SPARK Reference Manual

Release 0^20240111gitce5fad0

AdaCore and Capgemini Engineering

Feb 25, 2024

CONTENTS

CHAPTER

ONE

INTRODUCTION

SPARK is a programming language and a set of verification tools designed to meet the needs of high-assurance software development. SPARK is based on Ada, both subsetting the language to remove features that defy verification and also extending the system of contracts by defining new Ada aspects to support modular, constructive, formal verification.

The new aspects support the analysis of incomplete programs, abstraction and refinement and facilitate deep static analysis to be performed including information-flow analysis and formal verification of an implementation against a specification.

Meaningful static analysis is possible on complete programs without the SPARK specific aspects and pragmas (for programs which are otherwise within the SPARK subset), in fact the formal verification of an implementation against a specification of a complete program is possible using only the Ada contracts. Without the SPARK specific aspects, however, analysis has to be performed on a completed program and cannot be applied constructively during its development.

The current version of SPARK, sometimes referred to as SPARK 2014, is a much larger and more flexible language than its predecessor SPARK 2005. The language can be configured to suit a number of application domains and standards, from server-class high-assurance systems to embedded, hard real-time, critical systems.

A major feature of SPARK is the support for a mixture of proof and other verification methods such as testing. This facilitates the use of unit proof in place of unit testing, for example as formalized in avionics certification standard DO-178C and its DO-333 formal methods supplement. Certain units may be formally proven and other units validated through testing.

Ada 2012 introduced executable contracts such as Pre and Post conditions and new types of expression, in particular conditional expressions and quantifiers. SPARK uses these contracts and expressions and extends them with new aspects and pragmas.

The new aspects defined for SPARK all have equivalent pragmas which allows a SPARK program to be compiled by and executed by any Ada implementation; for instance an Ada 95 compiler provided that the use of Ada 2005 and Ada 2012 specific features is avoided. The SPARK attributes Initialized and Loop_Entry can be used only if the Ada implementation supports them.

The direct use of the new aspects requires an Ada 2012 compiler which supports them in a way consistent with the definition given here in the SPARK reference manual. The GNAT implementation is one such compiler.

As with the Ada contracts, the new SPARK aspects and pragmas have executable semantics and may be executed at run time. An expression in an Ada contract or SPARK aspect or pragma is called an *assertion expression* and it is the ability to execute such expressions which facilitates the mix of proof and testing.

The run-time checking of assertion expressions may be suppressed by using the Ada pragma Assertion_Policy but the static analysis and proof tools always use the assertion expressions whatever the assertion policy.

1.1 Structure of Introduction

This introduction contains the following sections:

- Section *How to Read and Interpret this Manual* describes how to read and interpret this document.
- Section *Method of Description* describes the conventions used in presenting the definition of SPARK.
- Section *Formal Analysis* gives a brief overview of the formal analysis to which SPARK programs are amenable.
- Section *Executable Contracts and Mathematical Numbers* gives a brief overview of the use of executable contracts.
- Section *Dynamic Semantics of SPARK Programs* gives details on the dynamic semantics of SPARK.
- Section *SPARK Strategic Requirements* defines the overall goals to be met by the SPARK language and toolset.
- Section *Explaining the Strategic Requirements* provides expanded detail on the main strategic requirements.

1.2 How to Read and Interpret this Manual

This RM (reference manual) is *not* a tutorial guide to SPARK. It is intended as a reference guide for users and implementors of the language. In this context, "implementors" includes those producing both compilers and verification tools.

This manual is written in the style and language of the Ada RM, so knowledge of Ada is assumed. Chapters 2 through 13 mirror the structure of the Ada RM. Chapters 14 onward cover all the annexes of the Ada RM. Moreover, this manual should be interpreted as an extension of the Ada RM (that is, SPARK is fully defined by this document taken together with the Ada RM).

The SPARK RM uses and introduces technical terms in its descriptions, those that are less well known or introduced are summarized in a *Glossary* following the sections covering the Ada annexes.

SPARK introduces a number of aspects. The language rules are written as if all the SPARK specific aspects are present but minimum requirements are placed on a tool which analyzes SPARK to be able to synthesize (from the source code) some of these aspects if they are not present. A tool may synthesize more aspects than the minimum required (see *Synthesis of SPARK Aspects*). An equivalent pragma is available for each of the new aspects but these are not covered explicitly in the language rules either. The pragmas used by SPARK are documented in *Language-Defined Pragmas (Annex L)*.

Readers interested in how SPARK 2005 constructs and idioms map into SPARK should consult the appendix *SPARK 2005 to SPARK 2014 Mapping Specification*.

1.3 Method of Description

In expressing the aspects, pragmas, attributes and rules of SPARK, the following chapters of this document follow the notational conventions of the Ada RM (section 1.1.4).

The following sections are given for each new language feature introduced for SPARK, following the Ada RM (other than *Verification Rules*, which is specific to SPARK):

1. Syntax: this section gives the format of any SPARK specific syntax.
2. Legality Rules: these are rules that are enforced at compile time. A construct is legal if it obeys *all* of the Legality Rules.
3. Static Semantics: a definition of the compile-time effect of each construct.

4. Dynamic Semantics: a definition of the run-time effect of each construct.
5. Verification Rules: these rules define checks to be performed on the language feature that relate to static analysis rather than simple legality rules.
6. Name Resolution Rules: There are very few SPARK specific name resolution rules. Where they exist they are placed under this heading.

A section might not be present if there are no rules specific to SPARK associated with the language feature.

When presenting rules, additional text may be provided in square brackets [ ]. This text is redundant in terms of defining the rules themselves and simply provides explanatory detail.

In addition, examples of the use of the new features are given along with the language definition detail.

1.4 Formal Analysis

SPARK will be amenable to a range of formal analyses, including but not limited to the following static analysis techniques:

- Data-flow analysis, which considers the initialization of variables and the data dependencies of subprograms (which parameters and variables get read or written).
- Information-flow analysis, which also considers the coupling between the inputs and outputs of a subprogram (which input values of parameters and variables influence which output values). The term *flow analysis* is used to mean data-flow analysis and information-flow analysis taken together.
- Formal verification of robustness properties. In Ada terminology, this refers to the proof that certain predefined checks, such as the ones which could raise Constraint_Error, will never fail at run time and hence the corresponding exceptions will not be raised.
- Formal verification of functional properties, based on contracts expressed as preconditions, postconditions, type invariants and so on. The term *formal verification* is used to mean formal verification of robustness properties and formal verification of functional properties taken together.

Data and information-flow analysis is not valid and might not be possible if the legality rules of Ada and those presented in this document are not met. Similarly, a formal verification might not be possible if the legality rules are not met and may be unsound if data-flow errors are present.

1.4.1 Further Details on Formal Verification

Many Ada constructs have dynamic semantics which include a requirement that some error condition must or may[1] be checked, and some exception must or may[1] be raised, if the error is detected (see Ada RM 1.1.5(5-8)). For example, evaluating the name of an array component includes a check that each index value belongs to the corresponding index range of the array (see Ada RM 4.1.1(7)).

For every such run-time check a corresponding obligation to prove that the error condition cannot be true is introduced. In particular, this rule applies to the run-time checks associated with any assertion (see Ada RM (11.4.2)); the one exception to this rule is pragma `Assume` (see *Proof Pragmas*).

In addition, the generation of verification conditions is unaffected by the suppression of checks (e.g., via pragma `Suppress`) or the disabling of assertions (e.g., via pragma `Assertion_Policy`). In other words, suppressing or disabling a check does not prevent generation of its associated verification conditions. Similarly, the verification conditions generated to ensure the absence of numeric overflow for operations of a floating point type T are unaffected by the value of T'Machine_Overflows.

[1] In the case of some bounded errors, performing a check (and raising an exception if the check fails) is permitted but not required.

All such generated verification conditions must be discharged before the formal program verification phase may be considered to be complete.

A SPARK implementation has the option of treating any construct which would otherwise generate an unsatisfiable verification condition as illegal, even if the construct will never be executed. For example, a SPARK implementation might reject the declaration

```
X : Positive := 0;
```

in almost any context. [Roughly speaking, if it can be determined statically that a runtime check associated with some construct will inevitably fail whenever the construct is elaborated, then the implementation is allowed (but not required) to reject the construct just as if the construct violated a legality rule.] For purposes of this rule, the Ada rule that Program_Error is raised if a function "completes normally without executing a return statement" is treated as a check associated with the end of the function body's sequence_of_statements. [This treatment gives SPARK implementations the option of imposing simpler (but more conservative) rules to ensure that the end of a function is not reachable. Strictly speaking, this rule gives SPARK implementations the option of rejecting many things that should not be rejected (e.g., "pragma Assert (False);" in an unreachable arm of a case statement); reasonable implementations will not misuse this freedom.]

Formal verification of a program may depend on properties of either the machine on which it is to be executed or on properties of the tools used to compile and build it. For example, a program might depend on the bounds of the type Standard.Long_Integer or on the implementation-dependent bounds chosen for the unconstrained base subtype associated with a declaration like "type T is range 1 .. 10;". In such cases it must be possible to provide the needed information as explicit inputs to the formal verification process. The means by which this is accomplished is not specified as part of the SPARK language definition.

1.5 Executable Contracts and Mathematical Numbers

Contracts, in the form of assertion expressions, are executable in Ada and SPARK and have the same semantics in both. The new aspects and pragmas introduced by SPARK where they are assertion expressions are also executable. Executable contracts have a number of advantages but also a few drawbacks that SPARK to a large extent mitigates.

The Ada pragma Assertion_Policy controls whether contracts and assertion expressions in general are executed and checked at run-time. Assertion expressions are always significant in static analysis and proof and, indeed, form the basis of the specification against which the implementation is verified.

In summary, Ada in itself enables contract-based, dynamic verification of complex properties of a program. SPARK enables contract-based static deductive verification of a large subset of Ada.

1.5.1 The Advantages of Executable Contracts

The possibility of making assertions and contracts executable benefits the programmer in a number of ways:

- it gives the programmer a gentle introduction to the use of contracts, and encourages the development of assertions and code in parallel. This is natural when both are expressed in the same programming language;
- executable assertions can be enabled and checked at run time, and this gives valuable information to the user. When an assertion fails, it means that the code failed to obey desired properties (i.e., the code is erroneous), or that the intent of the code has been incorrectly expressed (i.e., the assertion is erroneous) and experience shows that both situations arise equally often. In any case, the understanding of the code and properties of the programmer are improved. This also means that users get immediate benefits from writing additional assertions and contracts, which greatly encourages the adoption of contract-based programming;
- contracts can be written and dynamically verified even when the contracts or the program are too complex for automatic proof.

Executable contracts can be less expressive than pure mathematical ones, or more difficult to write in some situations but SPARK has features to largely mitigate these issues as described in the following subsections.

1.5.2 Mathematical Numbers and Arithmetic

In Ada numeric overflow may occur when evaluating an assertion expression this adds to the complexity of writing contracts and specifications using them, for instance, the expression

```
Post => X = (Y + Z) / 100
```

might raise a run-time exception if Y is an integer and Y + Z > Integer'Last even if the entire expression is less then Integer'Last.

SPARK requires checks that have to be proven to demonstrate that an overflow cannot occur, which would not be provable in the above example. Instead, the postcondition would would have to be rewritten, perhaps as something like:

```
Post => X = Integer ((Long_Integer (Y) + Long_Integer (Z)) / 100)
```

In general, the Ada library `Ada.Numerics.Big_Numbers.Big_Integers` can be used so that expressions (at least for Integer types) are treated as mathematical, with no overflow and no exception raised. Using this library, the above example can be rewritten:

```
Post => To_Big_Integer (X) = (To_Big_Integer (Y) + To_Big_Integer (Z)) / 100
```

1.5.3 Libraries for Specification and Proof

It is intended that SPARK toolchains have available libraries (as packages) of common paradigms such as sets, supported by an underlying model of the library packages with an expressive specification that makes automatic proof of (executable) contracts using these libraries practical.

1.6 Dynamic Semantics of SPARK Programs

Every valid SPARK program is also a valid Ada program. However, SPARK makes use of SPARK-defined attributes, aspects, and pragmas which an Ada compiler must process consistently with their SPARK definitions in order to compile and execute a SPARK program as an Ada program; this is possible because Ada permits implementation-defined attributes, aspects, and pragmas. The dynamic semantics of SPARK and of Ada are the same, assuming appropriate Ada support for those SPARK-defined constructs. That one sentence defines the dynamic semantics of SPARK; the only other description of dynamic semantics in the SPARK language definition is in defining these SPARK-defined attributes, aspects, and pragmas.

SPARK programs that have failed their static analysis checks can still be valid Ada programs. An incorrect SPARK program with, say, flow analysis anomalies or undischarged verification conditions can still be executed as long as the Ada compiler in question finds nothing objectionable. What one gives up in this case is the formal analysis of the program, such as proof of absence of run-time errors or the static checks performed by flow analysis such as the proof that all variables are initialized before use.

SPARK may make use of certain aspects, attributes and pragmas which are not defined in the Ada reference manual. Ada explicitly permits implementations to provide implementation-defined aspects, attributes and pragmas. If a SPARK program uses one of these aspects (e.g., Global), or attributes (e.g., Initialized) then it can only be compiled and executed by an implementation which supports the construct in a way consistent with the definition given here in the SPARK reference manual.

If the equivalent pragmas are used instead of the implementation-defined aspects and if the use of implementation-defined attributes is avoided, then a SPARK program may be compiled and executed by any Ada implementation (whether or not it recognizes the SPARK pragmas). Ada specifies that unrecognized pragmas are ignored: an Ada compiler that ignores the pragma is correctly implementing the dynamic semantics of SPARK and the SPARK tools will still be able to undertake all their static checks and proofs. If an Ada compiler defines a pragma with the same name as a SPARK specific pragma but has different semantics, then the compilation or execution of the program may fail.

1.7 Main Program

There is no aspect or pragma in SPARK indicating that a subprogram is a main program. Instead it is expected that any implementation of SPARK will have its own mechanism to allow the tools to identify the main program (albeit not within the language itself).

1.8 SPARK Strategic Requirements

The following requirements give the principal goals to be met by SPARK. Some are expanded in subsequent sections within this chapter.

- The SPARK language subset shall embody the largest subset of Ada to which it is currently practical to apply automatic formal verification, in line with the goals below. However, future advances in verification research and computing power may allow for expansion of the language and the forms of verification available. See section *Principal Language Restrictions* for further details.
- The use of Ada preconditions, postconditions and other assertions dictates that SPARK shall have executable semantics for assertion expressions. Such expressions may be executed, proven or both. See section *Executable Contracts and Mathematical Numbers* for further details.
- SPARK shall provide for mixing of verification evidence generated by formal analysis [for code written in the SPARK subset] and evidence generated by testing or other traditional means [for code written outside of the core SPARK language, including legacy Ada code, or code written in the SPARK subset for which verification evidence could not be generated]. See section *Combining Formal Verification and Testing* for further details. Note, however, that a core goal of is to provide a language expressive enough for the whole of a program to be written in SPARK, making it potentially entirely provable largely using automatic proof tools.
- SPARK shall support *constructive*, modular development which allows contracts to be specified on the declaration of program units and allows analysis and verification to be performed based on these contracts as early as possible in the development lifecycle, even before the units are implemented. As units are implemented the implementation is verified against its specification given in its contract. The contracts are specified using SPARK specific aspects.
- A SPARK analysis tool is required to synthesize at least some of the SPARK specific aspects, used to specify the contract of a program unit, if a contract is not explicitly specified, for instance the *Global Aspects* and the *Depends Aspects* from the implementation of the unit if it exists. The minimum requirements are given in *Synthesis of SPARK Aspects* but a particular tool may provide more precise synthesis and the synthesis of more aspects. The synthesized aspect is used in the analysis of the unit if the aspect is not explicitly specified. The synthesis of SPARK specific aspects facilitates different development strategies and the analysis of pre-existing code (see section *Synthesis of SPARK Aspects*).
- Although a goal of SPARK is to provide a language that supports as many Ada features as practical, there is another goal which is to support good programming practice guidelines and coding standards applicable to certain domains or standards. This goal is met either by standard Ada Restrictions and Profile pragmas, or via existing tools (e.g., pragma Restriction_Warnings in GNAT, or the coding standard checker GNATcheck).

- SPARK shall allow the mixing of code written in the SPARK subset with code written in full Ada. See section *In and Out of SPARK* for further details.
- Many systems are not written in a single programming language. SPARK shall support the development, analysis and verification of programs which are only partly in SPARK, with other parts in another language, for instance, C. SPARK specific aspects manually specified at unit level will form the boundary interface between the SPARK and other parts of the program.
- SPARK shall support entities which do not affect the functionality of a program but may be used in the test and verification of a program. See section *Adding Code for Specification and Verification*.
- SPARK shall support the analysis of external communication channels, which are typically implemented using volatile variables. See section *Volatile State* for further details.
- The language shall offer an unambiguous semantics. In Ada terminology, this means that all erroneous and unspecified behavior shall be eliminated either by direct exclusion or by adding rules which indirectly guarantee that some implementation-dependent choice, other than the fundamental data types and constants, cannot effect the externally-visible behavior of the program. For example, Ada does not specify the order in which actual parameters are evaluated as part of a subprogram call. As a result of the SPARK rules which prevent the evaluation of an expression from having side effects, two implementations might choose different parameter evaluation orders for a given call but this difference won't have any observable effect. [This means undefined, implementation-defined and partially-specified features may be outside of SPARK by definition, though their use could be allowed and a warning or error generated for the user. See section *In and Out of SPARK* for further details.] Where the possibility of ambiguity still exists it is noted, namely the reading of an invalid value from an external source and the use of Unchecked_Conversion, otherwise there are no known ambiguities in the language presented in this document.
- SPARK shall support provision of "formal analysis" as defined by the DO-333 formal methods supplement of the avionics certification standard DO-178C, which states "an analysis method can only be regarded as formal analysis if its determination of a property is sound. Sound analysis means that the method never asserts a property to be true when it is not true." A language with unambiguous semantics is required to achieve this and additionally any other language feature that for which sound analysis is difficult or impractical will be eliminated or its use constrained to meet this goal. See section *Principal Language Restrictions* for further details.

1.9 Explaining the Strategic Requirements

The following sections provide expanded detail on the main strategic requirements.

1.9.1 Principal Language Restrictions

To facilitate formal analyses and verification, SPARK enforces a number of global restrictions to Ada. While these are covered in more detail in the remaining chapters of this document, the most notable restrictions are:

- Restrictions on the use of access types and values, similar in some ways to the ownership model of the programming language Rust.
- All expressions (including function calls) are free of side effects.
- Aliasing of names is not permitted in general but the renaming of entities is permitted as there is a static relationship between the two names. In analysis all names introduced by a renaming declaration are replaced by the name of the renamed entity. This replacement is applied recursively when there are multiple renames of an entity.
- Backward goto statements are not permitted.
- The use of controlled types is not currently permitted.

- Tasks and protected objects are permitted only if the Ravenscar profile (or the Jorvik profile) is specified.
- Raising and handling of exceptions is not currently permitted (exceptions can be included in a program but proof must be used to show that they cannot be raised).

1.9.2 Combining Formal Verification and Testing

There are common reasons for combining formal verification on some part of a codebase and testing on the rest of the codebase:

1. Formal verification is only applicable to a part of the codebase. For example, it might not be possible to apply the necessary formal verification to Ada code that is not in SPARK.
2. Formal verification only gives strong enough results on a part of the codebase. This might be because the desired properties cannot be expressed formally, or because proof of these desired properties cannot be sufficiently automated.
3. Formal verification might be only cost-effective on a part of the codebase. (And it may be more cost-effective than testing on this part of the codebase.)

Since the combination of formal verification and testing cannot guarantee the same level of assurance as when formal verification alone is used, the goal when combining formal verification and testing is to reach a level of confidence at least as good as the level reached by testing alone.

Mixing of formal verification and testing requires consideration of at least the following three issues.

Demarcating the Boundary between Formally Verified and Tested Code

Contracts on subprograms provide a natural boundary for this combination. If a subprogram is proved to respect its contract, it should be possible to call it from a tested subprogram. Conversely, formal verification of a subprogram (including absence of run-time errors and contract checking) depends on called subprograms respecting their own contracts, whether these are verified by formal verification or testing.

In cases where the code to be tested is not SPARK, then additional information may be provided in the code – possibly at the boundary – to indicate this (see section *In and Out of SPARK* for further details).

Checks to be Performed at the Boundary

When a tested subprogram T calls a proved subprogram P, then the precondition of P must hold. Assurance that this is true is generated by executing the assertion that P’s precondition holds during the testing of T.

