conditions hold when calling these subprograms. When these conditions cannot be proved, GNATprove issues check messages that need to be handled like run-time check messages. As a result, the same precondition may be pushed up multiple levels of callers to a point where the condition is known to hold.

When a subprogram calls another subprogram, it may also be important to specify what can be guaranteed about the result of that call. For example, consider procedure `Call_Convert_Integer`, which calls the previously seen procedure `Convert_Integer`:

```
procedure Call_Convert_Integer (Y : in out Natural) is
   Z : Natural;
begin
   Convert_Integer (Y, Z);
   Y := Y - Z;  --<<--  medium: range check might fail
end Call_Convert_Integer;
```

When GNATprove analyzes `Call_Convert_Integer`, the only locally available information about the value of `Z` after the call to `Convert_Integer` is its type. This isn't sufficient to guarantee that the subtraction on the following line results in a non-negative result, so GNATprove issues a message about a possible range check failure on this code. In order for GNATprove to prove that the subtraction cannot lead to a range check failure, it needs to know that `Z` is equal to `Y` after calling `Convert_Integer`, which can be expressed as a postcondition as follows:

```
procedure Convert_Integer (X : Integer; U : out Natural)
  with Pre  => X >= 0,
       Post => X = U
is
begin
   U := X;
end Convert_Integer;
```

With such a postcondition, the range check inside `Call_Convert_Integer` is proved by GNATprove. Because of the postconditions added to subprograms, GNATprove checks that the corresponding conditions hold when returning from these subprograms. When these conditions cannot be proved, GNATprove issues check messages that need to be handled similarly to run-time check messages.

CHAPTER

SIX

# GOLD LEVEL - PROOF OF KEY INTEGRITY PROPERTIES

The goal of the Gold level is ensuring key integrity properties such as maintaining critical data invariants throughout execution and ensuring that transitions between states follow a specified safety automaton. Typically, these properties derive from software requirements. Together with the Silver level, these goals ensure program integrity, that is, the program keeps running within safe boundaries: the control flow of the program is correctly programmed and cannot be circumvented through run-time errors and data cannot be corrupted.

SPARK defines a number of useful features used to specify both data invariants and control flow constraints:

- Type predicates state properties that should always be true of any object of the type.
- Preconditions state properties that should always hold on subprogram entry.
- Postcondition state properties that should always hold on subprogram exit.

These features can be verified statically by running GNATprove in prove mode, similarly to what was done at the Silver level. At every point where a violation of the property may occur, GNATprove issues either an 'info' message, verifying that the property always holds, or a 'check' message about a possible violation. Of course, a benefit of proving properties is that they don't need to be tested, which can be used to reduce or completely remove unit testing.

These features can also be used to augment integration testing with dynamic verification that these key integrity properties are satisfied. To enable these additional verifications during execution, you can use either the compilation switch `-gnata` (which enables verification of all invariants and contracts at run time) or `pragma Assertion_Policy` (which enables a subset of the verifications) either inside the code (so that it applies to the code that follows in the current unit) or in a pragma configuration file (so that it applies to the entire program).

## Benefits

The SPARK code is guaranteed to respect key integrity properties as well as being free from all the defects already detected at Bronze and Silver levels: no reads of uninitialized variables, no possible interference between parameters and global variables, no unintended access to global variables, and no run-time errors. This is a unique feature of SPARK that is not found in other programming languages. In particular, such guarantees may be used in a safety case to make reliability claims.

The effort in achieving that level of confidence based on proof is relatively low compared to the effort required to achieve the same level based on testing. Indeed, confidence based on testing has to rely on a nearly comprehensive testing strategy. In fact, certification standards define criteria for approaching comprehensive testing, such as Modified Condition/Decision Coverage (MC/DC), which are notoriously expensive to achieve. Many certification standards allow the use of proof as a replacement for testing, in particular DO-178C in avionics, EN 50128 in railway and IEC 61508 in process and military. Obtaining proofs, as done in SPARK, can thus be used as cost effective alternative to unit testing.