Similarly, when a proved subprogram P calls a tested subprogram T, formal verification will have shown that the precondition of T holds. Hence, testing of T must show that the postcondition of T holds by executing the corresponding assertion. This is a necessary but not necessarily sufficient condition. Dynamically, there is no check that the subprogram has not updated entities not included in the postcondition.

In general, formal verification works by imposing requirements on the callers of proved code, and these requirements should be shown to hold even when formal verification and testing are combined. Any tool set that proposes a combination of formal verification and testing for SPARK should provide a detailed process for doing so, including any necessary additional testing of proof assumptions.

Conditions that Apply to the Tested Code

The unit of test and formal verification is a subprogram (the sequence of statements of a package body is regarded as a subprogram). There are several sources of conditions that apply to a tested subprogram:

- The need to validate a partial proof of a subprogram that calls a subprogram that is not itself proven but is only tested.
- The need to validate the assumptions on which a proof of a subprogram is based when a tested subprogram calls it.
- A tested subprogram may be flow analyzed if it is in SPARK even if it is not formally proven.
- A tested subprogram may have properties that are formally proven.

Flow analysis of a non-proven subprogram

If a subprogram is in SPARK but is too complex or difficult to prove formally then it still may be flow analyzed which is a fast and efficient process. Flow analysis in the absence of proof has a number of significant benefits as the subprogram implementation is

- checked that it is in SPARK;
- checked that there are no uses of uninitialized variables;
- checked that there are no ineffective statements; and
- checked against its specified Global and Depends aspects if they exist or alternatively facilitating their synthesis. This is important because this automatically checks one of the conditions on tested subprograms which are called from proven code (see *Conditions on a tested subprogram which is called from a partially proven subprogram*).

Proving properties of a tested subprogram

A tested subprogram which is in SPARK may have properties, such as the absence of run-time exceptions proven even though the full functionality of the subprogram is tested rather than proven. The extent to which proof is performed is controlled using pragma Assume (see *Proof Pragmas*).

To perform proof of absence of run-time exceptions but not the postcondition of a subprogram a pragma Assume stating the postcondition is placed immediately prior to each exit point from the subprogram (each return statement or the end of the body). Parts of the postcondition may be proved using a similar scheme.

If the proof of absence of one or more run-time exceptions is not proven automatically or takes too long to prove then pragma Assume may be used to suppress the proof of a particular check.

Pragma Assume informs the proof system that the assumed expression is always True and so the prover does not attempt to prove it. In general pragma Assume should be used with caution but it acts as a pragma Assert when the subprogram code is run. Therefore, in a subprogram that is tested it acts as an extra test.

Conditions on a tested subprogram which is called from a partially proven subprogram

When a subprogram which is to be partially proven calls a tested (but not proven subprogram) then the following conditions must be met by the called subprogram:

- if it is in SPARK then it should be flow analyzed to demonstrate that the implementation satisfies the Global aspect and Depends aspects pf the subprogram if they are given, otherwise conservative approximations will be synthesized from the implementation of the subprogram;
- if it is not in SPARK then at least a Global aspect shall be specified for the subprogram. The Global aspect must truthfully represent the global variables and state abstractions known to the SPARK program (not just the calling subprogram) and specify whether each of the global items are an Input, an Output or is In_Out. The onus is on the user to show that the Global (and Depends) aspect is correct as the SPARK tools do not check this because the subprogram is not in SPARK;
- it shall not update any variable or state abstraction known to the SPARK program, directly or indirectly, apart from through an actual parameter of the subprogram or a global item listed in its Global aspect. Updating a variable or state abstraction through an object of an access type or through a subprogram call is an indirect update. Here again, if the subprogram is not in SPARK and cannot be flow analyzed, the onus is on the user to show this condition is met; and
- if it has a postcondition sufficient testing to demonstrate to a high-level of confidence that the postcondition is always True must be performed.

A tool set may provide further tools to demonstrate that the Global aspects are satisfied by a non-SPARK subprogram and possibly partially check the postcondition.

Conditions on a tested subprogram which is calls a proven subprogram

A tested (but not proven) subprogram which calls a proven subprogram must satisfy the following conditions:

- if it is in SPARK then flow analysis of the tested subprogram should be performed. This demonstrates that all variables and state abstractions which are inputs to the called subprogram are initialized and that the outputs of the called subprogram are used;
- if it is not in SPARK the user must ensure that all variables and state abstractions that are inputs to the called subprogram are initialized prior to calling the subprogram. This is the responsibility of the user as the SPARK tools cannot check this as the subprogram is not in SPARK; and
- if it is in SPARK it may be possible to prove that the precondition of the called subprogram is always satisfied even if no other proof is undertaken, otherwise sufficient testing must be performed by the user to demonstrate to a high-level of confidence that the precondition of the subprogram will always be True when the subprogram is called. The proof of the called subprogram relies on its precondition evaluating to True.

1.9.3 Adding Code for Specification and Verification

Often extra entities, such as types, variables and functions may be required only for test and verification purposes. Such entities are termed *ghost* entities and their use is restricted so that they do not affect the functionality of the program. Complete removal of *ghost* entities has no functional impact on the program.

SPARK supports ghost subprograms, types, objects, and packages. Ghost subprograms may be executable or non-executable. Non-executable ghost subprograms have no implementation and can be used for the purposes of formal verification only. Such functions may have their specification defined within an external proof tool to facilitate formal verification. This specification is outside of the SPARK language and toolset and therefore cannot be checked by the tools. An incorrect definition of function may lead to an unsound proof which is of no use. Ideally any definition will be checked for soundness by the external proof tools.

If the postcondition of a function, F, can be specified in SPARK as F'Result = E, then the postcondition may be recast as the expression of an `expression_function_declaration` as shown below:

```
function F (V : T) return T1 is (E);
```

The default postcondition of an expression function is F'Result = E making E both the implementation and the expression defining the postcondition of the function. This is useful, particularly for ghost functions, as the expression which acts as the postcondition might not give the most efficient implementation but if the function is a ghost function this might not matter.

1.9.4 Synthesis of SPARK Aspects

SPARK supports a *constructive analysis* style where all program units require contracts specified by SPARK specific aspects to be provided on their declarations. Under this constructive analysis style, these contracts have to be designed and added at an early stage to assist modular analysis and verification, and then maintained by the user as a program evolves. When the body of a unit is implemented (or modified) it is checked that it conforms to its contract. However, it is mandated that a SPARK analysis tool shall be able to synthesize a conservative approximation of at least a minimum of SPARK specific aspects from the source code of a unit.

Synthesis of SPARK aspects is fundamental to the analysis of pre-existing code where no SPARK specific aspects are provided.

A SPARK analysis tool is required to be capable of synthesizing at least a basic, conservative *Global Aspects*, *Depends Aspects*, *Refined_Global Aspects*, *Refined_Depends Aspects*, *Abstract_State Aspects*, *Refined_State Aspects*, *Initializes Aspects* and *Default Initial Conditions* from either the implementation code or from other SPARK aspects as follows:

- if a subprogram has no Depends aspect but has a Global aspect, an approximation of the Depends aspect is obtained by constructing a `dependency_relation` by assuming that each output is dependent on every input, where outputs are all of the parameters of mode out and in-out, plus all the `global_items` that have a `mode_selector` of Output or In_Out, and inputs are all the parameters of mode in and in-out, plus all the `global_items` that have a `mode_selector` of Input or In_Out. This is a conservative approximation;
- if a subprogram has a Depends aspect but no Global aspect then the Global aspect is determined by taking each `input` of the `dependency_relation` which is not also an `output` and adding this to the Global aspect with a `mode_selector` of Input. Each `output` of the `dependency_relation` which is not also an `input` is added to the Global aspect with a `mode_selector` of Output. Finally, any other `input` and `output` of the `dependency_relation` which has not been added to the Global aspect is added with a `mode_selector` of In_Out;
- if neither a Global or Depends aspect is present, then first the globals of a subprogram are determined from an analysis of the entire program code. This is achieved in some tool dependent way. The globals of each subprogram determined from this analysis is used to synthesize the Global aspects and then from these the Depends aspects are synthesized as described above;
- if an Abstract_State is specified on a package and a Refined_State aspect is specified in its body, then Refined_Global and Refined_Depends aspects shall be synthesized in the same way as described above. From the Refined_Global, Refined_Depends and Refined_State aspects the abstract Global and Depends shall be synthesized if they are not present.
- if no abstract state aspect is specified on a package but it contains hidden state, then each variable that makes up the hidden state has a Abstract_State synthesized to represent it. At least a crude approximation of a single state abstraction for every variable shall be provided. A Refined_State aspect shall be synthesized which shows the constituents of each state.
- if no Default_Initial_Condition is specified for a private type declaration, then the synthesized value of this aspect of the type is determined by whether the full view of the private type defines full default initialization (see SPARK

RM 3.1). If it does, then the synthesized aspect value is a static *Boolean*_`expression` having the value True; if it does not, then the synthesized aspect value is a null literal.

The syntheses described above do not include all of the SPARK aspects and nor do the syntheses cover all facets of the aspects. In complex programs where extra or more precise aspects are required they might have to be specified manually.

An analysis tool may provide the synthesis of more aspects and more precise synthesis of the mandatory ones.

Some use cases where the synthesis of aspects is likely to be required are:

- Code has been developed as SPARK but not all the aspects are included on all subprograms by the developer. This is regarded as *generative analysis*, where the code was written with the intention that it would be analyzed.
- Code is in maintenance phase, it might or might not have all of the SPARK specific aspects. If there are aspects missing they are automatically for analysis purposes when possible. This is also regarded as generative analysis.
- Legacy code is analyzed which has no or incomplete SPARK specific aspects This is regarded as *retrospective analysis*, where code is being analyzed that was not originally written with analysis in mind. Legacy code will typically have a mix of SPARK and non-SPARK code (and so there is an interaction with the detail presented in section *In and Out of SPARK*). This leads to two additional process steps that might be necessary:
  - An automatic identification of what code is in SPARK and what is not.
  - Manual definition of the boundary between the SPARK and non-SPARK code by explicitly specifying accurate and truthful contracts using SPARK specific aspects on the declarations of non-SPARK program units.

1.9.5 In and Out of SPARK

There are various reasons why it may be necessary to combine SPARK and non-SPARK in the same program, such as (though not limited to):

- Use of language features that are not amenable to formal verification (and hence where formal verification will be mixed with testing).
- Use of libraries that are not written in SPARK.
- Need to analyze legacy code that was not developed as SPARK.

Hence, it must be possible within the language to indicate what parts are (intended to be) in and what parts are (intended to be) out, of SPARK.

The default is to assume none of the program text is in SPARK, although this can be overridden. A new aspect *SPARK_Mode* is provided, which may be applied to a unit declaration or a unit body, to indicate when a unit declaration or just its body is in SPARK and should be analyzed. If just the body is not in SPARK a SPARK compatible contract may be supplied on the declaration which facilitates the analysis of units which use the declaration. The tools cannot check that the the given contract is met by the body as it is not analyzed. The burden falls on the user to ensure that the contract represents the behavior of the body as seen by the SPARK parts of the program and – if this is not the case – the assumptions on which the analysis of the SPARK code relies may be invalidated.

In general a definition may be in SPARK but its completion need not be.

A finer grain of mixing SPARK and Ada code is also possible by justifying certain warnings and errors. Warnings may be justified at a project, library unit, unit, and individual warning level. Errors may be justifiable at the individual error level or be unsuppressible errors.

Examples of this are:

- A declaration occurring immediately within a unit might not be in, or might depend on features not in, the SPARK subset. The declaration might generate a warning or an error which may be justifiable. This does not necessarily render the whole of the program unit not in SPARK. If the declaration generates a warning, or if the error is justified, then the unit is considered to be in SPARK except for the errant declaration.

- It is the use of the entity declared by the errant declaration, for instance a call of a subprogram or the denoting of an object in an expression (generally within the statements of a body) that will result in an unsuppressible error. The body of a unit causing the unsuppressible message (or declaration if this is the cause) will need to be marked as not in SPARK to prevent its future analysis.

Hence, SPARK and non-SPARK code may mix at a fine level of granularity. The following combinations may be typical:

- Package (or generic package) specification in SPARK. Package body entirely not in SPARK.
- Visible part of package (or generic package) specification in SPARK. Private part and body not in SPARK.
- Package specification in SPARK. Package body almost entirely in SPARK, with a small number of subprogram bodies not in SPARK.
- Package specification in SPARK, with all bodies imported from another language.
- Package specification contains a mixture of declarations which are in SPARK and not in SPARK. A client of the package may be in SPARK if it only references SPARK declarations; the presence of non-SPARK constructs in a referenced package specification does not by itself mean that a client is not in SPARK.

Such patterns are intended to allow for mixed-language programming, mixed-verification using different analysis tools, and mixed-verification between formal verification and more traditional testing. A condition for safely combining the results of formal verification with other verification results is that formal verification tools explicitly list the assumptions that were made to produce their results. The proof of a property may depend on the assumption of other user-specified properties (for example, preconditions and postconditions) or implicit assumptions associated with the foundation and hypothesis on which the formal verification relies (for example, initialization of inputs and outputs, or non-aliasing between parameters). When a complete program is formally verified, these assumptions are discharged by the proof tools, based on the global guarantees provided by the strict adherence to a given language subset. No such guarantees are available when only part of a program is formally verified. Thus, combining these results with other verification results depends on the verification of global and local assumptions made during formal verification.

Full details on the SPARK_Mode aspect are given in the SPARK Toolset User's Guide (*Identifying SPARK Code*).

1.9.6 Volatile State

A variable or a state abstraction may be specified as external state to indicate that it represents an external communication channel, for instance, to a device or another subsystem. An external variable may be specified as volatile. A volatile state need not have the same value between two reads without an intervening update. Similarly an update of a volatile variable might not have any effect on the internal operation of a program, its only effects are external to the program. These properties require special treatment of volatile variables during flow analysis and formal verification.

SPARK follows the Ada convention that a read of a volatile variable may have an external effect as well as reading the value of the variable. SPARK extends this notion to cover updates of a volatile variable such that an update of a volatile variable may also have some other observable effect. SPARK further extends these principles to apply to state abstractions (see section *External State*).

CHAPTER

TWO

LEXICAL ELEMENTS

SPARK supports the full Ada language with respect to lexical elements. Users may choose to apply restrictions to simplify the use of wide character sets and strings.

2.1 Character Set

No extensions or restrictions.

2.2 Lexical Elements, Separators, and Delimiters

No extensions or restrictions.

2.3 Identifiers

No extensions or restrictions.

2.4 Numeric Literals

No extensions or restrictions.

2.5 Character Literals

No extensions or restrictions.

2.6 String Literals

No extensions or restrictions.

2.7 Comments

No extensions or restrictions.

2.8 Pragmas

SPARK introduces a number of new pragmas that facilitate program verification. These are described in the relevant sections of this document.

2.9 Reserved Words

No extensions or restrictions.

CHAPTER

THREE

DECLARATIONS AND TYPES

No extensions or restrictions.

3.1 Declarations

The view of an entity is in SPARK if and only if the corresponding declaration is in SPARK. When clear from the context, we say *entity* instead of using the more formal term *view of an entity*. If the initial declaration of an entity (e.g., a subprogram, a private type, or a deferred constant) requires a completion, it is possible that the initial declaration might be in SPARK (and therefore can be referenced in SPARK code) even if the completion is not in SPARK. [This distinction between views is much less important in "pure" SPARK than in the case where SPARK_Mode is used (as described in the SPARK Toolset User's Guide) to allow mixing of SPARK and non-SPARK code.]

A type is said to *define full default initialization* if it is

- a scalar type with a specified Default_Value; or
- an access type; or
- an array-of-scalar type with a specified Default_Component_Value; or
- an array type whose element type defines default initialization; or
- a record type, type extension, or protected type each of whose `component_declarations` either includes a `default_expression` or has a type which defines full default initialization and, in the case of a type extension, is an extension of a type which defines full default initialization; or
- a task type; or
- a private type whose Default_Initial_Condition aspect is specified to be a *Boolean*`_expression` and whose full view is not in SPARK; or
- a private type whose full view is in SPARK and defines full default initialization.

[The discriminants of a discriminated type play no role in determining whether the type defines full default initialization.]

3.2 Types and Subtypes

No extensions or restrictions.

3.2.1 Type Declarations

No extensions or restrictions.

3.2.2 Subtype Declarations

A `constraint` in SPARK cannot be defined using variable expressions except when it is the `range` of a `loop_parameter_specification`. Dynamic subtypes are permitted but they must be defined using constants whose values may be derived from expressions containing variables. Note that a formal parameter of a subprogram of mode **in** is a constant and may be used in defining a constraint. This restriction gives an explicit constant which can be referenced in analysis and proof.

An *expression with a variable input* reads a variable or calls a function which (directly or indirectly) reads a variable.

Legality Rules

1. [A `constraint`, excluding the `range` of a `loop_parameter_specification`, shall not be defined using an expression with a variable input; see *Expressions* for the statement of this rule.]

3.2.3 Classification of Operations

No restrictions or extensions.

3.2.4 Subtype Predicates

Static predicates and dynamic predicates are both in SPARK, but subject to some restrictions. A predicate might be introduced by the Ada aspects Static_Predicate and Dynamic_Predicate, or by the SPARK aspects Predicate and Ghost_Predicate.

A predicate introduced by aspects Predicate or Ghost_Predicate is regarded as static if it has an allowed form for Static_Predicate and is otherwise treated as a Dynamic_Predicate.

A predicate introduced by aspect Ghost_Predicate can reference a ghost entity (see section *Ghost Entities*), even if the subtype is not ghost itself. But the subtype cannot appear as a subtype_mark in a membership test. [As predicates participate in membership tests, a membership test may implicitly reference ghost entities in that case.]

Legality Rules

1. [A Dynamic_Predicate expression shall not have a variable input; see *Expressions* for the statement of this rule.]
2. If a Dynamic_Predicate applies to the subtype of a composite object, then a verification condition is generated to ensure that the object satisfies its predicate immediately after any subcomponent or slice of the given object is either
   - the target of an assignment statement or;
   - an actual parameter of mode **out** or **in out** in a call.

   [These verification conditions do not correspond to any run-time check. Roughly speaking, if object X is of subtype S, then verification conditions are generated as if an implicitly generated

   pragma Assert (X in S);

were present immediately after any assignment statement or call which updates a subcomponent (or slice) of X.]

[No such proof obligations are generated for assignments to subcomponents of the result object of an aggregate, an extension aggregate, or a delta aggregate. These are assignment operations but not assignment statements.]

3. A Static_Predicate or Dynamic_Predicate shall not apply to a subtype of a type that is effectively volatile for reading.

Verification Rules

4. A Dynamic_Predicate expression shall always terminate.

3.3 Objects and Named Numbers

3.3.1 Object Declarations

The Boolean aspect Constant_After_Elaboration may be specified as part of the declaration of a library-level variable. If the aspect is directly specified, the aspect_definition, if any, shall be a static [Boolean] expression. [As with most Boolean-valued aspects,] the aspect defaults to False if unspecified and to True if it is specified without an aspect_definition.

A variable whose Constant_After_Elaboration aspect is True, or any part thereof, is said to be *constant after elaboration*. [The Constant_After_Elaboration aspect indicates that the variable will not be modified after execution of the main subprogram begins (see section *Tasks and Synchronization*).]

A stand-alone constant is said to be *immutable* if it is not of an access-to-variable type. [Note that this is not exactly the same definition as for immutable parameters (see section *Anti-Aliasing*).]

Otherwise, the stand-alone constant is said to be *mutable*.

A stand-alone immutable constant is a *constant with variable inputs* if its initialization expression depends on:

- A variable or parameter; or
- Another *constant with variable inputs*

Otherwise, a stand-alone immutable constant is a *constant without variable inputs*.

Legality Rules

1. [The borrowed name of the expression of an object declaration defining a borrowing operation shall not have a variable input, except for a single occurrence of the root object of the expression; see *Expressions* for the statement of this rule.]

Verification Rules

2. Constants without variable inputs shall not be denoted in Global, Depends, Initializes or Refined_State aspect specifications. [Two elaborations of such a constant declaration will always yield equal initialization expression values.]

Examples

```
A : constant Integer := 12;
--  No variable inputs

B : constant Integer := F (12, A);
--  No variable inputs if and only if F is a function without global inputs
```

(continues on next page)

(continued from previous page)

```
--  (although it could have global proof inputs)

C : constant Integer := Param + Var;
--  Constant with variable inputs
```

3.3.2 Number Declarations

No extensions or restrictions.

3.4 Derived Types and Classes

The following rules apply to derived types in SPARK.

Legality Rules

1. A private type that is not visibly tagged but whose full view is tagged cannot be derived.

[The rationale for this rule is that, otherwise, given that visible operations on this type cannot have class-wide preconditions and postconditions, it is impossible to check the verification rules associated to overriding operations on the derived type.]

3.5 Scalar Types

The Ada RM states that, in the case of a fixed point type declaration, “The base range of the type does not necessarily include the specified bounds themselves”. A fixed point type for which this inclusion does not hold is not in SPARK.

For example, given

```
type T is delta 1.0 range -(2.0 ** 31) .. (2.0 ** 31);
```

it might be the case that (2.0 ** 31) is greater than T’Base’Last. If this is the case, then the type T is not in SPARK.

[This rule applies even in the case where the bounds specified in the `real_range_specification` of an `ordinary_fixed_point_definition` define a null range.]

3.5.1 Real types

Non-static expressions of type *root_real* are not supported [because the accuracy of their run-time evaluation depends on the implementation].

3.6 Array Types

No extensions or restrictions.

3.7 Discriminants

The following rules apply to discriminants in SPARK.

Legality Rules

1. The type of a `discriminant_specification` shall be discrete.
2. A `discriminant_specification` shall not occur as part of a derived type declaration.
3. [The `default_expression` of a `discriminant_specification` shall not have a variable input; see *Expressions* for the statement of this rule.]

3.8 Record Types

Default initialization expressions must not have variable inputs in SPARK.

Legality Rules

1. [The `default_expression` of a `component_declaration` shall not have any variable inputs, nor shall it contain a name denoting the current instance of the enclosing type; see *Expressions* for the statement of this rule.]

[The rule in this section applies to any `component_declaration`; this includes the case of a `component_declaration` which is a `protected_element_declaration`. In other words, this rule also applies to components of a protected type.]

3.9 Tagged Types and Type Extensions

Legality Rules

1. No construct shall introduce a semantic dependence on the Ada language defined package Ada.Tags. [See Ada RM 10.1.1 for the definition of semantic dependence. This rule implies, among other things, that any use of the Tag attribute is not in SPARK.]
2. The identifier External_Tag shall not be used as an `attribute_designator`.

3.9.1 Type Extensions

Legality Rules

1. A type extension shall not be declared within a subprogram body, block statement, or generic body which does not also enclose the declaration of each of its ancestor types.

3.9.2 Dispatching Operations of Tagged Types

No extensions or restrictions.

3.9.3 Abstract Types and Subprograms

No extensions or restrictions.

3.9.4 Interface Types

No extensions or restrictions.

3.10 Access Types

In order to reduce the complexity associated with the specification and verification of a program's behavior in the face of pointer-related aliasing, anonymous access-to-constant types and (named or anonymous) access-to-variable types are subject to an *ownership policy*.

Restrictions are imposed on the use of these access objects in order to ensure, roughly speaking (and using terms that have not been defined yet), that at any given point in a program's execution, there is a unique "owning" reference to any given allocated object. The "owner" of that allocated object is the object containing that "owning" reference. If an object's owner is itself an allocated object then it too has an owner; this chain of ownership will always eventually lead to a (single) nonallocated object.

Ownership of an allocated object may change over time (e.g., if an allocated object is removed from one list and then appended onto another) but at any given time the object has only one owner. Similarly, at any given time there is only one access path (i.e., the name of a "declared" (as opposed to allocated) object followed by a sequence of component selections, array indexings, and access value dereferences) which yields a given (non-null) access value. At least that's the general idea - this paragraph oversimplifies some things (e.g., see "borrowing" and "observing" below - these operations extend SPARK's existing "single writer, multiple reader" treatment of concurrency and of aliasing to apply to allocated objects), but hopefully it provides useful intuition.

This means that data structures which depend on having multiple outstanding references to a given object cannot be expressed in the usual way. For example, a doubly-linked list (unlike a singly-linked list) typically requires being able to refer to a list element both from its predecessor element and from its successor element; that would violate the "single owner" rule. Such data structures can still be expressed in SPARK (e.g., by storing access values in an array and then using array indices instead of access values), but they may be harder to reason about.

The single-owner model statically prevents storage leaks because a storage leak requires either an object with no outstanding pointers to it or an "orphaned" cyclic data structure (i.e., a set of multiple allocated objects each reachable from any other but with no references to any of those objects from any object outside of the set).

For purposes of flow analysis (e.g., Global and Depends aspect specifications), a read or write of some part of an allocated object is treated like a read or write of the owner of that allocated object. For example, an assignment to Some_Standalone_Variable.Some_Component.all is treated like an assignment to Some_Standalone_Variable.Some_Component. Similarly, there is no explicit mention of anything related to access types in a Refined_State or Initializes aspect specification; allocated objects are treated like components of their owners and, like components, they are not mentioned in these contexts. This approach has the benefit that the same SPARK language rules which prevent unsafe concurrent access to non-allocated variables also provide the same safeguards for allocated objects.

The rules which accomplish all of this are described below.

Static Semantics

Only the following (named or anonymous) access types are in SPARK:

- a named access-to-object type,
- the anonymous type of a stand-alone object (excluding a generic formal **in** mode object) which is not Part_Of a protected object,
- an anonymous type occurring as a parameter type, or as a function result type of a traversal function (defined below), or
- an access-to-subprogram type associated with the "Ada" or "C" calling convention.

[Redundant: For example, access discriminants and access-to-subprogram types with the "protected" calling convention are not in SPARK.]

User-defined storage pools are not in SPARK; more specifically, the package System.Storage_Pools, Storage_Pool aspect specifications, and the Storage_Pool attribute are not in SPARK.

In the case of a constant object of an access-to-variable type where the object is not a stand-alone object and not a formal parameter (e.g., if the object is a subcomponent of an enclosing object or is designated by an access value), a dereference of the object provides a constant view of the designated object [redundant: , despite the fact that the object is of an access-to-variable type. This is because a subcomponent of a constant is itself a constant and a dereference of a subcomponent is treated, for purposes of analysis, like a subcomponent].

A function is said to be a *traversal function* if the result type of the function is an anonymous access-to-object type and the function has at least one formal parameter. The traversal function is said to be an *observing traversal function* if the result type of the function is an anonymous access-to-constant type, and a *borrowing traversal function* if the result type of the function is an anonymous access-to-variable type. The first parameter of the function is called the *traversed* parameter. [Redundant: We will see later that if a traversal function yields a non-null result, then that result is "reachable" from the traversed parameter in the sense that it could be obtained from the traversed parameter by some sequence of component selections, array indexing operations, and access value dereferences.]

The *root object* of a name that denotes an object is defined as follows:

- if the name is a component_selection, an indexed_component, a slice, or a dereference (implicit or explicit) then it is the root object of the prefix of the name;
- if the name denotes a call on a traversal function, then it is the root object of the name denoting the actual traversed parameter;
- if the name denotes an object renaming, the root object is the root object of the renamed name;
- if the name is a function_call, and the function called is not a traversal function, the root object is the result object of the call;
- if the name is a qualified_expression or a type conversion, the root object is the root object of the operand of the name;
- otherwise, the name statically denotes an object and the root object is the statically denoted object.

A *path* is either:

- a stand-alone object or a formal parameter,
- a component_selection or dereference whose prefix is a path,
- a slice whose discrete range is made of two literals and whose prefix is a path which is not a slice, or
- an indexed_component whose expressions are literals and whose prefix is a path which is not a slice.

The *path extracted from a name* whose root object is a stand-alone object or a formal parameter and which does not contain any traversal function calls is defined as follows:

- if the name is a dereference (implicit or explicit), then it is a dereference of the path extracted from the prefix of the name;

- if the name is a component_selection, then it is a component_selection of the same component on the path extracted from the prefix of the name;
- if the name is an indexed_component, then it is an indexed_component with the literals that each index expression evaluates to, on the path extracted from the prefix of the name, or, if this path is a slice, the prefix of this slice;
- if the name is a slice, then it is a slice whose discrete range is constructed with the literals that the discrete range of the name evaluates to, on the path extracted from the prefix of the name, or, if this path is a slice, the prefix of this slice;
- if the name is a qualified_expression or a type conversion, then it is the path extracted from the path of the expression of the name;
- if the name denotes an object renaming, then it is the path extracted from the renamed name;
- otherwise, the name is a stand-alone object or formal parameter and the path is this object.

If a path P1 has another path P2 as a prefix, then P1 is an *extension* of P2.

Two names are said to be *potential aliases* when their root object is a stand-alone object or a formal parameter, they do not contain any traversal function calls, and either:

- they have the same extracted path,
- the extracted path of one of the names is a slice and the extracted path of the other is an indexed_component whose index is in the discrete range of the slice, or
- the extracted path of one of the names is a slice and the extracted path of the other is another slice and the discrete range of both slices overlap.

Two names N1 and N2 are said to *potentially overlap* if

- some prefix of N1 is a potential alias of N2 (or vice versa); or
- N1 is a call on a traversal function and the actual traversed parameter of the call potentially overlaps N2 (or vice versa).

[Note that for a given name N which denotes an object of an access type, the names N and N.all potentially overlap. Access value dereferencing is treated, for purposes of this definition, like record component selection or array indexing.]

The prefix and the name that are potential aliases are called the *potentially aliased parts* of the potentially overlapping names.

An object O1 is said to be a *reachable part* of an object O2 if:

- O1 is a part of O2; or
- O1 is a reachable part of the object designated by (the value of) an access-valued part of O2.

A path is said to denote a reachable part of an object, if it is the path extracted from a name which denotes this reachable part.

A path can be marked by one of the following *ownership markers* for this object: Persistent, Observed, Borrowed, or Moved. Due to aliasing, there can be several paths denoting a given object, with different associated markers.

A given path cannot have more than one marker at a given program point, but it may have different markers at different points in the program. For example, within a block_statement which declares a borrower (borrowers have not been defined yet), the path extracted from the borrowed name will be marked as Borrowed, while it will have no marks immediately before and immediately after the block_statement. [Redundant: This is a compile-time notion; no mapping of any sort is maintained at runtime.]

When a path P is marked as Observed or Persistent, then all names whose extracted path is an extension of P provide a constant view of their denoted object and its reachable parts (even if the root object is a variable). If P is marked as Persistent, then it will never be possible to modify its denoted object and its reachable parts again in the program, and it is OK to lose track of the owner of its potential access-to-variable parts.

When a path P is marked as Moved, then names whose extracted path is an extension of P cannot be used to read or modify the objected denoted by P or its reachable parts (although names whose extracted path is a strict prefix of P can be assigned to).

When a path P is marked as Borrowed, then names whose extracted path is an extension of P cannot be used to read or modify the objected denoted by P or its reachable parts, and names whose extracted path is a strict prefix of P cannot be assigned to.

A path P is said to have *unrestricted prefixes* if all prefixes of P are unmarked.

A path P is said to be *unrestricted*, if P has unrestricted prefixes and no extensions of P are marked as either Observed, Borrowed, or Moved [A path P can be unrestricted even if there are extensions of P which are marked as Persistent].

A path P said to be *observable*, if no prefixes of P and no extensions P are marked as either Borrowed or Moved.

The ownership rules presented in this section ensure that:

- [single-ownership] if a given object O is denoted by two distinct paths P1 and P2 at a given program point and P1 is unrestricted, then P2 is not observable.

Together with the fact that:

- [ownership-write] O can only be written from a name with an unrestricted extracted path and
- [ownership-read] O can only be read from a name with an observable extracted path,

these are enough to ensure absence of harmful aliasing.

Unless otherwise specified, all paths are initially unmarked except:

- a root object R is marked as Observed if R is a constant and does not have an access-to-variable type, and
- a dereference is marked as Persistent if its prefix is a path denoting an object of an access-to-constant type.

Certain constructs (described below) are said to *observe*, *borrow*, or *move* a path; these may change the ownership markers (to Observed, Borrowed, or Moved respectively) of a path within a certain portion of the program text (described below). In the first two cases (i.e. observing and borrowing), the ownership marker of the path reverts to its previous value at the end of this region of text. The markers are considered to be reverted after the finalization of the borrower/observer but before the finalization of the root of the borrowed or observed paths if they are declared in the same memory region.

If the root object of a name is a stand-alone object or a formal parameter, then the *known extracted path* of that name is either:

- the path extracted from the name, if it does not include any traversal function calls from the root object,
- the path extracted from the first parameter to the innermost traversal function call within the name otherwise.

[Redundant: The root of the known extracted path of a name is always the root object of the name.]

A *markable expression* is either a name whose root object is a stand-alone object or a formal parameter or a reference to the Access attribute whose prefix is a name whose root object is a stand-alone object or a formal parameter.

By extension, the root object and known extracted path of a markable expression are defined as the root object and known extracted path of the prefix for a reference to the Access attribute and of the name otherwise.

The following operations *observe* a path and identify a corresponding *observer*:

- An assignment operation that is used to initialize an access object, where this target object (the observer) is a stand-alone variable of an anonymous access-to-constant type, or a constant (including a formal parameter of a procedure or generic formal object of mode **in**) of an anonymous access-to-constant type.

  The source expression of the assignment shall be a markable expression. The known extracted path of the source of the assignment is observed by the assignment.

- Inside the body of a borrowing traversal function, an assignment operation that is used to initialize an access object, where this target object (the observer) is a stand-alone object of an anonymous access-to-variable type, and the source expression of the assignment is a markable expression whose root object is either the traversed parameter for the traversal function or another object of an access-to-variable type initialized as an observer. The known extracted path of the source of the assignment is observed by the assignment.

Such an operation is called an *observing operation*.

In the region of program text between the point where a path is observed and the end of the scope of the observer, the path is marked as Observed.

The following operations *borrow* a path and identify a corresponding *borrower*:

- An assignment operation that is used to initialize an access object, where this target object (the borrower) is a stand-alone variable or constant of an anonymous access-to-variable type, unless this assignment is already an *observing operation* inside the body of a borrowing traversal function, per the rules defining *observe* above.

  The source expression of the assignment shall be a markable expression. The known extracted path of the source of the assignment is borrowed by the assignment.

Such an operation is called a *borrowing operation*.

In the region of program text between the point where a path is borrowed and the end of the scope of the borrower, the path is marked as Borrowed.

An indirect borrower of a path is defined to be a borrower either of a borrower of the path or of an indirect borrower of the path. A direct borrower of a markable part is just another term for a borrower of the path, usually used together with the term "indirect borrower". The terms "indirect observer" and "direct observer" are defined analogously.

The following operations are said to be *move* operations:

- An assignment operation, where the target is a variable, a constant, or return object (see Ada RM 6.5) of a type containing subcomponents of a named access-to-variable type. [This includes the case of an object of named access-to-variable type.]

[Redundant: Passing a parameter by reference is not a move operation.]

A move operation results in a transfer of ownership. The state of the paths that are marked as Moved by the operation remain in this state until the object is assigned another value.

[Redundant: Roughly speaking, any access-valued parts of an object in the Moved state can be thought of as being "poisoned"; such a poisoned object is treated analogously to an uninitialized object in the sense that various rules statically prevent the reading of such a value. Thus, an assignment like:

```
Pointer_1 : Some_Access_Type := new Designated_Type'(...);
Pointer_2 : Some_Access_Type := Pointer_1;
```

does not violate the "single owner" rule because the move operation poisons Pointer_1, leaving Pointer_2 as the unique owner of the allocated object. Any attempt to read such a poisoned value is detected and rejected.

Note that a name may be "poisoned" even if its value is "obviously" null. For example, given:

```
X : Linked_List_Node := (Data => 123, Link => null);
Y : Linked_List_Node := X;
```

X.Link is poisoned by the assignment to Y.]

Legality Rules

1. At the point of a move operation, the source shall be a name which does not involve any traversal function calls from the root object or a reference to the Access attribute whose prefix is a name which does not involve any traversal function calls from the root object. In addition, if the source is a markable expression, the known

extracted path P of the source shall be unrestricted. If the source is a markable expression which is not a reference to the Access attribute, for all extensions Q of P with no additional dereferences designating objects of a named access-to-variable type, Q.all is marked as Moved after the move operation. If the source is a markable expression which is a reference to the Access attribute, the known extracted path of it prefix is marked as Moved after the move operation.

2. A name which is used as an actual parameter of an anonymous access-to-object type shall either be syntactically null, or shall have a root object which is either a stand-alone object or a formal parameter. In addition, if the parameter type is an access-to-variable type and the name is not syntactically null, it shall not involve any traversal function calls from its root object and the path extracted from the name shall be unrestricted.

3. A name whose type has subcomponents of a [named] access-to-variable type which is used as the target of an assignment or as an actual parameter of mode **out** or **in out** shall have a root object which is either a stand-alone object or a formal parameter, and it shall not involve any traversal function calls from this root object. In addition, if P is the path extracted from a name used as the target of an assignment operation or as an actual parameter of mode **out** in a call,

   - P shall have unrestricted prefixes,
   - there shall be no extension of P marked as Borrowed or Observed, and
   - all extensions of P marked as Moved shall contain additional dereferences.

   All paths with the target as a root are reset to their initial value after the operation.

   [Redundant: In the case of a call, the mark of an actual parameter of mode **in** or **in out** remains unchanged (although one might choose to think of it as being moved at the point of the call and then moved back when the call returns - either model yields the same results); an actual parameter of mode **out** becomes unrestricted.]

4. If the target of an assignment operation is an object of an anonymous access-to-object type (including copy-in for a parameter), then the source shall be a markable expression.

   [Redundant: One consequence of this rule is that every allocator is of a named access type.]

5. A declaration of a stand-alone object of an anonymous access type shall have an explicit initial value and shall occur immediately within a subprogram body, an entry body, or a block statement.

   [Redundant: Because such declarations cannot occur immediately within a package declaration or body, the associated borrowing/observing operation is limited by the scope of the subprogram, entry or block statement. Thus, it is not necessary to add rules restricting the visibility of such declarations.]

6. A return statement that applies to a traversal function that has an anonymous access-to-constant (respectively, access-to-variable) result type, shall return either the literal null or a markable expression whose root object is a direct or indirect observer (respectively, borrower) of the traversed parameter. [Redundant: Roughly speaking, a traversal function always yields either null or a result which is reachable from the traversed parameter.]

7. If a name whose type has subcomponents of a named access-to-variable type is a non-traversal function call or an allocator, it shall only occur in an acceptable context, namely:

   - As the initial expression of an object declaration which does not occur in a declare expression,
   - As the source of an assignment,
   - As the return value of a return statement,
   - As the expression of a type conversion or qualified expression itself occurring in an acceptable context,
   - As an aggregate itself occurring in an acceptable context, or
   - Anywhere inside a contract or an assertion. [While legal, such an expression inside a contract or assertion will leak memory. A verification rule below forbids leaking memory, leading to a violation on such uses. The intent is to allow the use of allocators and allocating functions inside contracts and assertions, but

make sure that users are aware of the possible memory leaks if such contracts and assertions are executed at runtime.]

8. For an assignment statement where the target is a stand-alone object of an anonymous access-to-object type, the source shall be a markable expression whose root object is the target object itself. In addition:

   - If the type of the target is an anonymous access-to-constant type or if the target is a local object of a borrowing traversal function whose initialization is an observing operation, the known extracted path of the source shall be observable for the target object;
   - If the type of the target is an anonymous access-to-variable type, which does not fall in the case above, then the target object shall be unrestricted.

9. At the point of a read of an object, or of passing an object as an actual parameter of mode **in** or **in out**, or of a call where the object is a global input of the callee, if the object is a markable expression, then its known extracted path shall be observable.

10. At the point of a return or a raise statement, or at any other point where a call completes normally or propagates an exception (e.g., the end of a procedure body), there shall be no paths marked as Moved with any inputs or outputs of the callee being returned from as a root. In the case of an input of the callee which is not also an output, this rule may be enforced at the point of the move operation (because there is no way for the Moved marker to be removed from the input), even in the case of a subprogram which never returns.

    Similarly, at the end of the elaboration of both the declaration and of the body of a package, there shall be no paths marked as Moved whose root is denoted by the name of an initialization_item of the package's Initializes aspect or by an input occuring in the input_list of such an initialization_item.

    At the end of the scope of an object of an anonymous access-to-variable type, or at any other point where the scope of an object of an anonymous access-to-variable type is exited normally, there shall be no paths marked as Moved with the object as a root.

11. For a borrowing operation, the borrowed path shall be unrestricted.

12. At the point of a call, no paths with any global output of the callee (i.e., an output other than a parameter of the callee or a function result) as a root shall be marked as Borrowed or Observed, and all such paths which are marked as Moved shall contain dereferences.

13. The prefix of an Old or Loop_Entry attribute reference shall not be of an anonymous access-to-object type nor of a type with subcomponents of a named access-to-variable type unless the prefix is a call to a non-traversal function.