### Impact on Process

In a certification context where proof replaces testing, if independence is required between development and verification activities, subprogram contracts that express software requirements should not be created by the developers implementing such requirements. This is similar to the independence that can be required between the developer and the tester of a module. However, programmers can be expected to write intermediate assertions and to run GNATprove to check that their implementation satisfies the requirements.

Depending on the complexity of the property being proven, it may be more or less costly to add the necessary contracts on types and subprograms and to achieve complete automatic proof by interacting with the tool. This typically requires some experience with the tool that can be developed by training and practice, which suggests that not all developers should be tasked with developing such contracts and proofs, but instead that a few developers should be designated for this task.

As with the proof of AoRTE at Silver level, special treatment is required for loops, which may need the addition of loop invariants to prove properties inside and after the loop. The detailed process for doing so is described in SPARK User's Guide, as well as examples of loops and their corresponding loop invariants.

### Costs and Limitations

The analysis may take a long time, up to a few hours, on large programs, but it is guaranteed to terminate. It may also take more or less time depending on the proof strategy adopted (as indicated by the switches passed to GNATprove). Proof is, by construction, limited to local understanding of the code, which requires using sufficiently precise types of variables and some preconditions and postconditions on subprograms to communicate relevant properties to their callers.

Even if a property is provable, automatic provers may fail to prove it due to limitations of the heuristic techniques used in automatic provers. In practice, these limitations are mostly visible on non-linear integer arithmetic (such as division and modulo) and on floating-point arithmetic.

Some properties may not be easily expressible in the form of data invariants and subprogram contracts, for example properties on execution traces or temporal properties. Other properties may require the use of non-intrusive instrumentation in the form of ghost code.

## 6.1 Type predicates

Type predicates are boolean expressions that constrain the value of objects of a given type. You can attach a type predicate to a scalar type or subtype:

```
type Even is new Integer
  with Predicate => Even mod 2 = 0;
```

In the case above, the use of the type name `Even` in the predicate expression denotes the current object of type `Even`, which we're saying must be even for the expression to evaluate to `True`. Similarly, a type predicate can be attached to an array type or subtype:

```
subtype Simple_String is String
  with Predicate =>
     Simple_String'First = 1 and Simple_String'Last in Natural;

type Sorted is array (1 .. 10) of Integer
  with Predicate => (for all J in 1 .. 9 => Sorted(J) <= Sorted(J+1));
```

`Simple_String` is the same as standard String except that objects of this type always start at index 1 and have a unique representation for the null string, which normally ends at index 0. Type Sorted uses a more complex quantified expression to express that contiguous elements in the array are sorted in increasing order. Finally, a type predicate can also be attached to a record type or subtype:

```
type Name (Size : Positive) is record
   Data : String(1 .. Size);
   Last : Positive;
end record
  with Predicate => Last <= Size;
```

Discriminated record `Name` is a typical example of a variable-sized record, where the internal array `Data` is indexed up to the value of component `Last`. The predicate expresses an essential invariant of objects of type `Name`, namely that `Last` will always be no greater than `Size`. This assures that to `Data(Last)` will be in bounds.

## 6.2 Preconditions

Preconditions are boolean expressions that should be true each time a subprogram is called and are typically used to express API constraints that ensure correct execution of the subprogram. They can thus replace or complement comments and/or defensive code that expresses and/or checks such constraints. Compare the following three styles of expressing that string `Dest` should be at least as long as string `Src` when copying `Src` into `Dest`. The first way is to express the constraint in a comment attached to the subprogram declaration:

```
procedure Copy (Dest : out String; Src : in String);
--  Copy Src into Dest. Require Dest length to be no less than Src length,
--  otherwise an exception is raised.
```

Though readable by humans, this constraint cannot be verified automatically. The second way is to express the constraint is using defensive code inside the subprogram body:

```
procedure Copy (Dest : out String; Src : in String) is
begin
   if Dest'Length < Src'Length then
      raise Constraint_Error;
   end if;
   --  copies Src into Dest here
end Copy;
```

While this constraint can be verified at run time, it's hidden inside the implementation of the subprogram and can't be verified statically with GNATprove. The third way is to express the constraint is as a precondition:

```
procedure Copy (Dest : out String; Src : in String)
  with Pre => Dest'Length >= Src'Length;
--  Copy Src into Dest.
```

This constraint is readable by humans and it can be verified at run time and statically by GNATprove.