14. A derived tagged type shall not have a component of a named access-to-variable type.

15. If the designated type of a named nonderived access type is incomplete at the point of the access type's declaration then the incomplete type declaration and its completion shall occur in the same declaration list. [This implies that the incomplete type shall not be declared in the limited view of a package, and that if it is declared in the private part of a package then its completion shall also occur in that private part.]

16. A path rooted at an effectively volatile object shall not be moved, borrowed, or observed. [This rule is meant to avoid introducing aliases between volatile variables used by another task or thread. Borrowers can also break the invariant on the borrowed object for the time of the borrow.]

17. A path rooted at a non-ghost object shall only be moved, or borrowed, if the target object of the move or borrow is itself non-ghost. [This rule is meant to avoid introducing aliases between a non-ghost variable and a ghost variable. Otherwise writes or deallocation through the ghost variable would have an effect on the non-ghost underlying memory.]

18. Objects of an anonymous access-to-object types shall not be converted (implicitly or explicitly) to a named access type.

19. Evaluation of equality operators, and membership tests where one or more of the choices are expressions, shall not include directly or indirectly calls to the primitive equality on access types, unless one of the operands is syntactically null.

20. Instances of Unchecked_Deallocation shall not have a general access type as a parameter.

Verification Rules

21. When an object R which does not have an anonymous access-to-object type is finalized or when it is passed as an actual parameter of mode **out**, all extensions of the path extracted from R which denote an object of a pool-specific access type and have unrestricted prefixes shall be null.

    Similarly, at the point of a call, for each global output R of the callee (i.e., an output other than a parameter of the callee or a function result) that is not also an input, all paths rooted at R which denote an object of a pool-specific access type and which have unrestricted prefixes shall be null.

    [Redundant: This rule applies to any finalization associated with a call to an instance of Ada.Unchecked_Deallocation. For details, see the Ada RM 13.11.2 rule "Free(X), … first performs finalization of the object designated by X".]

    [Redundant:This rule effectively forbids the use of allocators and calls to allocating functions inside contracts or assertions.]

22. Allocators and conversions from a pool-specific access type to a named access-to-constant type or a general access-to-variable type shall only occur at library level.

    In the same way, a reference to the Access attribute of a named access-to-object type whose prefix contains a dereference of a pool-specific access-type shall occur at library level.

    [Redundant: Together with the previous one, this rule disallows storage leaks. Without these rules, it would be possible to "lose" the last reference to an allocated object.]

23. When converting from a [named or anonymous] access-to-subprogram type to another, if the converted expression is not null, a verification condition is introduced to ensure that the precondition of the source of the conversion is implied by the precondition of the target of the conversion. Similarly, a verification condition is introduced to ensure that the postcondition of the target is implied by the postcondition of the converted access-to-subprogram expression.

3.11 Declarative Parts

No extensions or restrictions.

CHAPTER

FOUR

NAMES AND EXPRESSIONS

The term *assertion expression* denotes an expression that appears inside an assertion, which can be a pragma Assert, a precondition or postcondition, a type invariant or (subtype) predicate, or other assertions introduced in SPARK.

4.1 Names

No extensions or restrictions.

4.1.1 Indexed Components

No extensions or restrictions.

4.1.2 Slices

No extensions or restrictions.

4.1.3 Selected Components

Some constructs which would unconditionally raise an exception at run time in Ada are rejected as illegal in SPARK if this property can be determined prior to formal program verification.

Legality Rules

1. If the prefix of a record component selection is known statically to be constrained so that the selected component is not present, then the component selection (which, in Ada, would raise Constraint_Error if it were to be evaluated) is illegal.

4.1.4 Attributes

Many of the Ada language defined attributes are in SPARK but there are exclusions. For a full list of attributes supported by SPARK see *Language-Defined Attributes*.

A SPARK implementation is permitted to support other attributes which are not Ada or SPARK language defined attributes and these should be documented in the User Guide for the tool.

Legality Rules

1. The prefix of an Access attribute reference shall be the name of a subprogram or a name denoting an object whose root object is either a standalone object or a subprogram parameter (see section *Access Types* for the definition of a the root object of a name denoting an object).

2. A subprogram used as the prefix of an Access attribute reference:

   - shall not be declared within a protected type or object;
   - shall not be a dispatching operation of a tagged type; and
   - shall not be a declared in the scope of a type with an invariant if this type is mentioned in the subprogram's profile unless it is a boundary subprogram (see section *Type Invariants* for the definition of a boundary subprogram).

3. The Volatile_Function aspect of a subprogram used as the prefix of an Access attribute reference, if specified, shall not be True (see section *External State* for the definition of Volatile_Function).

4. The Side_Effects aspect of a subprogram used as the prefix of an Access attribute reference, if specified, shall not be True (see section *Functions With Side Effects* for the definition of Side_Effects).

5. A reference to the Access attribute whose type is an anonymous access-to-object type or a named access-to-variable type shall occur directly inside a stand-alone object declaration, an assignment, or a return statement.

6. The prefix of an Access attribute reference whose type is a named access-to-constant type shall either be a name denoting a part of a stand-alone constant whose type is neither a named access-to-variable type nor an anonymous access-to-object type, or shall include a dereference whose prefix has a named access-to-constant type.

Verification Rules

7. A subprogram used as the prefix of an Access attribute reference shall have no global inputs and outputs (see section *Subprogram Declarations* for inputs and outputs of subprograms).

8. On an Access attribute reference whose prefix is the name of a subprogram, a verification condition is introduced to ensure that the precondition of the prefix of the attribute reference is implied by the precondition of its expected type. Similarly, a verification condition is introduced to ensure that the postcondition of the expected type is implied by the postcondition of the prefix of the attribute reference.

4.1.5 User-Defined References

Legality Rules

1. User-defined references are not allowed.
2. The aspect Implicit_Dereference is not permitted.

4.1.6 User-Defined Indexing

Legality Rules

1. User-defined indexing is not allowed.
2. The aspects Constant_Indexing and Variable_Indexing are not permitted.

4.2 Literals

No extensions or restrictions.

4.3 Aggregates

Legality Rules

1. The box symbol, `<>`, shall not be used in an aggregate unless each of the corresponding components satisfies one the following conditions:
   - the type of the component defines full default initialization, or
   - the type of the component has relaxed initialization (see *Relaxed Initialization*), or
   - the type of one of the enclosing aggregates has relaxed initialization.
2. If the `ancestor_part` of an `extension_aggregate` is a `subtype_mark`, then the type of the denoted subtype shall define full default initialization.

[The box symbol cannot be used in an aggregate to produce an uninitialized scalar value or a composite value having an uninitialized scalar value as a subcomponent. Similarly for an ancestor subtype in an extension aggregate.]

4.3.1 Record Aggregates

No extensions or restrictions.

4.3.2 Extension Aggregates

No extensions or restrictions.

4.3.3 Array Aggregates

No extensions or restrictions.

4.3.4 Delta Aggregates

In SPARK, a delta aggregate may be used to specify new values for subcomponents of the copied base value, instead of only new values for direct components of the copied base value. This allows a more compact expression of updated values with a single delta aggregate, instead of multiple nested delta aggregates.

Thus, the rules applicable to SPARK delta aggregates are the same as the ones applicable to Ada delta aggregates, when considering the expansion of SPARK delta aggregates into nested Ada delta aggregates, except that SPARK delta aggregates could necessitate fewer copies. In particular, we don't repeat here the Name Resolution Rules for Ada delta aggregates.

The syntax of delta aggregates is revised as follows, which extends the syntax of delta_aggregate in Ada.

Syntax

```
delta_aggregate ::= record_delta_aggregate | array_delta_aggregate

record_delta_aggregate ::=
  ( base_expression with delta record_subcomponent_association_list )

record_subcomponent_association_list ::=
  record_subcomponent_association {, record_subcomponent_association}

record_subcomponent_association ::=
  record_subcomponent_choice_list => expression

record_subcomponent_choice_list ::=
  record_subcomponent_choice {'|' record_subcomponent_choice}

record_subcomponent_choice ::=
    component_selector_name
  | record_subcomponent_choice (expression)
  | record_subcomponent_choice . component_selector_name

array_delta_aggregate ::=
    ( base_expression with delta array_component_association_list )
  | '[' base_expression with delta array_component_association_list ']'
  | ( base_expression with delta array_subcomponent_association_list )
  | '[' base_expression with delta array_subcomponent_association_list ']'

array_subcomponent_association_list ::=
  array_subcomponent_association {, array_subcomponent_association}

array_subcomponent_association ::=
  array_subcomponent_choice_list => expression

array_subcomponent_choice_list ::=
  array_subcomponent_choice {'|' array_subcomponent_choice}

array_subcomponent_choice ::=
    ( expression )
  | array_subcomponent_choice (expression)
  | array_subcomponent_choice . component_selector_name
```

Legality Rules

1. For an array_delta_aggregate, the discrete_choice shall not be **others**.
2. For an array_delta_aggregate, the dimensionality of the type of the delta_aggregate shall be 1.
3. For an array_delta_aggregate, the base_expression and each expression in every array_component_association or array_subcomponent_association shall be of a nonlimited type.
4. For a record_delta_aggregate, no record_subcomponent_choices that consists of only component_selector_names shall be the same or a prefix of another record_subcomponent_choice.
5. For an array_subcomponent_choice or a record_subcomponent_choice, the component_selector_name shall not be a subcomponent that depends on discriminants of an unconstrained record subtype with defaulted discriminants unless its prefix consists of only component_selector_names. [Rationale: As a result of this rule, accessing the subcomponent can only lead to a discriminant check failure if the subcomponent was not present in the object denoted by the base_expression, prior to any update.]

Dynamic Semantics

6. The evaluation of a delta_aggregate begins with the evaluation of the base_expression of the delta_aggregate; then that value is used to create and initialize the anonymous object of the aggregate. The bounds of the anonymous object of an array_delta_aggregate and the discriminants (if any) of the anonymous object of a record_delta_aggregate are those of the base_expression. If a record_delta_aggregate is of a specific tagged type, its tag is that of the specific type; if it is of a class-wide type, its tag is that of the base_expression.

7. For a delta_aggregate, for each discrete_choice or each subcomponent associated with a record_subcomponent_associated, array_component_association or array_subcomponent_association (in the order given in the enclosing discrete_choice_list or subcomponent_association_list, respectively):

   - if the associated subcomponent belongs to a variant, a check is made that the values of the governing discriminants are such that the anonymous object has this component. The exception Constraint_Error is raised if this check fails.
   - if the associated subcomponent is a subcomponent of an array, then for each represented index value (in ascending order, if the discrete_choice represents a range):
     - the index value is converted to the index type of the array type.
     - a check is made that the index value belongs to the index range of the corresponding array part of the anonymous object; Constraint_Error is raised if this check fails.
   - the expression of the record_subcomponent_association, array_component_association or array_subcomponent_association is evaluated, converted to the nominal subtype of the associated subcomponent, and assigned to the corresponding subcomponent of the anonymous object.

Examples

```ada
type Point is record
   X, Y : Integer;
end record;

type Segment is array (1 .. 2) of Point;

S : Segment;

S := (S with delta (1).X | (2).Y => S(2).X, (1).Y => S(2).Y);

type Triangle is array (1 .. 3) of Segment;

T : Triangle;

T := (T with delta (2)(1).Y => T(1)(2).X);
```

4.3.5 Container Aggregates

No extensions or restrictions.

4.4 Expressions

An expression is said to be *side-effect free* if the evaluation of the expression does not update any object. The evaluation of an expression free from side effects only retrieves or computes a value.

Legality Rules

1. An expression shall be side-effect free, unless it is a call to a function with side effects (see section *Functions With Side Effects*). [Strictly speaking, this "rule" is a consequence of other rules, most notably the rule that a function without side effects cannot have outputs other than its result, and that calls to function with side effectss are not subexpressions.]

2. An expression (or range) in SPARK occurring in certain contexts (listed below) shall not have a variable input. This means that such an expression shall not read a variable, nor shall it call a function which (directly or indirectly) reads a variable. These contexts include:

   - a constraint other than the range of a loop parameter specification (see *Subtype Declarations*);
   - the default_expression of a component declaration (see *Record Types*);
   - the default_expression of a discriminant_specification (see *Discriminants*);
   - a Dynamic_Predicate aspect specification (see *Subtype Predicates*);
   - a Type_Invariant aspect specification (see *Type Invariants*);
   - the expression of a Priority aspect specification (see *Tasks and Synchronization*);
   - an indexing expression of an indexed_component or the discrete_range of a slice in an object renaming declaration which renames part of that indexed_component or slice, or a prefix of a dereference (either implicit or explicit) in an object renaming declaration which renames part of the designated object (see *Object Renaming Declarations*);
   - a generic actual parameter corresponding to a generic formal object having mode **in** (see *Generic Instantiation*);
   - the borrowed name of the expression of an object declaration defining a borrowing operation, except for a single occurrence of the root object of the expression (see *Access Types*).

except when the context itself occurs within a declare expression. For purposes of the above rule, a generic actual parameter corresponding to a generic formal object of mode **in out** is considered to be an object renaming declaration which renames the named object.

[An expression in one of these contexts may read a constant which is initialized with the value of a variable.]

[These rules simplify analysis by eliminating the need to deal with implicitly created anonymous constants. An expression which does not have a variable input will always yield the same result if it is (conceptually, for purposes of static analysis) reevaluated later. This is not true of an expression that has a variable input because the value of the variable might have changed.]

[For purposes of these rules, the current instance of a type or subtype is not considered to be a variable input in the case of a Dynamic_Predicate or Type_Invariant condition, but is considered to be a variable input in the case of the default_expression of a component declaration.]

4.5 Operators and Expression Evaluation

Ada grants implementations the freedom to reassociate a sequence of predefined operators of the same precedence level even if this changes the behavior of the program with respect to intermediate overflow (see Ada RM 4.5). SPARK assumes that an implementation does not take advantage of this permission; in particular, a proof of the absence of intermediate overflow in this situation may depend on this assumption.

A SPARK tool is permitted to provide a warning where operators may be re-associated by a compiler.

[The GNAT Ada compiler does not take advantage of this permission. The GNAT compiler also provides an option for rejecting constructs to which this permission would apply. Explicit parenthesization can always be used to force a particular association in this situation.]

4.6 Type Conversions

No extensions or restrictions.

4.7 Qualified Expressions

No extensions or restrictions.

4.8 Allocators

A function is said to be an *allocating function* if the result type of the function is a named access-to-variable type or a composite type with subcomponents of a [named] access-to-variable type. [Redundant: The only functions with a result of a type with subcomponents of an access-to-variable type in SPARK are allocating functions and borrowing traversal functions defined in section *Access Types*; a function cannot be both an allocating function and a traversal function.]

Legality Rules

1. The designated type of the type of an uninitialized allocator shall define full default initialization.
2. An allocator or a call to an allocating function shall only occur in an *allocating context*. An expression occurs in an allocating context if it is:
   - the [right-hand side] expression of an assignment statement; or
   - the initialization expression of an object declaration which does not occur inside a declare expression; or
   - the return expression of a `simple_return_statement`; or
   - the expression of the `extended_return_object_declaration` of an `extended_return_statement`; or
   - the expression of a type conversion, a qualified expression or a parenthesized expression occurring in an allocating context; or
   - the expression corresponding to a component value in an aggregate occurring in an allocating context; or
   - the expression of an initialized allocator; or
   - inside an assertion.

[This restriction is meant to prevent storage leaks, together with the rules on access objects, see section *Access Types*. Note that allocators or calls to allocating functions inside assertions are allowed, but should be reported by the analysis tool as leading to a memory leak. In practice, such memory leaks cannot happen if the corresponding assertions are not enabled in the final executable.]

3. The type of an allocator shall not be anonymous.

4.9 Static Expressions and Static Subtypes

No extensions or restrictions.

CHAPTER

FIVE

STATEMENTS

SPARK restricts the use of some statements, and adds a number of pragmas which are used for verification, particularly involving loop statements.

5.1 Simple and Compound Statements - Sequences of Statements

SPARK excludes certain kinds of statements that complicate verification.

Legality Rules

1. A `simple_statement` shall not be a `requeue_statement`, an `abort_statement`, or a `code_statement`.
2. A `compound_statement` shall not be an `accept_statement` or a `select_statement`.
3. A statement is only in SPARK if all the constructs used in the statement are in SPARK.

5.2 Assignment Statements

No extensions or restrictions.

5.3 If Statements

No extensions or restrictions.

5.4 Case Statements

No extensions or restrictions.

5.5 Loop Statements

5.5.1 User-Defined Iterator Types

Legality Rules

1. The generic package Ada.Iterator_Interfaces shall not be referenced. [In particular, Ada.Iterator_Interfaces shall not be instantiated. An alternative mechanism for defining iterator types is described in the next section.]

5.5.2 Generalized Loop Iteration

Static Semantics

1. Ada's generalized loop iteration is supported in SPARK, but only in a modified form. Ada's existing generalized loop iteration is defined in terms of other constructs which are not in SPARK (e.g., access discriminants).

2. Instead, SPARK provides a new mechanism for defining an iterable container type (see Ada RM 5.5.1). Iteration over the elements of an object of such a type is then allowed as for any iterable container type (see Ada RM 5.5.2), although with dynamic semantics as described below. Similarly, SPARK provides a new mechanism for defining an iterator type (see Ada RM 5.5.1), which then allows generalized iterators as for any iterator type (see Ada RM 5.5.2). Other forms of generalized loop iteration are not in SPARK.

3. The type-related operational representation aspect Iterable may be specified for any non-array type. The `aspect_definition` for an Iterable aspect specification for a subtype of a type T shall follow the following grammar for `iterable_specification`:

```
iterable_specification ::=
  (First       => name,
   Next        => name,
   Has_Element => name[,
   Element     => name])
```

4. If the aspect Iterable is visibly specified for a type, the (view of the) type is defined to be an iterator type (view). If the aspect Iterable is visibly specified for a type and the specification includes an Element argument then the (view of the) type is defined to be an iterable container type (view). [The visibility of an aspect specification is defined in Ada RM 8.8]. [Because other iterator types and iterable container types as defined in Ada RM 5.5.1 are necessarily not in SPARK, this effectively replaces, rather than extends, those definitions].

Legality Rules

5. Each of the four (or three, if the optional argument is omitted) names shall denote an explicitly declared primitive function of the type, referred to respectively as the First, Next, Has_Element, and Element functions of the type. All parameters of all four subprograms shall be of mode In.

6. The First function of the type shall take a single parameter, which shall be of type T. The "iteration cursor subtype" of T is defined to be result subtype of the First function. The First function's name shall be resolvable from these rules alone. [This means the iteration cursor subtype of T can be determined without examining the other subprogram names]. The iteration cursor subtype of T shall be definite and shall not be limited.

7. The Next function of the type shall have two parameters, the first of type T and the second of the cursor subtype of T; the result subtype of the function shall be the cursor subtype of T.

8. The Has_Element function of the type shall have two parameters, the first of type T and the second of the cursor subtype of T; the result subtype of the function shall be Boolean.

9. The Element function of the type, if one is specified, shall have two parameters, the first of type T and the second of the cursor subtype of T; the default element subtype of T is then defined to be the result subtype of the Element function.

10. Reverse container element iterators are not in SPARK. The loop parameter of a container element iterator is a constant object.

11. A container element iterator shall only occur as the loop_parameter_specification of a quantified_expression[, and not as the iteration_scheme of a loop statement].

Dynamic Semantics

12. Iteration associated with a generalized iterator or a container element iterator proceeds as follows. An object of the iteration cursor subtype of T (hereafter called "the cursor") is created and is initialized to the result of calling First, passing in the given container object. Each iteration begins by calling Has_Element, passing in the container and the cursor. If False is returned, execution of the associated loop is completed. If True is returned then iteration continues and the loop parameter for the next iteration of the loop is either (in the case of a generalized iterator) the cursor or (in the case of a container element iterator) the result of calling the Element function, passing in the container and the cursor. At the end of the iteration, Next is called (passing in the container and the cursor) and the result is assigned to the cursor.

5.5.3 Loop Invariants, Variants and Entry Values

Two loop-related pragmas, Loop_Invariant and Loop_Variant, and a loop-related attribute, Loop_Entry are defined. The pragma Loop_Invariant is used to specify the essential non-varying properties of a loop. Pragma Loop_Variant is intended for use in ensuring termination. The Loop_Entry attribute is used to refer to the value that an expression had upon entry to a given loop in much the same way that the `Old` attribute in a subprogram postcondition can be used to refer to the value an expression had upon entry to the subprogram.

Syntax

```
loop_variant_parameters      ::= structural_loop_variant_item | numeric_loop_variant_
↪items
numeric_loop_variant_items   ::= numeric_loop_variant_item {, numeric_loop_variant_item}
numeric_loop_variant_item    ::= change_direction => expression
structural_loop_variant_item ::= Structural => expression
change_direction             ::= Increases | Decreases
```

Static Semantics

1. Pragma Loop_Invariant is like a pragma Assert except it also acts as a *cut point* in formal verification. A cut point means that a prover is free to forget all information about modified variables that has been established within the loop. Only the given Boolean expression is carried forward.

2. Pragma Loop_Variant is used to demonstrate that a loop will terminate by specifying expressions that will increase or decrease as the loop is executed.

Legality Rules

3. Loop_Invariant is an assertion just like pragma Assert with respect to syntax of its Boolean actual parameter, name resolution, legality rules and dynamic semantics, except for extra legality rules given below.

4. Loop_Variant is an assertion and has an expected actual parameter which is a specialization of an Ada expression. Otherwise, it has the same name resolution and legality rules as pragma Assert, except for extra legality rules given below.

5. The following constructs are said to be *restricted to loops*:

   - A Loop_Invariant pragma;

- A Loop_Variant pragma;
- A `block_statement` whose `sequence_of_statements` or `declarative_part` immediately includes a construct which is restricted to loops.

6. A construct which is restricted to loops shall occur immediately within either:

   - the `sequence_of_statements` of a `loop_statement`; or
   - the `sequence_of_statements` or `declarative_part` of a `block_statement`.

   The construct is said to apply to the innermost enclosing loop.

   [Roughly speaking, a Loop_Invariant or Loop_Variant pragma shall only occur immediately within a loop statement except that intervening block statements are ignored for purposes of this rule.]

7. The expression of a `numeric_loop_variant_item` shall be either of a discrete type or of a subtype of `Ada.Numerics.Big_Numbers.Big_Integers.Big_Integer`. In the second case, the associated `change_direction` shall be Decreases.

8. The expression of a `structural_loop_variant_item` shall denote a variable of an anonymous access-to-object type.

9. Two Loop_Invariant or Loop_Variant pragmas which apply to the same loop shall occur in the same `sequence_of_statements`, separated only by [zero or more] other Loop_Invariant or Loop_Variant pragmas.

Dynamic Semantics

10. Other than the above legality rules, pragma Loop_Invariant is equivalent to pragma `Assert`. Pragma Loop_Invariant is an assertion (as defined in Ada RM 11.4.2(1.1/3)) and is governed by the Loop_Invariant assertion aspect [and may be used in an Assertion_Policy pragma].

11. The elaboration of a Checked Loop_Variant pragma containing `numeric_loop_variant_items` begins by evaluating the `expressions` in textual order. For every expression whose type is a subtype of `Ada.Numerics.Big_Numbers.Big_Integers.Big_Integer`, a check is performed that it is non-negative. For the first elaboration of the pragma within a given execution of the enclosing loop statement, no further action is taken. For subsequent elaborations of the pragma, one or more of these expression results are each compared to their corresponding result from the previous iteration as follows: comparisons are performed in textual order either until unequal values are found or until values for all expressions have been compared. In either case, the last pair of values to be compared is then checked as follows: if the `change_direction` for the associated `loop_variant_item` is Increases (respectively, Decreases) then a check is performed that the expression value obtained during the current iteration is greater (respectively, less) than the value obtained during the preceding iteration. The exception Assertions.Assertion_Error is raised if this check fails. All comparisons and checks are performed using predefined operations. Pragma Loop_Variant is an assertion (as defined in Ada RM 11.4.2(1.1/3)) and is governed by the Loop_Variant assertion aspect [and may be used in an Assertion_Policy pragma].

Verification Rules

12. The variable denoted by the expression of a `structural_loop_variant_item` shall be updated on all paths reentering the loop to a strict subcomponent of the structure it used to denote.

13. No deep parts of the value designated by the variable denoted by the expression of a `structural_loop_variant_item` shall be written by the loop. [This ensures that the previous rule is sufficient to prove loop termination on acyclic data structures.]

Examples

The following example illustrates some pragmas of this section

```
procedure Loop_Var_Loop_Invar is
   type Total is range 1 .. 100;
   subtype T is Total range 1 .. 10;
```

(continues on next page)

(continued from previous page)

```
   I : T := 1;
   R : Total := 100;
begin
   while I < 10 loop
      pragma Loop_Invariant (R >= 100 - 10 * I);
      pragma Loop_Variant (Increases => I,
                           Decreases => R);
      R := R - I;
      I := I + 1;
   end loop;
end Loop_Var_Loop_Invar;
```

Note that in this example, the loop variant is unnecessarily complex, stating that `I` increases is enough to prove termination of this simple loop.

Attribute Loop_Entry

Static Semantics

1. For a prefix *X* that denotes an object of a nonlimited type, the following attribute is defined:

   ```
   X'Loop_Entry [(loop_name)]
   ```

2. X'Loop_Entry [(loop_name)] denotes a constant object of the type of X. [The value of this constant is the value of X on entry to the loop that is denoted by `loop_name` or, if no `loop_name` is provided, on entry to the closest enclosing loop.]

Legality Rules

3. A Loop_Entry `attribute_reference` *applies to* a `loop_statement` in the same way that an `exit_statement` does (see Ada RM 5.7). For every rule about `exit_statements` in the Name Resolution Rules and Legality Rules sections of Ada RM 5.7, a corresponding rule applies to Loop_Entry `attribute_references`.

4. In many cases, the language rules pertaining to the Loop_Entry attribute match those pertaining to the Old attribute (see Ada LRM 6.1.1), except with "Loop_Entry" substituted for "Old". These include:

   - prefix name resolution rules (including expected type definition)
   - nominal subtype definition
   - accessibility level definition
   - run-time tag-value determination (in the case where *X* is tagged)
   - interactions with abstract types
   - interactions with anonymous access types
   - forbidden attribute uses in the prefix of the `attribute_reference`.

   The following rules are not included in the above list; corresponding rules are instead stated explicitly below:

   - the requirement that an Old `attribute_reference` shall only occur in a postcondition expression;
   - the rule disallowing a use of an entity declared within the postcondition expression;
   - the rule that a potentially unevaluated Old `attribute_reference` shall statically name an entity;
   - the prefix of the `attribute_reference` shall not contain a Loop_Entry `attribute_reference`.

5. A Loop_Entry attribute_reference shall occur within a Loop_Variant or Loop_Invariant pragma, or an Assert, Assume or Assert_And_Cut pragma appearing in a position where a Loop_Invariant pragma would be allowed.

   [Roughly speaking, a Loop_Entry attribute_reference can occur in an Assert, Assume or Assert_And_Cut pragma immediately within a loop statement except that intervening block statements are ignored for purposes of this rule.]

6. The prefix of a Loop_Entry attribute_reference shall not contain a use of an entity declared within the loop_statement but not within the prefix itself.

   [This rule is to allow the use of I in the following example:

```
loop
   pragma Assert
     (Boolean'(Var > Some_Function (Param => (for all I in T => F (I))))'Loop_
↪Entry);
```

   In this example the value of the inequality ">" that would have been evaluated on entry to the loop is obtained even if the value of Var has since changed].

7. The prefix of a Loop_Entry attribute_reference shall statically name an entity, or shall denote an object_renaming_declaration, if

   * the attribute_reference is potentially unevaluated; or
   * the attribute_reference does not apply to the innermost enclosing loop_statement.

   [This rule follows the corresponding Ada RM rule for 'Old: the prefix of an Old attribute_reference that is potentially unevaluated shall statically name an entity. This rule has the same rationale. If the following was allowed:

```
procedure P (X : in out String; Idx : Positive) is
begin
   Outer :
      loop
         if Idx in X'Range then
            loop
               pragma Loop_Invariant (X(Idx) > X(Idx)'Loop_Entry(Outer));
```

   this would introduce an exception in the case where Idx is not in X'Range.]

8. The prefix of a Loop_Entry attribute_reference shall not contain a Loop_Entry attribute_reference.

Dynamic Semantics

9. For each X'Loop_Entry other than one occurring within an Ignored assertion expression, a constant is implicitly declared at the beginning of the associated loop statement. The constant is of the type of X and is initialized to the result of evaluating X (as an expression) at the point of the constant declaration. The value of X'Loop_Entry is the value of this constant; the type of X'Loop_Entry is the type of X. These implicit constant declarations occur in an arbitrary order.

10. The previous paragraph notwithstanding, the implicit constant declaration is not elaborated if the loop_statement has an iteration_scheme whose evaluation yields the result that the sequence_of_statements of the loop_statement will not be executed (loosely speaking, if the loop completes after zero iterations).

    [Note: This means that the constant is not elaborated unless the loop body will execute (or at least begin execution) at least once. For example, a while loop

```
while <condition> do
   sequence_of_statements; -- contains Loop_Entry uses
end loop;
```

may be thought of as being transformed into

```
if <condition> then
   declare
   ... implicitly declared Loop_Entry constants
   begin
      loop
         sequence_of_statements;
         exit when not <condition>;
      end loop;
   end;
end if;
```

The rule also prevents the following example from raising Constraint_Error:

```
 declare
    procedure P (X : in out String) is
    begin
       for I in X'Range loop
          pragma Loop_Invariant (X(X'First)'Loop_Entry >= X(I));
          X := F(X); -- modify X
       end loop;
    end P;
    Length_Is_Zero : String := "";
 begin
    P (Length_Is_Zero);
end; -- ...]
```

5.6 Block Statements

No extensions or restrictions.

5.7 Exit Statements

No extensions or restrictions.

5.8 Goto Statements

Legality Rules

1. A `goto_statement` shall be located before the target statement in the innermost `sequence_of_statements` enclosing the target statement.

5.9 Proof Pragmas

This section discusses the pragmas Assert_And_Cut and Assume.

Two SPARK pragmas are defined, Assert_And_Cut and Assume. Each is an assertion and has a single Boolean parameter (an assertion expression) and may be used wherever pragma Assert is allowed, with the additional restriction that pragma Assert_And_Cut must be part of a `sequence_of_statements`.

Assert_And_Cut may be used when the given expression sums up all the work done so far in the enclosing `sequence_of_statements`, so that the rest of the enclosing body can be verified (informally or formally) while treating the whole prefix preceding Assert_And_Cut as a single opaque (local) subprogram call, with post-condition provided by the Assert_And_Cut expression. This allows dividing up a subprogram into sections for the purposes of testing or formal verification. The pragma also serves as useful documentation.

A Boolean expression which is an actual parameter of pragma Assume can be assumed to be True for the remainder of the subprogram. If the Assertion_Policy is Check for pragma Assume and the Boolean expression does not evaluate to True, the exception Assertions.Assertion_Error will be raised. However, in proof, no verification of the expression is performed and in general it cannot. It has to be used with caution and is used to state axioms.

Static Semantics

1. Pragma Assert_And_Cut is an assertion the same as a pragma Assert except it also acts as a cut point in formal verification. The cut point means that a prover is free to forget all information about modified variables that has been established from the statement list before the cut point. Only the given Boolean expression is carried forward.
2. Pragma Assume is an assertion the same as a pragma Assert except that there is no verification condition to prove the truth of the Boolean expression that is its actual parameter. [Pragma Assume indicates to proof tools that the expression can be assumed to be True.]

Legality Rules

3. Pragmas Assert_And_Cut and Assume have the same syntax for their Boolean actual parameter, name resolution rules and dynamic semantics as pragma Assert.

Verification Rules

4. The verification rules for pragma Assume are significantly different to those of pragma Assert. [It would be difficult to overstate the importance of the difference.] Even though the dynamic semantics of pragma Assume and pragma Assert are identical, pragma Assume does not introduce a corresponding verification condition. Instead the prover is given permission to assume the truth of the assertion, even though this has not been proven. [A single incorrect Assume pragma can invalidate an arbitrarily large number of proofs - the responsibility for ensuring correctness rests entirely upon the user.]

Examples

```ada
-- The up-time timer is updated once a second
package Up_Timer
  with SPARK_Mode
is
   type Time_Register is limited private;
   type Times is range 0 .. 2**63 - 1;

   procedure Inc (Up_Time : in out Time_Register);

   function Get (Up_Time : Time_Register) return Times;

private
   type Time_Register is record
      Time : Times := 0;
   end record;
end Up_Timer;
```

```ada
package body Up_Timer
  with SPARK_Mode
is
   procedure Inc (Up_Time : in out Time_Register) is
   begin
      -- The up timer is incremented every second.
      -- The system procedures require that the system is rebooted
      -- at least once every three years - as the Timer_Reg is a 64 bit
      -- integer it cannot reach Times'Last before a system reboot.
      pragma Assume (if Times'Last = 2**63 - 1 then Up_Time.Time < Times'Last);

      -- Without the previous assume statement it would not be possible
      -- to prove that the following addition would not overflow.
      Up_Time.Time := Up_Time.Time + 1;
   end Inc;

   function Get (Up_Time : Time_Register) return Times is (Up_Time.Time);
end Up_Timer;
```

CHAPTER

SIX

SUBPROGRAMS

6.1 Subprogram Declarations

We distinguish the *declaration view* introduced by a `subprogram_declaration` from the *implementation view* introduced by a `subprogram_body` or an `expression_function_declaration`. For subprograms that are not declared by a `subprogram_declaration`, the `subprogram_body` or `expression_function_declaration` also introduces a declaration view which may be in SPARK even if the implementation view is not.

A *subprogram with side effects* is either a procedure, a protected entry, or a function with side effects (see *Functions With Side Effects*). A subprogram with side effects may have output parameters, write global variables, raise exceptions and not terminate.

Rules are imposed in SPARK to ensure that the execution of a function call does not modify any variables declared outside of the function, unless it is a function with side effects. Outside of this special case, it follows as a consequence of these rules that the evaluation of any SPARK expression is side-effect free.

We also introduce the notion of a *global item*, which is a name that denotes a global object or a state abstraction (see *Abstract_State Aspects*). Global items are presented in Global aspects (see *Global Aspects*).

An *entire object* is an object which is not a subcomponent of a larger containing object. More specifically, an *entire object* is an object declared by an `object_declaration` (as opposed to, for example, a slice or the result object of a function call) or a formal parameter of a subprogram. In particular, a component of a protected unit is not an *entire object*.

Static Semantics

1. The *exit* value of a global item or parameter of a subprogram is its value immediately following the call of the subprogram.
2. The *entry* value of a global item or parameter of a subprogram is its value at the call of the subprogram.
3. An *output* of a subprogram is a global item or parameter whose final value, or the final value of any of its reachable parts (see *Access Types*), may be updated by a successful call to the subprogram. The result of a function is also an output. A global item or parameter which is an external state with the property Async_Readers => True, and for which intermediate values are written during an execution leading to a successful call, is also an output even if the final state is the same as the initial state. (see *External State*). [On the contrary, a global item or parameter is not an output of the subprogram if it is updated only on paths that lead to a statement raising an unexpected exception or to a `pragma Assert (statically_False)`.]
4. An *input* of a subprogram is a global item or parameter whose initial value (or that of any of its reachable parts - see *Access Types*) may be used in determining the exit value of an output of the subprogram. For a global item or parameter which is an external state with Async_Writers => True, each successive value read from the external state is also an input of the subprogram (see *External State*). As a special case, a global item or parameter is also an input if it is mentioned in a `null_dependency_clause` in the Depends aspect of the subprogram (see *Depends Aspects*).

5. An output of a subprogram is said to be *fully initialized* by a call if all parts of the output are initialized as a result of any successful execution of a call of the subprogram. In the case of a parameter X of a class-wide type T'Class, this set of "all parts" is not limited to the (statically known) parts of T. For example, if the underlying dynamic tag of X is T2'Tag, where T2 is an extension of T that declares a component C, then C would be included in the set. In this case, this set of "all parts" is not known statically. [In order to fully initialize such a parameter, it is necessary to use some form of dispatching assignment. This can be done by either a direct (class-wide) assignment to X, passing X as an actual out-mode parameter in a call where the formal parameter is of a class-wide type, or passing X as a controlling out-mode parameter in a dispatching call.] The meaning of "all parts" in the case of a parameter of a specific tagged type is determined by the applicable Extensions_Visible aspect (see *Extensions_Visible Aspects*). [A state abstraction cannot be fully initialized by initializing individual constituents unless its refinement is visible.]

Legality Rules

6. The declaration of a function without side effects shall not have a `parameter_specification` with a mode of **out** or **in out**. This rule also applies to a `subprogram_body` for a function without side effects for which no explicit declaration is given. A function without side effects shall have no outputs other than its result.

7. A subprogram parameter of mode **in** shall not be an output of its subprogram unless the type of the parameter is an access type and the subprogram is a subprogram with side effects.

Verification Rules

8. At the point of a call, all inputs of the callee except for those that have relaxed initialization (see *Relaxed Initialization*) shall be fully initialized. Similarly, upon return from a call all outputs of the callee except for those that have relaxed initialization shall be fully initialized.

9. If a call propagates an exception, all global outputs of the callee and all output parameters which either have a *by reference* type or are marked as aliased shall be fully initialized when the exception is propagated except for those that have relaxed initialization.

10. A function without side effects shall always return normally.

11. A call to a ghost procedure occurring outside of a ghost context shall always return normally.

6.1.1 Preconditions and Postconditions

Legality Rules

1. The corresponding expression for an inherited Pre'Class or Post'Class of an inherited subprogram S of a tagged type T shall not call a non-inherited primitive function of type T.

[The notion of corresponding expression is defined in Ada RM 6.1.1(18/4) as follows: If a Pre'Class or Post'Class aspect is specified for a primitive subprogram S of a tagged type T, or such an aspect defaults to True, then a corresponding expression also applies to the corresponding primitive subprogram S of each descendant of T.]

[The rationale for this rule is that, otherwise, if the contract applicable to an inherited subprogram changes due to called subprograms in its contract being overridden, then the inherited subprogram would have to be re-verified for the derived type. This rule forbids the cases that require re-verification.]

2. The Pre aspect shall not be specified for a primitive operation of a type T at a point where T is tagged. [Pre'Class should be used instead to express preconditions.]

[The rationale for this rule is that, otherwise, the combination of dynamic semantics and verification rules below would force an identical Pre'Class each time Pre is used on a dispatching operation.]

3. A subprogram_renaming_declaration shall not declare a primitive operation of a tagged type.

[Consider

```
package Outer is
   type T is tagged null record;
   package Nested is
      procedure Op (X : T) with Pre => ..., Post => ... ;
      -- not a primitive, so Pre/Post specs are ok
   end Nested;
   procedure Renamed_Op (X : T) renames Nested.Op; -- illegal
end Outer;
```

Allowing this example in SPARK would introduce a case of a dispatching operation which is subject to a Pre (and Post) aspect specification. This rule is also intended to avoid problematic interactions between the Pre/Pre'Class/Post/Post'Class aspects of the renamed subprogram and the Pre'Class/Post'Class inheritance associated with the declaration of a primitive operation of a tagged type.

Note that a dispatching subprogram can be renamed as long as the renaming does not itself declare a dispatching operation. Note also that this rule would never apply to a renaming-as-body.]

Verification Rules

For a call on a nondispatching operation, a verification condition is introduced (as for any run-time check) to ensure that the specific precondition check associated with the statically denoted callee will succeed. Upon entry to such a subprogram, the specific preconditions of the subprogram may then be assumed.

For a call (dispatching or not) on a dispatching operation, a verification condition is introduced (as for any run-time check) to ensure that the class-wide precondition check associated with the statically denoted callee will succeed.

The verification condition associated with the specific precondition of a dispatching subprogram is imposed on the callee, as opposed to on callers of the subprogram. Upon entry to a subprogram, the class-wide preconditions of the subprogram may be assumed. Given this, the specific preconditions of the subprogram must be proven.

The callee is responsible for discharging the verification conditions associated with any postcondition checks, class-wide or specific. The success of these checks may then be assumed by the caller.

In the case of an overriding dispatching operation whose Pre'Class attribute is explicitly specified, a verification condition is introduced to ensure that the specified Pre'Class condition is implied by the Pre'Class condition of the overridden inherited subprogram(s). Similarly, in the case of an overriding dispatching operation whose Post'Class attribute is explicitly specified, a verification condition is introduced to ensure that the specified Post'Class condition implies the Post'Class condition of the overridden inherited subprogram(s). [These verification conditions do not correspond to any run-time check. They are intended to, in effect, require users to make explicit the implicit disjunction/conjunction of class-wide preconditions/postconditions that is described in Ada RM 6.1.1.]