## 6.3 Postconditions

Postconditions are boolean expressions that should be true each time a subprogram returns. Postconditions are similar to the normal assertions used by programmers to check properties at run time (with `pragma Assert`), but are more powerful:

1. When a subprogram has multiple returns, it is easy to forget to add a `pragma Assert` before one of the exit points. Postconditions avoid that pitfall.
2. Postconditions can express relations between values of variables at subprogram entry and at subprogram exit, using the attribute `X'Old` to denote the value of variable `X` at subprogram entry.

Postconditions can be used to express major transformations in the program that are performed by some subprograms. For example, data collected from a network may need to be sanitized and then normalized before being fed to the main part of the program. This can be expressed with postconditions:

```
type Status is (Raw, Sanitized, Normalized);
type Chunk is record
   Data : String (1 .. 256);
   Stat : Status;
end record;

procedure Sanitize (C : in out Chunk)
  with Pre  => C.Stat = Raw,
       Post => C.Stat = Sanitized;

procedure Normalize (C : in out Chunk)
  with Pre  => C.Stat = Sanitized,
       Post => C.Stat = Normalized;

procedure Main_Treatment (C : in Chunk)
  with Pre => C.Stat = Normalized;
```

In the code segment above, preconditions and postconditions are used to track the status of the data chunk `C` so that we can guarantee that transformations are performed in the specified order.

## 6.4 Ghost Code

Sometimes, the variables and functions present in a program are insufficient to specify intended properties and to verify these properties with GNATprove. This is the case if the property that should be verified is never used explicitly in the code. For example, the property that a collection is sorted can be maintained for efficient modifications and queries on the collection without the need to have an explicit function `Is_Sorted`. However, this function is essential to state the property that the collection remains sorted.

In such a case, SPARK allows you to insert additional code in the program that's useful for specification and verification, specially identified with the aspect `Ghost` so that it can be discarded during compilation. So-called ghost code in SPARK are these parts of the code that are only meant for specification and verification and have no effect on the functional behavior of the program at run time.

Various kinds of ghost code are useful in different situations: * Ghost functions are typically used to express properties used in contracts. * Global ghost variables are typically used to keep track of the current state

> of a program or maintain a log of past events of some type. This information can then be referred to in contracts.

Typically, the current state of the program may be stored in a global ghost variable, whose value may be suitably constrained in preconditions and postconditions. For example, the program may need to proceed through a number of steps, from sanitization through normalization to main treatment. A ghost variable `Current_State` may then be used to record the current status of the program and its value may be used in contracts as follows:

```
type Status is (Raw, Sanitized, Normalized) with Ghost;
Current_State : Status with Ghost;
```

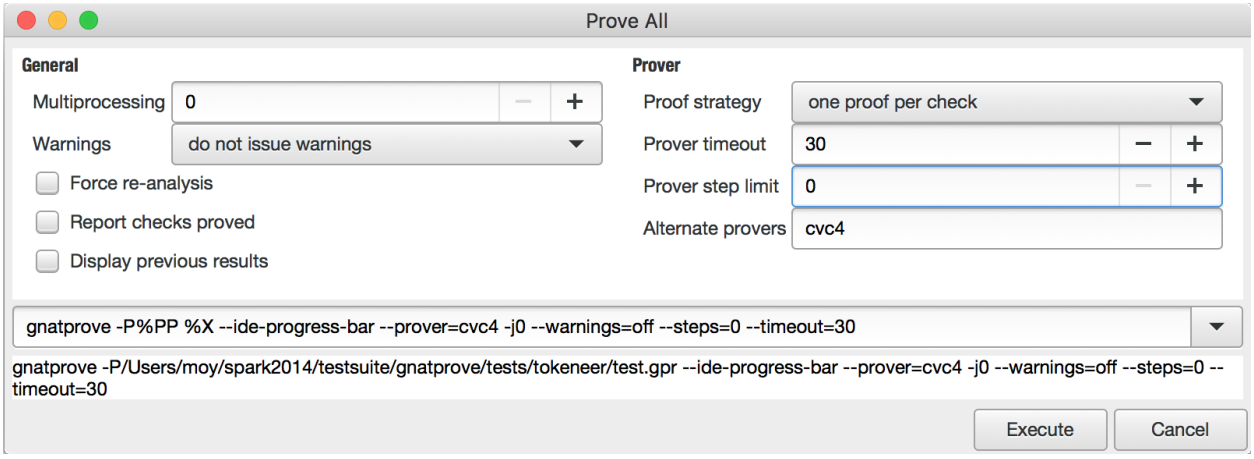
```
procedure Sanitize
  with Pre  => Current_State = Raw,
       Post => Current_State = Sanitized;

procedure Normalize
  with Pre  => Current_State = Sanitized,
       Post => Current_State = Normalized;

procedure Main_Treatment
  with Pre => Current_State = Normalized;
```