6.1.2 Subprogram Contracts

In order to extend Ada's support for specification of subprogram contracts (e.g., the Pre and Post) by providing more precise and/or concise contracts, the SPARK aspects, Global, Depends, and Contract_Cases are defined.

Legality Rules

1. The Global, Depends and Contract_Cases aspects may be specified for a subprogram with an `aspect_specification`. More specifically, such aspect specifications are allowed in the same contexts as Pre or Post aspect specifications. [In particular, these aspects may be specified for a generic subprogram but not for an instance of a generic subprogram.]
2. The Global and Depends (but not Contract_Cases) aspects may be specified for an abstract subprogram.
3. The Global, Depends and Contract_Cases aspects shall not be specified for a null procedure.

See section *Contract Cases* for further detail on Contract_Case aspects, section *Global Aspects* for further detail on Global aspects and section *Depends Aspects* for further detail on Depends aspects.

6.1.3 Contract Cases

The Contract_Cases aspect provides a structured way of defining a subprogram contract using mutually exclusive subcontract cases. The final case in the Contract_Case aspect may be the keyword **others** which means that, in a specific call to the subprogram, if all the `conditions` are False this `contract_case` is taken. If an **others** `contract_case` is not specified, then in a specific call of the subprogram exactly one of the guarding `conditions` should be True.

A Contract_Cases aspect may be used in conjunction with the language-defined aspects Pre and Post in which case the precondition specified by the Pre aspect is augmented with a check that exactly one of the `conditions` of the `contract_case_list` is satisfied and the postcondition specified by the Post aspect is conjoined with conditional expressions representing each of the `contract_cases`. For example:

```
procedure P (...)
   with Pre  => General_Precondition,
        Post => General_Postcondition,
        Contract_Cases => (A1 => B1,
                           A2 => B2,
                           ...
                           An => Bn);
```

is short hand for

```
procedure P (...)
  with Pre  => General_Precondition
                 and then Exactly_One_Of (A1, A2, ..., An),
       Post => General_Postcondition
                 and then (if A1'Old then B1)
                 and then (if A2'Old then B2)
                 and then ...
                 and then (if An'Old then Bn);
```

where

A1 .. An are Boolean expressions involving the entry values of formal parameters and global objects and

B1 .. Bn are Boolean expressions that may also use the exit values of formal parameters, global objects and results.

`Exactly_One_Of(A1,A2...An)` evaluates to True if exactly one of its inputs evaluates to True and all other of its inputs evaluate to False.

The Contract_Cases aspect is specified with an `aspect_specification` where the `aspect_mark` is Contract_Cases and the `aspect_definition` must follow the grammar of `contract_case_list` given below.

Syntax

```
contract_case_list  ::= (contract_case {, contract_case})
contract_case       ::= condition => consequence
                      | others => consequence
```

where

`consequence ::=` *Boolean*_`expression`

Legality Rules

1. A Contract_Cases aspect may have at most one **others** `contract_case` and if it exists it shall be the last one in the `contract_case_list`.

2. A `consequence` expression is considered to be a postcondition expression for purposes of determining the legality of Old or Result `attribute_references`.

Static Semantics

3. A Contract_Cases aspect is an assertion (as defined in RM 11.4.2(1.1/3)); its assertion expressions are as described below. Contract_Cases may be specified as an `assertion_aspect_mark` in an Assertion_Policy pragma.

Dynamic Semantics

4. Upon a call of a subprogram which is subject to an enabled Contract_Cases aspect, Contract_Cases checks are performed as follows:

   - Immediately after the specific precondition expression is evaluated and checked (or, if that check is disabled, at the point where the check would have been performed if it were enabled), all of the `conditions` of the `contract_case_list` are evaluated in textual order. A check is performed that exactly one (if no **others** `contract_case` is provided) or at most one (if an **others** `contract_case` is provided) of these `conditions` evaluates to True; Assertions.Assertion_Error is raised if this check fails.

   - Immediately after the specific postcondition expression is evaluated and checked (or, if that check is disabled, at the point where the check would have been performed if it were enabled), exactly one of the `consequences` is evaluated. The `consequence` to be evaluated is the one corresponding to the one `condition` whose evaluation yielded True (if such a `condition` exists), or to the **others** `contract_case` (if every `condition`'s evaluation yielded False). A check is performed that the evaluation of the selected `consequence` evaluates to True; Assertions.Assertion_Error is raised if this check fails.

5. If an Old `attribute_reference` occurs within a `consequence` other than the `consequence` selected for (later) evaluation as described above, then the associated implicit constant declaration (see Ada RM 6.1.1) is not elaborated. [In particular, the prefix of the Old `attribute_reference` is not evaluated].

Verification Rules

The verification conditions associated with the Contract_Cases runtime checks performed at the beginning of a call are assigned in the same way as those associated with a specific precondition check. More specifically, the verification condition is imposed on the caller or on the callee depending on whether the subprogram in question is a dispatching operation.

Examples

```
-- This subprogram is specified using a Contract_Cases aspect.
-- The prover will check that the cases are disjoint and
-- cover the domain of X.
procedure Incr_Threshold (X : in out Integer; Threshold : in Integer)
  with Contract_Cases => (X < Threshold  => X = X'Old + 1,
                          X >= Threshold => X = X'Old);

-- This is the equivalent specification not using Contract_Cases.
-- It is noticeably more complex and the prover is not able to check
-- for disjoint cases or that the domain of X is covered.
procedure Incr_Threshold_1 (X : in out Integer; Threshold : in Integer)
  with Pre  => (X < Threshold and not (X >= Threshold))
                or else (not (X < Threshold) and X >= Threshold),
       Post => (if X'Old < Threshold then X = X'Old + 1
                elsif X'Old >= Threshold then X = X'Old);

-- Contract_Cases can be used in conjunction with pre and postconditions.
procedure Incr_Threshold_2 (X : in out Integer; Threshold : in Integer)
```

(continues on next page)

(continued from previous page)

```
with Pre  => X in 0 .. Threshold,
     Post => X >= X'Old,
     Contract_Cases => (X < Threshold => X = X'Old + 1,
                        X = Threshold => X = X'Old);
```

6.1.4 Global Aspects

A Global aspect of a subprogram lists the global items whose values are used or affected by a call of the subprogram.

The Global aspect shall only be specified for the initial declaration of a subprogram (which may be a declaration, a body or a body stub), of a protected entry, or of a task unit. The implementation of a subprogram body shall be consistent with the subprogram's Global aspect. Similarly, the implementation of an entry or task body shall be consistent with the entry or task's Global aspect.

Note that a Refined_Global aspect may be applied to a subprogram body when using state abstraction; see section *Refined_Global Aspects* for further details.

The Global aspect is introduced by an `aspect_specification` where the `aspect_mark` is Global and the `aspect_definition` must follow the grammar of `global_specification`

For purposes of the rules concerning the Global, Depends, Refined_Global, and Refined_Depends aspects, when any of these aspects are specified for a task unit the task unit's body is considered to be the body of a nonreturning procedure and the current instance of the task unit is considered to be a formal parameter (of that notional procedure) of mode **in out**. [For example, rules which refer to the "subprogram body" refer, in the case of a task unit, to the task body.] [Because a task (even a discriminated task) is effectively a constant, one might think that a mode of **in** would make more sense. However, the current instance of a task unit is, strictly speaking, a variable; for example, it may be passed as an actual **out** or **in out** mode parameter in a call.] The Depends and Refined_Depends aspect of a task unit T need not mention this implicit parameter; an implicit specification of "T => T" is assumed, although this may be confirmed explicitly.

Similarly, for purposes of the rules concerning the Global, Refined_Global, Depends, and Refined_Depends aspects as they apply to protected operations, the current instance of the enclosing protected unit is considered to be a formal parameter (of mode **in** for a protected function, of mode **in out** otherwise) and a protected entry is considered to be a protected procedure. [For example, rules which refer to the "subprogram body" refer, in the case of a protected entry, to the entry body. As another example, the Global aspect of a subprogram nested within a protected operation might name the current instance of the protected unit as a global in the same way that it might name any other parameter of the protected operation.]

[Note that AI12-0169 modifies the Ada RM syntax for an `entry_body` to allow an optional `aspect_specification` immediately before the `entry_barrier`. This is relevant for aspects such as Refined_Global and Refined_Depends.]

Syntax

```
global_specification        ::= (moded_global_list {, moded_global_list})
                              | global_list
                              | null_global_specification
moded_global_list           ::= mode_selector => global_list
global_list                 ::= global_item
                              | (global_item {, global_item})
mode_selector               ::= Input | Output | In_Out | Proof_In
global_item                 ::= name
null_global_specification   ::= null
```

Static Semantics

1. A `global_specification` that is a `global_list` is shorthand for a `moded_global_list` with the `mode_selector` Input.

2. A `global_item` is *referenced* by a subprogram if:

   - It denotes an input or an output of the subprogram, or;
   - Its entry value is used to determine the value of an assertion expression within the subprogram, or;
   - Its entry value is used to determine the value of an assertion expression within another subprogram that is called either directly or indirectly by this subprogram.

3. A `null_global_specification` indicates that the subprogram does not reference any `global_item` directly or indirectly.

4. If a subprogram's Global aspect is not otherwise specified and either

   - the subprogram is a library-level subprogram declared in a library unit that is declared pure (i.e., a subprogram to which the implementation permissions of Ada RM 10.2.1 apply); or
   - a Pure_Function pragma applies to the subprogram

   then a Global aspect of *null* is implicitly specified for the subprogram.

Name Resolution Rules

5. A `global_item` shall denote an entire object or a state abstraction. [This is a name resolution rule because a `global_item` can unambiguously denote a state abstraction even if a function having the same fully qualified name is also present].

Legality Rules

6. The Global aspect may only be specified for the initial declaration of a subprogram (which may be a declaration, a body or a body stub), of a protected entry, or of a task unit.

7. A `global_item` occurring in a Global aspect specification of a subprogram shall not denote a formal parameter of the subprogram.

8. A `global_item` shall not denote a state abstraction whose refinement is visible. [A state abstraction cannot be named within its enclosing package's body other than in its refinement. Its constituents shall be used rather than the state abstraction.]

9. Each `mode_selector` shall occur at most once in a single Global aspect.

10. A function without side effects shall not have a `mode_selector` of Output or In_Out in its Global aspect.

11. A user-defined primitive equality operation on a record type shall have a Global aspect of `null`, unless the record type has only limited views (see *Overloading of Operators*).

    [This avoids the case where such a record type is a component of another composite type, whose predefined equality operation now depends on variables through the primitive equality operation on its component.]

12. The `global_items` in a single Global aspect specification shall denote distinct entities. Additionally, if a `global_item` is a state abstraction, none of its constituents shall appear as a `global_item` in the same Global aspect specification.

13. If a subprogram is nested within another and if the `global_specification` of the outer subprogram has an entity denoted by a `global_item` with a `mode_specification` of Input or the entity is a formal parameter with a mode of **in**, then a `global_item` of the `global_specification` of the inner subprogram shall not denote the same entity with a `mode_selector` of In_Out or Output.

14. A `global_item` occurring with mode Input in the Global aspect specification of a function annotated with Pure_Function aspect or pragma shall denote a constant object whose type is not an owning type (see *Access Types*).

[This restriction ensures that two calls to the function with the same parameters return the same value, so that the compiler can safely apply corresponding optimizations.]

Dynamic Semantics

There are no dynamic semantics associated with a Global aspect as it is used purely for static analysis purposes and is not executed.

Verification Rules

15. For a subprogram that has a `global_specification`, an object (except a constant without variable inputs) or state abstraction that is declared outside the scope of the subprogram, shall only be referenced within its implementation if it is a `global_item` in the `global_specification`.

16. A `global_item` shall occur in a Global aspect of a subprogram if and only if it denotes an entity (except for a constant without variable inputs) that is referenced by the subprogram.

17. Where the refinement of a state abstraction is not visible (see *State Refinement*) and a subprogram references one or more of its constituents, the constituents may be represented by a `global_item` that denotes the state abstraction in the `global_specification` of the subprogram. [The state abstraction encapsulating a constituent is known from the Part_Of indicator on the declaration of the constituent.]

18. Each entity denoted by a `global_item` in a `global_specification` of a subprogram that is an input or output of the subprogram shall satisfy the following mode specification rules [which are checked during analysis of the subprogram body]:

    - a `global_item` that denotes an input but not an output has a `mode_selector` of Input;
    - a `global_item` has a `mode_selector` of Output if:
      - it denotes an output but not an input, other than the use of a discriminant or an attribute related to a property, not its value, of the `global_item` [examples of attributes that may be used are A'Last, A'First and A'Length; examples of attributes that are dependent on the value of the object and shall not be used are X'Old and X'Loop_Entry] and
      - it does not have relaxed initialization (see *Relaxed Initialization*);
    - a `global_item` that denotes an output which is not an input and which has relaxed initialization may have a `mode_selector` of Output or In_Out;
    - otherwise the `global_item` denotes both an input and an output, and has a `mode_selector` of In_Out.

    [For purposes of determining whether an output of a subprogram shall have a `mode_selector` of Output or In_Out, reads of array bounds, discriminants, or tags of the output are ignored. Reads of array bounds, discriminants, or tag of any reachable part of the output are not considered either if they are constrained by their subtype. Similarly, for purposes of determining whether an entity is fully initialized as a result of any successful execution of the call, the mutable discriminants of the output itself are not considered. This implies that given an output of a discriminated type that is not known to be constrained ("known to be constrained" is defined in Ada RM 3.3), the discriminants of the output might or might not be updated by the call.]

19. An entity that is denoted by a `global_item` which is referenced by a subprogram but is neither an input nor an output but is only referenced directly, or indirectly in assertion expressions has a `mode_selector` of Proof_In.

20. A `global_item` shall not denote a constant object other than a formal parameter [of an enclosing subprogram] of mode **in**, a generic formal object of mode **in**, a constant of (named or anonymous) access-to-variable type, or a *constant with variable inputs*.

    If a `global_item` denotes a generic formal object of mode **in**, then the corresponding `global_item` in an instance of the generic unit may denote a constant which has no variable inputs. [This can occur if the corresponding actual parameter is an expression which has no variable inputs]. Outside of the instance, such a `global_item` is ignored. For example,

```
package Global_And_Generics is

   generic
      X : Integer;
   package G is
      procedure P (Y : in out Integer) with
        Global  => X,
        Depends => (Y =>+ X);
   end G;

   procedure Q (Z : in out Integer) with
     Global  => null,
     Depends => (Z =>+ null);

end Global_And_Generics;
```

```
package body Global_And_Generics is

   package body G is
      procedure P (Y : in out Integer) is
      begin
         Y := Integer'Max (X, Y);
      end P;
   end G;

   package I is new G
     (X => 123); -- actual parameter lacks variable inputs

   -- Q's Global and Depends aspects don't mention I.X even though
   -- Q calls I.P which does reference I.X as a global.
   -- As seen from outside of I, I.P's references to I.X in its
   -- Global and Depends aspect specifications are ignored.
   procedure Q (Z : in out Integer) is
   begin
      I.P (Y => Z);
   end Q;

end Global_And_Generics;
```

21. The `mode_selector` of a `global_item` denoting a *constant with variable inputs* shall be `Input` or `Proof_In`.

22. The `mode_selector` of a `global_item` denoting a variable marked as a *constant after elaboration* shall be `Input` or `Proof_In` [, to ensure that such variables are only updated directly by package elaboration code]. A subprogram or entry having such a `global_item` shall not be called during library unit elaboration[, to ensure only the final (“constant”) value of the object is referenced].

Examples

```
with Global => null; -- Indicates that the subprogram does not reference
                     -- any global items.
with Global => V;    -- Indicates that V is an input of the subprogram.
with Global => (X, Y, Z);  -- X, Y and Z are inputs of the subprogram.
with Global => (Input    => V); -- Indicates that V is an input of the subprogram.
with Global => (Input    => (X, Y, Z)); -- X, Y and Z are inputs of the subprogram.
```

(continues on next page)

(continued from previous page)

```
with Global => (Output   => (A, B, C)); -- A, B and C are outputs of
                                        -- the subprogram.
with Global => (In_Out   => (D, E, F)); -- D, E and F are both inputs and
                                        -- outputs of the subprogram
with Global => (Proof_In => (G, H));    -- G and H are only used in
                                        -- assertion expressions within
                                        -- the subprogram
with Global => (Input    => (X, Y, Z),
                Output   => (A, B, C),
                In_Out   => (P, Q, R),
                Proof_In => (T, U));
                -- A global aspect with all types of global specification
```

6.1.5 Depends Aspects

A Depends aspect defines a *dependency relation* for a subprogram which may be given in the `aspect_specification` of the subprogram. A dependency relation is a sort of formal specification which specifies a simple relationship between inputs and outputs of the subprogram. It may be used with or without a postcondition.

The Depends aspect shall only be specified for the initial declaration of a subprogram (which may be a declaration, a body or a body stub), of a protected entry, or of a task unit.

Unlike a postcondition, the Depends aspect must be complete in the sense that every input and output of the subprogram must appear in it. A postcondition need only specify properties of particular interest.

Like a postcondition, the dependency relation may be omitted from a subprogram declaration when it defaults to the conservative relation that each output depends on every input of the subprogram. A particular SPARK tool may synthesize a more accurate approximation from the subprogram implementation if it is present (see *Synthesis of SPARK Aspects*).

For accurate information flow analysis the Depends aspect should be present on every subprogram.

A Depends aspect for a subprogram specifies for each output every input on which it depends. The meaning of *X depends on Y* in this context is that the input value(s) of *Y* may affect:

- the exit value of *X*; and
- the intermediate values of *X* if it is an external state (see section *External State*), or if the subprogram is a nonreturning procedure [, possibly the notional nonreturning procedure corresponding to a task body].

This is written *X* => *Y*. As in UML, the entity at the tail of the arrow depends on the entity at the head of the arrow.

If an output does not depend on any input this is indicated using a **null**, e.g., *X* => **null**. An output may be self-dependent but not dependent on any other input. The shorthand notation denoting self-dependence is useful here, X =>+ **null**.

Note that a Refined_Depends aspect may be applied to a subprogram body when using state abstraction; see section *Refined_Depends Aspects* for further details.

See section *Global Aspects* regarding how the rules given in this section apply to protected operations and to task bodies.

The Depends aspect is introduced by an `aspect_specification` where the `aspect_mark` is Depends and the `aspect_definition` must follow the grammar of `dependency_relation` given below.

Syntax

```
dependency_relation    ::= null
                         | (dependency_clause {, dependency_clause})
dependency_clause      ::= output_list =>[+] input_list
                         | null_dependency_clause
null_dependency_clause ::= null => input_list
output_list            ::= output
                         | (output {, output})
input_list             ::= input
                         | (input {, input})
                         | null
input                  ::= name
output                 ::= name | function_result
```

where

`function_result` is a function Result `attribute_reference`.

Name Resolution Rules

1. An `input` or `output` of a `dependency_relation` shall denote only an entire object or a state abstraction. [This is a name resolution rule because an `input` or `output` can unambiguously denote a state abstraction even if a function having the same fully qualified name is also present.]

Legality Rules

2. The Depends aspect shall only be specified for the initial declaration of a subprogram (which may be a declaration, a body or a body stub), of a protected entry, or of a task unit.
3. An `input` or `output` of a `dependency_relation` shall not denote a state abstraction whose refinement is visible [a state abstraction cannot be named within its enclosing package's body other than in its refinement].
4. The *explicit input set* of a subprogram is the set of formal parameters of the subprogram of mode **in** and **in out** along with the entities denoted by `global_items` of the Global aspect of the subprogram with a `mode_selector` of Input and In_Out.
5. The *input set* of a subprogram is the explicit input set of the subprogram augmented with those formal parameters of mode **out** and those `global_items` with a `mode_selector` of Output having discriminants, array bounds, or a tag which can be read and whose values are not implied by the subtype of the parameter. More specifically, it includes formal parameters of mode **out** and `global_items` with a `mode_selector` of Output which are of an unconstrained array subtype, an unconstrained discriminated subtype, or a tagged type (with one exception). The exception mentioned in the previous sentence is in the case where the formal parameter is of a specific tagged type and the applicable Extensions_Visible aspect is False. In that case, the tag of the parameter cannot be read and so the fact that the parameter is tagged does not cause it to included in the subprogram's *input_set*, although it may be included for some other reason (e.g., if the parameter is of an unconstrained discriminated subtype).
6. The *output set* of a subprogram is the set of formal parameters of the subprogram of mode **in out** and **out** along with the entities denoted by `global_items` of the Global aspect of the subprogram with a `mode_selector` of In_Out and Output and (for a function) the `function_result` or (for a subprogram with side effects) the set of formal parameters of the subprogram of mode **in** of an access-to-variable type.
7. The entity denoted by each `input` of a `dependency_relation` of a subprogram shall be a member of the input set of the subprogram.
8. Every member of the explicit input set of a subprogram shall be denoted by at least one `input` of the `dependency_relation` of the subprogram.
9. The entity denoted by each `output` of a `dependency_relation` of a subprogram shall be a member of the output set of the subprogram.

10. Every member of the output set of a subprogram shall be denoted by exactly one `output` in the `dependency_relation` of the subprogram.
11. For the purposes of determining the legality of a Result `attribute_reference`, a `dependency_relation` is considered to be a postcondition of the function to which the enclosing `aspect_specification` applies.
12. In a `dependency_relation` there can be at most one `dependency_clause` which is a `null_dependency_clause` and if it exists it shall be the last `dependency_clause` in the `dependency_relation`.
13. An entity denoted by an `input` which is in an `input_list` of a `null_dependency_clause` shall not be denoted by an `input` in another `input_list` of the same `dependency_relation`.
14. The `inputs` in a single `input_list` shall denote distinct entities.
15. A `null_dependency_clause` shall not have an `input_list` of **null**.

Static Semantics

16. A `dependency_clause` with a "+" symbol in the syntax `output_list =>+ input_list` means that each `output` in the `output_list` has a *self-dependency*, that is, it is dependent on itself. [The text (A, B, C) =>+ Z is shorthand for (A => (A, Z), B => (B, Z), C => (C, Z)).]
17. A `dependency_clause` of the form A =>+ A has the same meaning as A => A. [The reason for this rule is to allow the short hand: ((A, B) =>+ (A, C)) which is equivalent to (A => (A, C), B => (A, B, C)).]
18. A `dependency_clause` with a **null** `input_list` means that the final value of the entity denoted by each `output` in the `output_list` does not depend on any member of the input set of the subprogram (other than itself, if the `output_list =>+` **null** self-dependency syntax is used).
19. The `inputs` in the `input_list` of a `null_dependency_clause` may be read by the subprogram but play no role in determining the values of any outputs of the subprogram.
20. A Depends aspect of a subprogram with a **null** `dependency_relation` indicates that the subprogram has no `inputs` or `outputs`. [From an information flow analysis viewpoint it is a null operation (a no-op).]
21. A function without side effects without an explicit Depends aspect specification has the default `dependency_relation` that its result is dependent on all of its inputs. [Generally an explicit Depends aspect is not required for a function declaration.]
22. A subprogram with side effects without an explicit Depends aspect specification has a default `dependency_relation` that each member of its output set is dependent on every member of its input set. [This conservative approximation may be improved by analyzing the body of the subprogram if it is present.]

Dynamic Semantics

There are no dynamic semantics associated with a Depends aspect as it is used purely for static analysis purposes and is not executed.

Verification Rules

23. Each entity denoted by an `output` given in the Depends aspect of a subprogram shall be an output in the implementation of the subprogram body and the output shall depend on all, but only, the entities denoted by the `inputs` given in the `input_list` associated with the `output`.
24. Each output of the implementation of the subprogram body is denoted by an `output` in the Depends aspect of the subprogram.
25. Each input of the implementation of a subprogram body is denoted by an `input` of the Depends aspect of the subprogram.
26. If not all parts of an output are updated, then the updated entity is dependent on itself as the parts that are not updated have their current value preserved.

[In the case of a parameter of a tagged type (specific or class-wide), see the definition of "fully initialized" for a clarification of what the phrase "all parts" means in the preceding sentence.]

Examples

```
procedure P (X, Y, Z in : Integer; Result : out Boolean)
  with Depends => (Result => (X, Y, Z));
-- The exit value of Result depends on the entry values of X, Y and Z

procedure Q (X, Y, Z in : Integer; A, B, C, D, E : out Integer)
  with Depends => ((A, B) => (X, Y),
                   C      => (X, Z),
                   D      => Y,
                   E      => null);
-- The exit values of A and B depend on the entry values of X and Y.
-- The exit value of C depends on the entry values of X and Z.
-- The exit value of D depends on the entry value of Y.
-- The exit value of E does not depend on any input value.

procedure R (X, Y, Z : in Integer; A, B, C, D : in out Integer)
  with Depends => ((A, B) =>+ (A, X, Y),
                   C      =>+ Z,
                   D      =>+ null);
-- The "+" sign attached to the arrow indicates self-dependency, that is
-- the exit value of A depends on the entry value of A as well as the
-- entry values of X and Y.
-- Similarly, the exit value of B depends on the entry value of B
-- as well as the entry values of A, X and Y.
-- The exit value of C depends on the entry value of C and Z.
-- The exit value of D depends only on the entry value of D.

procedure S
  with Global  => (Input  => (X, Y, Z),
                   In_Out => (A, B, C, D)),
       Depends => ((A, B) =>+ (A, X, Y, Z),
                   C      =>+ Y,
                   D      =>+ null);
-- Here globals are used rather than parameters and global items may appear
-- in the Depends aspect as well as formal parameters.

function F (X, Y : Integer) return Integer
  with Global  => G,
       Depends => (F'Result => (G, X),
                   null     => Y);
-- Depends aspects on functions are only needed for special cases like here where the
-- parameter Y has no discernible effect on the result of the function.
```

6.1.6 Global and Depends Aspects of Dispatching Subprograms

Additional rules apply to the Global and Depends aspects on a dispatching subprogram, in order to ensure that the effects of dynamically calling an overriding subprogram are properly captured by the aspects of the statically denoted callee.

Static Semantics

1. A Global aspect specification G2 is said to be a *valid overriding* of another such specification, G1, if the following conditions are met:
   - each Input-mode item of G2 is an Input-mode or an In_Out-mode item of G1 or a direct or indirect constituent thereof; and
   - each In_Out-mode item of G2 is an In_Out-mode item of G1 or a direct or indirect constituent thereof; and
   - each Output-mode item of G2 is an Output-mode or In_Out-mode item of G1 or a direct or indirect constituent thereof; and
   - each Output-mode item of G1 which is not a state abstraction whose refinement is visible at the point of G2 is an Output-mode item of G2; and
   - for each Output-mode item of G1 which is a state abstraction whose refinement is visible at the point of G2, each direct or indirect constituent thereof is an Output-mode item of G2.
2. A Depends aspect specification D2 is said to be a *valid overriding* of another such specification, D1, if the set of dependencies of D2 is a subset of the dependencies of D1 or, in the case where D1 mentions a state abstraction whose refinement is visible at the point of D2, if D2 is derivable from such a subset as described in *Refined_Depends Aspects*.

Legality Rules

3. The Global aspect of an overriding subprogram shall be a valid overriding of the Global aspect(s) of the overridden inherited subprogram(s).
4. The Depends aspect of an overriding subprogram shall be a valid overriding of the Depends aspect(s) of the overridden inherited subprogram(s).

6.1.7 Extensions_Visible Aspects

The Extensions_Visible aspect provides a mechanism for ensuring that "hidden" components of a formal parameter of a specific tagged type are unreferenced. For example, if a formal parameter of a specific tagged type T is converted to a class-wide type and then used as a controlling operand in a dispatching call, then the (dynamic) callee might reference components of the parameter which are declared in some extension of T. Such a use of the formal parameter could be forbidden via an Extensions_Visible aspect specification as described below. The aspect also plays a corresponding role in the analysis of callers of the subprogram.

Static Semantics

1. Extensions_Visible is a Boolean-valued aspect which may be specified for a noninstance subprogram or a generic subprogram. If directly specified, the aspect_definition shall be a static [Boolean] expression. The aspect is inherited by an inherited primitive subprogram. If the aspect is neither inherited nor directly specified for a subprogram, then the aspect is False, except in the case of the predefined equality operator of a type extension. In that case, the aspect value is that of the primitive [(possibly user-defined)] equality operator for the parent type.

Legality Rules

2. If the Extensions_Visible aspect is False for a subprogram, then certain restrictions are imposed on the use of any parameter of the subprogram which is of a specific tagged type (or of a private type whose full view is a specific

tagged type). Such a parameter shall not be converted (implicitly or explicitly) to a class-wide type. Such a parameter shall not be passed as an actual parameter in a call to a subprogram whose Extensions_Visible aspect is True. These restrictions also apply to any parenthesized expression, qualified expression, or type conversion whose operand is subject to these restrictions, to any Old or Loop_Entry `attribute_reference` whose prefix is subject to these restrictions, to any delta aggregate whose expression is subject to these restrictions, and to any conditional expression having at least one dependent_expression which is subject to these restrictions. [A subcomponent of a parameter is not itself a parameter and is therefore not subject to these restrictions. A parameter whose type is class-wide is not subject to these restrictions. An Old or Loop_Entry `attribute_reference` does not itself violate these restrictions (despite the fact that (in the tagged case) each of these attributes yields a result having the same underlying dynamic tag as their prefix).]

3. A subprogram whose Extensions_Visible aspect is True shall not override an inherited primitive operation of a tagged type whose Extensions_Visible aspect is False. [The reverse is allowed.]

4. If a nonnull type extension inherits a procedure having both a False Extensions_Visible aspect and one or more controlling out-mode parameters, then the inherited procedure requires overriding. [This is because the inherited procedure would not initialize the noninherited component(s).]

5. The Extensions_Visible aspect shall not be specified for a subprogram which has no parameters of either a specific tagged type or a private type unless the subprogram is declared in an instance of a generic unit and the corresponding subprogram in the generic unit satisfies this rule. [Such an aspect specification, if allowed, would be ineffective.]

6. [These rules ensure that the value of the underlying tag (at run time) of the actual parameter of a call to a subprogram whose Extensions_Visible aspect is False will have no effect on the behavior of that call. In particular, if the actual parameter has any additional components which are not components of the type of the formal parameter, then these components are unreferenced by the execution of the call.]

Verification Rules

7. [SPARK typically requires that an actual parameter corresponding to an in mode or in out mode formal parameter in a call shall be fully initialized before the call; similarly, the callee is typically responsible for fully initializing any out-mode formal parameters before returning. For details (including interactions with relaxed initialization), see the verification rule about full initialization of subprogram inputs and outputs (which include parameters) in *Subprogram Declarations* and then *Relaxed Initialization*].

8. In the case of a formal parameter of a specific tagged type T (or of a private type whose full view is a specific tagged type), the set of components which shall be initialized in order to meet these requirements depends on the Extensions_Visible aspect of the callee. If the aspect is False, then that set of components is the [statically known] set of nondiscriminant components of T. If the aspect is True, then this set is the set of nondiscriminant components of the specific type associated with the tag of the corresponding actual parameter. [In general, this is not statically known. This set will always include the nondiscriminant components of T, but it may also include additional components.]

9. [To put it another way, if the applicable Extensions_Visible aspect is True, then the initialization requirements (for both the caller and the callee) for a parameter of a specific tagged type T are the same as if the formal parameter's type were T'Class. If the aspect is False, then components declared in proper descendants of T need not be initialized. In the case of an out mode parameter, such initialization by the callee is not only not required, it is effectively forbidden because such an out-mode parameter could not be fully initialized without some form of dispatching (e.g., a class-wide assignment or a dispatching call in which an out-mode parameter is a controlling operand). Such a dispatching assignment will always fully initialize its controlling out-mode parameters, regardless of the Extensions_Visible aspect of the callee. An assignment statement whose target is of a class-wide type T'Class is treated, for purposes of formal verification, like a call to a procedure with two parameters of type T'Class, one of mode out and one of mode in.]

10. [In the case of an actual parameter of a call to a subprogram whose Extensions_Visible aspect is False where the corresponding formal parameter is of a specific tagged type T, these rules imply that formal verification can safely assume that any components of the actual parameter which are not components of T will be neither read

nor written by the call.]

6.1.8 Subprogram_Variant Aspects

The aspect Subprogram_Variant may be specified for subprograms; it can be used to ensure termination of recursive subprograms in a way that is similar to how pragma Loop_Variant can be used to ensure termination of loops.

Syntax

```
subprogram_variant_list ::= structural_subprogram_variant_item | numeric_subprogram_
↪variant_items
numeric_subprogram_variant_items ::= numeric_subprogram_variant_item {, numeric_
↪subprogram_variant_item}
numeric_subprogram_variant_item ::= change_direction => expression
structural_subprogram_variant_item ::= Structural => expression
change_direction        ::= Increases | Decreases
```

The aspect_definition for a Subprogram_Variant aspect_specification shall be a subprogram_variant_list. The Subprogram_Variant aspect of an inherited subprogram for a derived type is always unspecified.

Two Subprogram_Variant aspects are said to be *compatible* if either both are structural subprogram variants or both are numeric subprogram variants, the lengths of the two `numeric_subprogram_variant_items` are equal, and corresponding pairs of the elements of the two lists agree with respect to both `change_direction` and the type of their respective `expressions`. An unspecified Subprogram_Variant aspect is compatible with, and only with, another unspecified Subprogram_Variant aspect (including itself).

Two subprograms are said to be *statically mutually recursive*, if they are mutually recursive taking into account only direct calls (that is, ignoring dispatching calls and calls through access-to-subprogram values). For example, if subprogram Aa calls Bb (that is, Aa statically contains a direct call to Bb), Bb calls Cc, Cc calls Dd, and Dd calls Aa, then any 2 of those 4 subprograms (e.g., Bb and Dd) are statically mutually recursive. The case of a direct recursive call is just a special case of a statically mutually recursive call; thus, it is possible [and not unusual] for a subprogram to be statically mutually recursive with itself and with no other subprogram.

In some cases (described in more detail below) involving a call where the calling subprogram and the called subprogram have compatible (specified) Subprogram_Variant aspects, a runtime check (or a verification condition corresponding to such a runtime check) may be be introduced to ensure that the “variant of the call progresses”. For numeric subprogram variants, this means that the values of the caller’s `expressions` (which were saved upon entry to the caller, as will be described below) are compared in textual order with those of the callee (which are evaluated only as needed as part of the check) until either a pair of unequal values is encountered or until all pairs have been compared. In either case, a check is performed that the last pair of values to be compared satisfies the following condition: if the `change_direction` for the associated `subprogram_variant_item` is Increases (respectively, Decreases) then the expression value obtained for the call is greater (respectively, less) than the value that was saved upon entry to the caller.

Static Semantics

1. [Aspect Subprogram_Variant can be used to demonstrate that execution of any of a set of statically mutually recursive subprogram(s) will not result in unbounded recursion. This is accomplished by specifying expressions that will increase or decrease at each (mutually) recursive call.]
2. Subprogram_Variant is an assertion aspect [and may be used in an Assertion_Policy pragma]. Subprogram_Variant is an assertion (as defined in Ada RM 11.4.2(1.1/3)); any Subprogram_Variant runtime checking associated with a call (see below) is governed by the Subprogram_Variant assertion policy that is in effect at the point of the call.

Legality Rules

3. A Subprogram_Variant aspect may be specified for the same subprograms that a Pre aspect may be specified for. [This implies, for example, that the Subprogram_Variant aspect cannot be specified for an abstract subprogram.]

4. The expression of a `numeric_subprogram_variant_item` shall be either of a discrete type or of a subtype of `Ada.Numerics.Big_Numbers.Big_Integers.Big_Integer`. In the second case, the associated `change_direction` shall be Decreases.

5. The expression of a `structural_subprogram_variant_item` shall denote a formal parameter of the subprogram.

6. The Subprogram_Variant assertion policy in effect at the point of a direct recursive call (i.e., a call where the calling subprogram is the same as the callee) and at the point where the subprogram is declared shall agree.

7. For purposes of the rules given in this section (including static semantics, dynamic semantics, legality rules, and verification rules), a call to an inherited subprogram associated with a derived type is treated as if the call were replaced with the equivalent call to the corresponding primitive subprogram of the parent or progenitor type described in the "Dynamic Semantics" section of Ada RM 3.4. This notional transformation is applied repeatedly in the case of multiple levels of subprogram inheritance.

Dynamic Semantics

8. At the beginning of a subprogram with a specified numeric Subprogram_Variant aspect, the `expressions` are evaluated in textual order and their values are each saved in a constant that is implicitly declared at the beginning of the subprogram body[, in the same way as for an unconditionally evaluated Old attribute reference (see Ada RM 6.1.1)]. For every expression whose type is a subtype of `Ada.Numerics.Big_Numbers.Big_Integers.Big_Integer`, a check is performed that it is non-negative.

9. For a direct recursive call (i.e., the calling subprogram is the same as the callee), if the subprogram variant is numeric, for every expression in the variant of the call whose type is a subtype of `Ada.Numerics.Big_Numbers.Big_Integers.Big_Integer`, a check is performed that it is non-negative. Then, a check is made that the variant of the call progresses (as described above). If the check fails, Assertion_Error is raised. [No runtime check is performed in the case of a direct call from one subprogram to a different subprogram, even if the two subprograms are statically mutually recursive. No runtime check is performed for a dispatching call or a call through an access-to-subprogram value.] No runtime check is performed if the Subprogram_Variant assertion policy in effect at the point of the call is not Check.

Verification Rules

10. Statically mutually recursive subprograms shall have compatible variants.

11. A statically mutually recursive call (that is, a direct call where the caller and the callee are statically mutually recursive) where the Subprogram_Variant aspects of the two subprograms are specified shall not occur with a precondition expression, within a subtype predicate expression, within a type invariant expression, within a Default_Initial_Condition expression, within a discrete_expression of a Subprogram_Variant aspect specification, or as part of the default initialization of a type. Such a call shall also not occur inside the elaboration of a package unless the package is located within a subprogram and not within a declare block.

12. For a statically mutually recursive call to a subprogram whose numeric Subprogram_Variant aspect is specified, a verification condition is introduced to ensure that the evaluation of the `expressions` of the `subprogram_variant_list` of the callee does not a raise any exception. Then, for every expression in the variant of the called subprogram whose type is a subtype of `Ada.Numerics.Big_Numbers.Big_Integers.Big_Integer`, a check is performed that it is non-negative. Finally, a verification condition is generated to ensure that the variant of the call progresses. This verification condition is already implicitly generated in the case where the caller and the callee are the same (a direct recursive call) as a consequence of the runtime check taking place in that case. It is also generated in the case of other mutually recursive calls, although no checks are introduced at runtime due to compiler implementation constraints.

13. For a statically mutually recursive call to a subprogram whose structural Subprogram_Variant aspect is specified, a verification condition is generated to ensure that the actual parameter corresponding to the formal parameter denoted by the `expression` is a path rooted either at the formal parameter of the enclosing subprogram denoted by the expression of its Subprogram_Variant aspect or at a local object of an anonymous access-to-object type

ultimately borrowing or observing a part of this formal parameter, that this path corresponds to a strict subcomponent of the structure denoted by the formal parameter of the enclosing subprogram, and that no deep parts of this structure are updated before the call. [This ensures that the rule is sufficient to prove recursion termination on acyclic data structures.]

6.1.9 Exceptional Cases

The aspect Exceptional_Cases may be specified for procedures and functions with side effects; it can be used to list exceptions that might be raised by the subprogram with side effects in the context of its precondition, and associate them with a specific postcondition. The Exceptional_Cases aspect is specified with an aspect_specification where the aspect_mark is Exceptional_Cases and the aspect_definition must follow the grammar of exceptional_case_list given below.

Syntax

```
exceptional_case_list ::= ( exceptional_case   {,  exceptional_case  })
exceptional_case      ::= exception_choice {'|' exception_choice} => consequence
```

where `consequence` is a boolean expression.

Name Resolution Rules

The boolean expression in the consequences should be resolved as regular postconditions. In particular, references to the Old attribute are allowed to occur in them.

Static Semantics

All prefixes of references to the Old attribute in exceptional cases are expected to be evaluated at the beginning of the call regardless of whether or not the particular exception is raised. This allows to introduce constants for these prefixes at the beginning of the subprogram together with the ones introduced for the regular postcondition.

Dynamic Semantics

Exceptional_Cases aspects are ignored for execution.

Legality Rules

1. Parameters of modes *out* or *in out* of the subprogram which are neither of a *by-reference* type nor marked as aliased shall only occur in the consequences of an exceptional case:
   - directly or indirectly in the prefix of a reference to the Old attribute, or
   - directly as a prefix of the Constrained, First, Last, Length, or Range attributes.

Verification Rules

2. If an exception raised in a subprogram annotated with Exceptional_Cases is not handled and causes the subprogram body to complete, then a verification condition is introduced to make sure that the consequence associated to the exceptional case covering the exception evaluates to True. [Because of the verification conditions introduced when raising unexpected exceptions, there is always an exceptional case covering the propagated exception.]