See the SPARK User's Guide for more examples of ghost code: http://docs.adacore.com/spark2014-docs/html/ug/en/source/specification_features.html#ghost-code

## 6.5 Investigating Unproved Properties

As seen at Silver level in the section *Investigating Unproved Run-time Checks*, it's expected that many messages about possible violations of properties (assertions, contracts) are issued the first time a program is analyzed, for similar reasons:

1. The analysis done by GNATprove relies on the information provided in the program to compute possible relations between variables. For proving properties, this information lies mostly in the contracts added by programmers. If the contracts are not precise enough, GNATprove cannot prove the desired properties.
2. The initial analysis performed at proof level 0 is the fastest but the least precise. At the Gold level, we advise starting at level 2, so all provers are requested to use reasonable effort (steps). During the interaction with GNATprove, while contracts and assertions are added in the program, it is in general a good idea to perform analysis with only CVC4 enabled (`--prover=cvc4`), no step limit (`--steps=0`) and a higher timeout for individual proof attempts (`--timeout=30`) to get both faster and more precise results. Note that using timeouts instead of steps is not portable between machines, so it's better to reserve it for interactive use. The following snapshot shows the popup window from GPS (using the *Advanced User profile* set through the *Preference* → *SPARK* menu) with these settings:




Proving properties requires interacting with GNATprove inside GPS. Thus, we suggest you select a unit (preferably one with few dependences over other unproved units, ideally a leaf unit not depending on other unproved units) with some unproved checks. Open GPS on your project, display this unit inside GPS, and put the focus on this unit. Inside this unit, select a subprogram (preferably one with few calls to other unproved subprograms, ideally a leaf subprogram not calling other unproved subprograms) with some unproved checks. This is the first subprogram you will analyze at Gold level.

For each unproved property in this subprogram, you should follow the following steps:

1. Understand why the property can't be false at run time. If you don't understand why a property holds, GNATprove can't either. You may discover at this stage that indeed the property may fail at run time, in which case you first need to correct the program so that it can't fail.
2. Determine if the reasons for the property to hold are known locally. GNATprove analysis is modular, which means it only looks at locally available information to determine whether a check succeeds or not. This information consists mostly of the types of parameters and global variables, the precondition of the subprogram, and the postconditions of the subprogram it calls. If the information is not locally available, you should change types and/or add contracts to make it locally available tothe analysis.
3. If the property depends on the value of a variable being modified in a loop, you may need to add a loop invariant, i.e. a specific annotation in the form of a `pragma Loop_Invariant` inside the loop, that summarizes the effect of the loop on the variable value. See the specific section of the SPARK User's Guide on that topic: http://docs.adacore.com/spark2014-docs/html/ug/en/source/how_to_write_loop_invariants.html.
4. Once you are confident this property should be provable, run SPARK in proof mode on the specific line with the check by right-clicking on this line in the editor panel inside GPS, selecting *SPARK* → *Prove Line* from the contextual menu, selecting *2* as value for *Proof level* (and possibly setting the switches `--prover=cvc4 --steps=0 --timeout=30` in the textual box, as described above) and checking the *Report checks proved* box, all in the GPS panel, and clicking *Execute*. GNATprove should either output a message that confirms that the check is proved or the same message as before. In the latter case, you will need to interact with GNATprove to investigate why the check is still not proved.
5. It may sometimes be difficult to distinguish cases where some information is missing for the provers to prove the property from cases where the provers are incapable of proving the property even with the necessary information. The are multiple actions you can take that may help distinguishing those cases, as documented in a specific section of the SPARK User's Guide on that topic (see subsections on 'Investigating Unprovable Properties' and 'Investigating Prover Shortcomings'): http://docs.adacore.com/spark2014-docs/html/ug/en/source/how_to_investigate_unproved_checks.html. Usually, the most useful action to narrow down the issue to its core is to insert assertions in the code that test whether the property (or part of it) can be proved at some specific point in the program. For example, if a postcondition states a property `(P or Q)` and the implementation contains many branches and paths, try adding assertions that `P` holds or `Q` holds where they're expected to hold. This may point to a specific path in the program and/or a specific part of the property which cause the issue. This may also help provers to manage to prove both the assertion and the property. In such a case, it's good practice to retain in the code only those essential assertions that help getting automatic proof and to remove other intermediate assertions that you inserted during the interaction.
6. If the check turns out to be unprovable due to limitations in the proving technology, you have to justify its presence by inserting a `pragma Annotate` after the line where the check message is reported so future runs of GNATprove will not report it again. See SPARK User's Guide at http://docs.adacore.com/spark2014-docs/html/ug/en/source/how_to_investigate_unproved_checks.html.

CHAPTER

SEVEN

# EXAMPLE

As an example, we applied the guidelines in this document to the top-level program in the GNATprove tool itself, called `gnatprove`, which handles the configuration switches, calls other executables to do the analysis and reports messages. We started by manually adding a pragma `SPARK_Mode (On)` to every file of the project. Since `gnatprove` is small (26 units for a total of 4,500 sloc), we didn't need any automation for this step. We then ran GNATprove in check mode. It appeared that no unit of the project was valid SPARK, mostly because of string access types in configuration phase and because of uses of standard container packages for reporting, both of which are not in SPARK.

We chose to concentrate on the `print_table` package, which display the results of a run of GNATprove as a table. It stores the results inside a two dimensional array and then prints them in the `gnatprove.out` file. It's relatively small, but exhibits possible run-time errors, for example when indexing the array or when computing the size required for the table.

## 7.1 Stone Level

We first ran GNATprove in check mode on the unit. We found a non-conformance due to the initializing function for the table having global outputs. Indeed, the unit was designed to construct a unique table, by storing elements line by line from right to left. The current position in the table was stored as a global variable in the package body. As the table array itself was of an unconstrained type (we do not know the number of lines and columns required a priori) it was not stored as a global variable, but carried explicitly as a parameter of the storage procedure. The initialization function both returned a new array, and initialized the value of the current position, thus having global outputs:

```
function Create_Table (Lines, Cols : Natural) return Table is
   T : constant Table (1 .. Lines, 1 .. Cols) :=
     (others => (others => Cell'(Content => Null_Unbounded_String,
                                 Align   => Right_Align)));
begin
   Cur_Line := 1;
   Cur_Col := 1;
   return T;
end Create_Table;
```

To deal with a function with output globals, the guidelines advise either hiding the function body for analysis if the effect does not matter for proof or turning it into a procedure with an 'out' parameter. None of these solutions was adequate here as the effects of this function do matter and the array cannot be given as an 'out' parameter since it's unconstrained. In fact, the non-conformance here comes from a bad implementation choice, which separated the table from its cursors, allowing for unexpected behaviors if two tables were to be initialized. We therefore chose to redesign the code and introduced a record with discriminants to hold both the array and the current position. As this record is of an unconstrained type, it's not stored in a global variable but rather passed explicitly as a parameter as the array used to be.

```
type Table (L, C : Natural) is record
   Content  : Table_Content (1 .. L, 1 .. C);
   Cur_Line : Positive;
   Cur_Col  : Positive;
end record;

function Create_Table (Lines, Cols : Natural) return Table;
```

Other than this non-conformance, GNATprove issued a dozen warnings about assuming no effect of functions from the `Ada.Strings.Unbounded` and `Ada.Text_IO` libraries. This is fine: these functions indeed should have no effects on variables visible to GNATprove. It simply means that issues that may arise when using these libraries are outside of the scope of GNATprove and should be verified in a different way.

We ran GNATprove in check mode on the unit without further errors. We therefore reached the Stone level on this unit.

## 7.2 Bronze Level

Next, we ran GNATprove in flow analysis mode on the unit. No check messages were emitted: we only got a new warning stating that the procedure `Dump_Table`, which writes the value of the table to the `gnatprove.out` file, had no effect. This is expected: functions from the `Ada.Text_IO` library are assumed to have no effect.

As no global variables are accessed by the unit subprograms after the modification outlined earlier, we added global contracts on every subprogram enforcing this:

```
function Create_Table (Lines, Cols : Natural) return Table with
  Global => null;
```

These contracts are all verified by GNATprove. We thus reached Bronze level on this unit.

## 7.3 Silver Level

We then ran GNATprove in prove mode on the unit. 13 check messages were emitted:

- 3 array index checks when accessing at the current position in the table,
- 1 assertion used as defensive coding,
- 2 range checks on Natural, and
- 7 overflow checks when computing the maximal size of the array.

To prove the array index checks, we needed to state that the position is valid in the array when storing a new element. To do this, we put preconditions on the storing procedure:

```
procedure Put_Cell
  (T     : in out Table;
   S     : String;
   Align : Alignment_Type := Right_Align)
with
  Global => null,
  Pre    => T.Cur_Line <= T.L and then T.Cur_Col <= T.C;
```

With this precondition, GNATprove could successfully verify the index checks as well as two overflow checks located at the increment of the position cursors.

The assertion states that the procedure `New_Line`, which moves the current position to the beginning of the next line, can only be called if we're at the end of the current line. We transformed it into a precondition:

```
procedure New_Line (T : in out Table)
with
  Global => null,
  Pre    => T.Cur_Col = T.C + 1 and then T.Cur_Line <= T.L;
```

Avoiding run-time errors in the computation of the maximum size of the table was more complicated. It required bounding both the maximum number of elements in the table and the size of each element. To bound the maximal number of elements in the table, we introduce a constrained subtype of `Positive` for the size of the table, as described in the guidelines:

```
subtype Less_Than_Max_Size is Natural range 0 .. Max_Size;

type Table (L, C : Less_Than_Max_Size) is record
   Content  : Table_Content (1 .. L, 1 .. C);
   Cur_Line : Positive;
   Cur_Col  : Positive;
end record;
```

We couldn't introduce a range for the size of the elements stored in the table, as they aren't scalars but instead unbounded strings. We thus resorted to a predicate to express the constraint:

```
type Cell is record
   Content : Unbounded_String;
   Align   : Alignment_Type;
end record with
  Predicate => Length (Content) <= Max_Size;
```

Next, we needed to add an additional precondition to `Put_Cell` to appropriately constrain the strings that can be stored in the table. With these constraints, as well as some loop invariants to propagate the bound throughout the computation of the maximum size of the table, every check message was discharged except for three, which needed additional information on the subprograms from the unbounded strings library. We introduced an assumption for why the checks could not fail and justified it by quoting the Ada Reference Manual as it is easier to review a single assumption than try to understand what can cause a more complex check to fail.

```
pragma Assume (Length (Null_Unbounded_String) = 0,
               "Null_Unbounded_String represents the null String.");
T.Content (T.Cur_Line, T.Cur_Col) :=
  Cell'(Content => To_Unbounded_String (S),
        Align   => Align);
```

There were no more unproved check messages when running GNATprove in prove mode on `print_table`. We thus reached Silver level on this unit.

## 7.4 Gold Level

The subprograms defined in `Print_Table` are annotated with precise comments describing their effects on the table. As an example, here is the comment associated with `Put_Cell`:

```
procedure Put_Cell
  (T     : in out Table;
   S     : String;
   Align : Alignment_Type := Right_Align);
```

```
--  Print a string into the current cell of the table, and move to the next
--  cell. Note that this does not move to the next line,  you need to call
--  New_Line below after printing to the last cell of a line.
```

We used these comments to derive postconditions on the procedures used to create the table. So that the postcondition of `Put_Cell` is easier to read, we decided to introduce a ghost expression function `Is_Set` that relates the values of the contents of the table before and after the update:

```
function Is_Set
  (T    : Table_Content;
   L, C : Less_Than_Max_Size;
   S    : String;
   A    : Alignment_Type;
   R    : Table_Content) return Boolean
--  Return True if R is S updated at position (L, C) with Cell (S, A)
is
   --  T and R range over the same ranges

  (T'First (1) = R'First (1) and then T'Last (1) = R'Last (1)
   and then T'First (2) = R'First (2) and then T'Last (2) = R'Last (2)

   --  They contain the same values except at position L, C where R
   --  contains S and A.

   and then L in T'Range (1) and then C in T'Range (2)
   and then To_String (R (L, C).Content) = S
   and then R (L, C).Align = A
   and then (for all I in T'Range (1) =>
               (for all J in T'Range (2) =>
                  (if I /= L and J /= C then R (I, J) = T (I, J)))))
with Ghost;
```

Using this function, we can rephrase the comment of `Put_Cell` as a simple postcondition:

```
procedure Put_Cell
  (T     : in out Table;
   S     : String;
   Align : Alignment_Type := Right_Align)
with
  Global => null,
  Pre    => T.Cur_Line <= T.L and then T.Cur_Col <= T.C
            and then S'Length <= Max_Size,

  --  S has been printed inside the current cell with alignment Align

  Post   => Is_Set (T => T.Content'Old,
                    L => T.Cur_Line'Old,
                    C => T.Cur_Col'Old,
                    S => S,
                    A => Align,
                    R => T.Content)

  --  We have moved to the next cell, but not moved to the next line,
  --  even if needed.

            and T.Cur_Line = T.Cur_Line'Old
            and T.Cur_Col = T.Cur_Col'Old + 1;
```

For GNATprove to verify this postcondition, we had to introduce yet another assumption relating the result of `To_String` on an instance of `To_Unbounded_String` to its input:

```
pragma Assume (To_String (To_Unbounded_String (S)) = S,
               String'("If S is a String, then "
                       & "To_String(To_Unbounded_String(S)) = S."));
```

In the same way, we translated the comments provided on every subprogram dealing with the creation of the table. We didn't supply any contract for the subprogram responsible for dumping the table into a file because it's modeled in SPARK as having no effect.

These contracts are all verified by GNATprove. We thus reached Gold level on this unit.

CHAPTER

EIGHT

# REFERENCES

The e-learning website AdaCore University proposes a freely available course on SPARK in five lessons at http://university.adacore.com/courses/spark-2014/

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/

The official book on SPARK is "Building High Integrity Applications with SPARK" by McCormick and Chapin, edited by Cambridge University Press. It is sold online by Amazon and many others.

For a historical account of the evolution of SPARK technology and its use in industry, see the article "Are We There Yet? 20 Years of Industrial Theorem Proving with SPARK" by Chapman and Schanda, at http://proteancode.com/keynote.pdf

The website http://www.spark-2014.org/ is a portal for up-to-date information and resources on SPARK, including an active blog detailing the latest evolutions.

The document "AdaCore Technologies for CENELEC EN 50128:2011" presents the usage of AdaCore's technology in conjunction with the CENELEC EN 50128:2011 standard. It describes in particular where the SPARK technology fits best and how it can best be used to meet various requirements of the standard. See: http://www.adacore.com/knowledge/technical-papers/adacore-technologies-for-cenelec-en-501282011/

A similar document "AdaCore Technologies for DO-178C/ED-12C" will be available soon, presenting the usage of AdaCore's technology in conjunction with the DO-178C/ED-12C standard, and describing in particular the use of SPARK in relation with the Formal Methods supplement DO-333/ED-216.