6.1.10 Always_Terminates Aspects

The aspect Always_Terminates may be specified for subprograms with side effects; it can be used to provide a condition under which the subprogram shall necessarily complete (either return normally or raise an exception). This aspect may also be specified on packages to provide a default for all subprograms with side effects declared in the package or in one of its nested packages. The Always_Terminates aspect is specified with an aspect_specification where the aspect_mark is Always_Terminates and the optional aspect_definition is a boolean expression. An Always_Terminates aspect with no aspect_definition is equivalent to an Always_Terminates aspect with an aspect_definition of True. [An execution which does not complete can for example run forever, exit the whole program using GNAT.OS_Lib.OS_Exit, or transfer the control to another execution in a non-standard way.]

Name Resolution Rules

The boolean expression in the aspect_definition should be resolved as a precondition.

Static Semantics

1. If the aspect Always_Terminates is specified for a package, it shall not have an aspect definition.
2. If the aspect Always_Terminates for a package specification or a subprogram with side effects P is not otherwise specified and P is declared directly inside a package (explicitly or implicitly) annotated with an aspect Always_Terminates then an Always_Terminates aspect of True is implicitly specified for P.

Dynamic Semantics

Always_Terminates aspects are ignored for execution.

Legality Rules

3. The Always_Terminates aspect may only be specified for the initial declaration of a subprogram with side effects (which may be a declaration, a body or a body stub), or of a package specification.

Verification Rules

4. A verification condition is introduced on loops and calls occuring inside functions without side effects or package elaborations to make sure that they necessarily complete.
5. A verification condition is introduced on loops and calls occuring inside subprograms with side effects annotated with an Always_Terminates aspect to make sure that they necessarily complete in cases where the boolean condition mentioned in the Always_Terminates aspect evaluates to True on entry of the subprogram with side effects.

6.1.11 Functions With Side Effects

The aspect Side_Effects may be specified for functions; it can be used to indicate that a function should be handled like a procedure with respect to parameter modes, Global contract, exceptional contract and termination: it may have output parameters, write global variables, raise exceptions and not terminate. Such a function is called a *function with side effects*.

Note that a function with side effects may also be a volatile function (see section *External State*).

Static Semantics

1. Side_Effects is a Boolean-valued aspect which may be specified for a noninstance function or a generic function. If directly specified, the aspect_definition shall be a static [Boolean] expression. The aspect is inherited by an inherited primitive function. If the aspect is neither inherited nor directly specified for a function, then the aspect is False.

Legality Rules

2. [Redundant: The declaration of a function with side effects may have a `parameter_specification` with a mode of **out** or **in out**. This rule also applies to a `subprogram_body` for a function with side effects for which no explicit declaration is given.]

3. [Redundant: A function with side effects may have a `mode_selector` of Output or In_Out in its Global aspect.]

4. A call to a function with side effects may only occur as the [right-hand side] expression of an assignment statement. [Redundant: In particular, functions with side effects cannot be called inside assertions.]

5. A function with side effects shall not have a Pure_Function aspect or pragma.

6. A function with side effects shall not be an expression function.

7. A function with side effects shall not be a traversal function (see section *Access Types*).

8. A user-defined primitive equality operation on a record type shall not be a function with side effects, unless the record type has only limited views (see *Overloading of Operators*).

   [This avoids the case where such a record type is a component of another composite type, whose predefined equality operation now has side effects through the primitive equality operation on its component.]

6.2 Formal Parameter Modes

In flow analysis, particularly information flow analysis, the update of a component of composite object is treated as updating the whole of the composite object with the component set to its new value and the remaining components of the composite object with their value preserved.

This means that if a formal parameter of a subprogram is a composite type and only individual components, but not all, are updated, then the mode of the formal parameter should be **in out**.

In general, it is not possible to statically determine whether all elements of an array have been updated by a subprogram if individual array elements are updated. The mode of a formal parameter of an array with such updates should be **in out**.

A formal parameter with a mode of **out** is treated as not having an entry value (apart from any discriminant or attributes of properties of the formal parameter). Hence, a subprogram cannot read a value of a formal parameter of mode **out** until the subprogram has updated it.

Verification Rules

1. A subprogram formal parameter of a composite type which is updated but not fully initialized by the subprogram shall have a mode of **in out**, unless it has relaxed initialization (see section *Relaxed Initialization*).

2. A subprogram formal parameter of mode **out** shall not be read by the subprogram until it has been updated by the subprogram. The use of a discriminant or an attribute related to a property, not its value, of the formal parameter is not considered to be a read of the formal parameter. [Examples of attributes that may be used are A'First, A'Last and A'Length; examples of attributes that are dependent on the value of the formal parameter and shall not be used are X'Old and X'Loop_Entry.]

6.3 Subprogram Bodies

6.3.1 Conformance Rules

No extensions or restrictions.

6.3.2 Inline Expansion of Subprograms

No extensions or restrictions.

6.4 Subprogram Calls

No extensions or restrictions.

6.4.1 Parameter Associations

No extensions or restrictions.

6.4.2 Anti-Aliasing

An alias is a name which refers to the same object as another name. The presence of aliasing is inconsistent with the underlying flow analysis and proof models used by the tools which assume that different names represent different entities. In general, it is not possible or is difficult to deduce that two names refer to the same object and problems arise when one of the names is used to update the object (although object renaming declarations are not problematic in SPARK).

A common place for aliasing to be introduced is through the actual parameters and between actual parameters and global variables in a call to a subprogram with side effects. Extra verification rules are given that avoid the possibility of problematic aliasing through actual parameters and global variables. Except for functions with side effects (see *Functions With Side Effects*), a function is not allowed to have side effects and cannot update an actual parameter or global variable. Therefore, such function calls cannot introduce problematic aliasing and are excluded from the anti-aliasing rules given below for calls to subprograms with side effects.

Static Semantics

1. An object is said to be *interfering* if it is unsynchronized (see section *Tasks and Synchronization*) or it is synchronized only due to being *constant after elaboration* (see section *Object Declarations*).

   Two names that potentially overlap (see section *Access Types*) and which each denotes an interfering object are said to *potentially introduce aliasing via parameter passing*. [This definition has the effect of exempting most synchronized objects from the anti-aliasing rules given below; aliasing of most synchronized objects via parameter passing is allowed.]

2. A formal parameter is said to be *immutable* if it is of mode **in** and neither of an access-to-variable type nor of an anonymous access-to-constant type. [Note that access parameters are of mode **in** too.]

   Otherwise, the formal parameter is said to be *mutable*.

Verification Rules

3. A call to a subprogram with side effects shall only pass two actual parameters which potentially introduce aliasing via parameter passing when either

  - both of the corresponding formal parameters are either
    - immutable; or
    - of mode **in** and of an anonymous access-to-constant type; or
  - at least one of the corresponding formal parameters is immutable and is of a by-copy type. [Note that this includes parameters of named access-to-constant and (named or anonymous) access-to-subprograms types. Ownership rules prevent other problematic aliasing, see section *Access Types*.]

4. If an actual parameter in a call to a subprogram with side effects and a `global_item` referenced by the called subprogram potentially introduce aliasing via parameter passing, then
   - the corresponding formal parameter shall be either
     - immutable; or
     - of mode **in** and of an anonymous access-to-constant type; and
   - if the `global_item`'s mode is Output or In_Out, then the corresponding formal parameter shall be immutable and of a by-copy type.
5. A call to a function with side effects shall only pass an actual parameter which potentially introduces aliasing via parameter passing with an object referenced from the [left-hand side] name of the enclosing assignment statement, when the corresponding formal parameter is either
   - immutable; or
   - of mode **in** and of an anonymous access-to-constant type.

   [The rationale for this rule is that, otherwise, the result of the evaluation of the assignment statement would depend on the order of evaluation chosen by the compiler, as the object assigned to might depend on this choice.]
6. A call to a function with side effects shall not reference a `global_item` of mode Output or In_Out which potentially introduces aliasing via parameter passing with an object referenced from the [left-hand side] name of the enclosing assignment statement.

   [The rationale for this rule is the same as for the previous rule.]
7. A call to a function with side effects shall not reference the symbol `@` to refer to the target name of the assignment.

   [The rationale for this rule is the same as for the previous rule.]
8. Where one of these rules prohibits the occurrence of an object V or any of its subcomponents as an actual parameter, the following constructs are also prohibited in this context:
   - A type conversion whose operand is a prohibited construct;
   - A call to an instance of Unchecked_Conversion whose operand is a prohibited construct;
   - A qualified expression whose operand is a prohibited construct;
   - A prohibited construct enclosed in parentheses.

Examples

```
procedure Anti_Aliasing is
   type Rec is record
      X : Integer;
      Y : Integer;
   end record;

   type Arr is array (1 .. 10) of Integer;

```

(continues on next page)

(continued from previous page)

```
   Local_1, Local_2 : Integer := 0;

   Rec_1 : Rec := (0, 0);

   Arr_1 : Arr := (others => 0);

   procedure One_In_One_Out (X : in Integer; Y : in out Integer)
   is
   begin
      Y := X + Y;
   end One_In_One_Out;

   procedure Two_In_Out (X, Y : in out Integer) with Global => null
   is
      Temp : Integer;
   begin
      Temp := Y;
      Y := X + Y;
      X := Temp;
   end Two_In_Out;

   procedure With_In_Global (I : in out Integer)
     with Global => Local_1
   is
   begin
      I := I + Local_1;
   end With_In_Global;

begin
   -- This is ok because parameters are by copy and there
   -- is only one out parameter
   One_In_One_Out (Local_1, Local_1);

   -- This is ok the variables do not overlap even though
   -- they are part of the same record.
   Two_In_Out (Rec_1.X, Rec_1.Y);

   -- This is ok the variables do not overlap they
   -- can statically determined to be distinct elements
   Two_In_Out (Arr_1 (1), Arr_1 (2));

   -- This is not ok because Global and formal in out parameter overlap
   With_In_Global (Local_1);

end Anti_Aliasing;
```

6.4.3 Exception Propagation

Verification Rules

1. A call to a procedure annotated with an aspect Exceptional_Cases (see *Exceptional Cases*) introduces an obligation to prove that potentially raised exceptions are expected as defined in *Raise Statements and Raise Expressions*.

6.5 Return Statements

No extensions or restrictions.

6.6 Overloading of Operators

Legality Rules

1. [A user-defined primitive equality operation on a record type shall have a Global aspect of `null`, unless the record type has only limited views; see *Global Aspects* for the statement of this rule.]
2. [A user-defined primitive equality operation on a record type shall not be a volatile function, unless the record type has only limited views; see *External State - Variables and Types* for the statement of this rule.]
3. [A user-defined primitive equality operation on a record type shall not be a function with side effects, unless the record type has only limited views; see *Functions With Side Effects* for the statement of this rule.]
4. **[A user-defined primitive equality operation on a non-ghost record type**
   shall not be ghost, unless the record type has only limited views; see *Ghost Entities* for the statement of this rule.]

6.7 Null Procedures

No extensions or restrictions.

6.8 Expression Functions

Legality Rules

1. Contract_Cases, Global and Depends aspects may be applied to an expression function as for any other function declaration if it does not have a separate declaration. If it has a separate declaration then the aspects are applied to that. It may have refined aspects applied (see *State Refinement*).

6.9 Ghost Entities

Ghost entities are intended for use in discharging verification conditions and in making it easier to express assertions about a program. The essential property of ghost entities is that they have no effect on the dynamic behavior of a valid SPARK program. More specifically, if one were to take a valid SPARK program and remove all ghost entity declarations from it (considering the association of a ghost formal parameter in a generic instantiation as a declaration) and all "innermost" statements, declarations, and pragmas which refer to those declarations (replacing removed statements with null statements when syntactically required), then the resulting program might no longer be a valid SPARK program (e.g., it might no longer be possible to discharge all of the program's verification conditions) but its dynamic semantics (when viewed as an Ada program) should be unaffected by this transformation. [This transformation might affect the performance characteristics of the program (e.g., due to no longer evaluating provably true assertions), but that is not what we are talking about here. In rare cases, it might be necessary to make a small additional change after the removals (e.g., adding an Elaborate_Body pragma) in order to avoid producing a library package that no longer needs a body (see Ada RM 7.2(4))].

Static Semantics

1. SPARK defines the Boolean-valued representation aspect Ghost. Ghost is an aspect of all entities (e.g., subprograms, types, objects). An entity whose Ghost aspect is True is said to be a ghost entity; terms such as "ghost function" or "ghost variable" are defined analogously (e.g., a function whose Ghost aspect is True is said to be a ghost function). In addition, a subcomponent of a ghost object is a ghost object.

   Ghost is an assertion aspect. [This means that Ghost can be named in an Assertion_Policy pragma.]

2. The Ghost aspect of an entity declared inside of a ghost entity (e.g., within the body of a ghost subprogram) is defined to be True. The Ghost aspect of an entity implicitly declared as part of the explicit declaration of a ghost entity (e.g., an implicitly declared subprogram associated with the declaration of a ghost type) is defined to be True. The Ghost aspect of a child of a ghost library unit is defined to be True.

3. A statement or pragma is said to be a "ghost statement" if

   - it occurs within a ghost subprogram or package; or
   - it is a call to a ghost procedure; or
   - it is an assignment statement whose target is a ghost variable; or
   - it is a pragma which specifies an aspect of a ghost entity; or
   - it is an assertion pragma which encloses a name denoting a ghost entity.

4. If the Ghost assertion policy in effect at the point of a ghost statement or the declaration of a ghost entity is Ignore, then the elaboration of that construct (at run time) has no effect, other Ada or SPARK rules notwithstanding. Similarly, the elaboration of the completion of a ghost entity has no effect if the Ghost assertion policy in effect at the point of the entity's initial declaration is Ignore. [A Ghost assertion policy of Ignore can be used to ensure that a compiler generates no code for ghost constructs.] Such a declaration is said to be a *disabled ghost declaration*; terms such as "disabled ghost type" and "disabled ghost subprogram" are defined analogously.

Legality Rules

5. The Ghost aspect may only be specified [explicitly] for the declaration of a subprogram, a generic subprogram, a type (including a partial view thereof), an object (or list of objects, in the case of an `aspect_specification` for an `object_declaration` having more than one `defining_identifier`), a package, a generic package, or a generic formal parameter. The Ghost aspect may be specified via either an `aspect_specification` or via a pragma. The representation pragma Ghost takes a single argument, a name denoting one or more entities whose Ghost aspect is then specified to be True. [In particular, SPARK does not currently include any form of ghost components of non-ghost record types, or ghost parameters of non-ghost subprograms. SPARK does define ghost state abstractions, but these are described elsewhere.]

6. A Ghost aspect value of False shall not be explicitly specified except in a confirming aspect specification. [For example, a non-ghost declaration cannot occur within a ghost subprogram.]

   The value specified for the Ghost assertion policy in an Assertion_Policy pragma shall be either Check or Ignore. [In other words, implementation-defined assertion policy values are not permitted.] The Ghost assertion policy in effect at any point of a SPARK program shall be either Check or Ignore.

7. A ghost type or object shall not be effectively volatile. A ghost object shall not be imported or exported. [In other words, no ghost objects for which reading or writing would constitute an external effect (see Ada RM 1.1.3).]

8. A ghost primitive subprogram of a non-ghost type extension shall not override an inherited non-ghost primitive subprogram. A non-ghost primitive subprogram of a type extension shall not override an inherited ghost primitive subprogram. [A ghost subprogram may be a primitive subprogram of a non-ghost tagged type. A ghost type extension may have a non-ghost parent type or progenitor; primitive subprograms of such a type may override inherited (ghost or non-ghost) subprograms.]

9. A Ghost pragma which applies to a declaration occuring in the visible part of a package shall not occur in the private part of that package. [This rule is to ensure that the ghostliness of a visible entity can be determined without having to look into the private part of the enclosing package.]

10. A ghost entity shall only be referenced:

    - from within an assertion expression; or
    - from within an aspect specification [(i.e., either an `aspect_specification` or an aspect-specifying pragma)]; or
    - within the declaration or completion of a ghost entity (e.g., from within the body of a ghost subprogram); or
    - within a ghost statement; or
    - within a `with_clause` or `use_clause`; or
    - within a renaming_declaration which either renames a ghost entity or occurs within a ghost subprogram or package; or
    - within an actual parameter in a generic instantiation when the corresponding generic formal parameter is ghost.

    A ghost attribute like `Initialized` shall only be referenced where a ghost entity would be allowed.

11. A ghost entity shall not be referenced within an aspect specification [(including an aspect-specifying pragma)] which specifies an aspect of a non-ghost entity except in the following cases:

    - the reference occurs within an assertion expression which is not a predicate expression, unless the predicate is introduced by aspect Ghost_Predicate; or
    - the specified aspect is either Global, Depends, Refined_Global, Refined_Depends, Initializes, or Refined_State. [For example, the Global aspect of a non-ghost subprogram might refer to a ghost variable.]

    [Predicate expressions are excluded because predicates participate in membership tests; no Assertion_Policy pragma has any effect on this participation. In the case of a Static_Predicate expression, there are also other reasons (e.g., case statements).]

12. An **out** or **in out** mode actual parameter in a call to a ghost subprogram shall be a ghost variable.

13. In a generic declaration:

    - the default expression (if any) for a ghost generic formal object [both of mode **in** and] of access-to-variable type shall be a ghost object [otherwise writing to a reachable part (see *Access Types*) of the ghost formal object would have an effect on a non-ghost variable]; and

- the default subprogram (if any) for a ghost generic formal procedure shall be a ghost procedure [otherwise a call to the ghost formal procedure could have effects on non-ghost variables, if the default non-ghost procedure is writing to non-ghost variables].

14. In a generic instantiation:

    - the actual parameter for a ghost generic formal object of mode **in out** or both of mode **in** and of access-to-variable type, shall be a ghost object [otherwise writing to a reachable part (see *Access Types*) of the ghost formal object would have an effect on a non-ghost variable];
    - the actual parameter for a ghost generic formal procedure shall be a ghost procedure [otherwise a call to the ghost formal procedure could have effects on non-ghost variables, if the actual non-ghost procedure is writing to non-ghost variables]; and
    - the actual parameter for a ghost generic formal package shall be a ghost package [otherwise an object or a procedure in the package could lead to the problems mentions in the two previous cases].

15. If the Ghost assertion policy in effect at the point of the declaration of a ghost entity is Ignore, then the Ghost assertion policy in effect at the point of any reference to that entity shall be Ignore. If the Ghost assertion policy in effect at the point of the declaration of a ghost variable is Check, then the Ghost assertion policy in effect at the point of any assignment to a part of that variable shall be Check. [This includes both assignment statements and passing a ghost variable as an **out** or **in out** mode actual parameter.]

16. An Assertion_Policy pragma specifying a Ghost assertion policy shall not occur within a ghost subprogram or package. If a ghost entity has a completion then the Ghost assertion policies in effect at the declaration and at the completion of the entity shall be the same. [This rule applies to subprograms, packages, types, and deferred constants.]

    The Ghost assertion policies in effect at the point of the declaration of an entity and at the point of an aspect specification which applies to that entity shall be the same.

17. The Ghost assertion policies in effect at the declaration of a state abstraction and at the declaration of each constituent of that abstraction shall be the same.

18. The Ghost assertion policies in effect at the declaration of a primitive subprogram of a ghost tagged type and at the declaration of the ghost tagged type shall be the same.

19. If a tagged type is not a disabled ghost type, and if a primitive operation of the tagged type overrides an inherited operation, then the corresponding operation of the ancestor type shall be a disabled ghost subprogram if and only if the overriding subprogram is a disabled ghost subprogram.

20. If the Ghost assertion policy in effect at the point of an a reference to a Ghost entity which occurs within an assertion expression is Ignore, then the assertion policy which governs the assertion expression (e.g., Pre for a precondition expression, Assert for the argument of an Assert pragma) shall [also] be Ignore.

21. A ghost type shall not have a task or protected part. A ghost object shall not be of a type which yields synchronized objects (see section *Tasks and Synchronization*). A ghost object shall not have a volatile part. A synchronized state abstraction shall not be a ghost state abstraction (see *Abstract_State Aspects*).

22. A user-defined primitive equality operation on a non-ghost record type shall not be ghost, unless the record type has only limited views (see *Overloading of Operators*).

    [This avoids the case where such a record type is a component of another non-ghost composite type, whose predefined non-ghost equality operation now calls a ghost function through the primitive equality operation on its component.]

Verification Rules

23. A ghost subprogram with side effects shall not have a non-ghost [global] output.

24. An output of a non-ghost subprogram other than a state abstraction or a ghost global shall not depend on a ghost input. [It is intended that this follows as a consequence of other rules. Although a non-ghost state abstraction

output which depends on a ghost input may have a non-ghost constituent, other rules prevent such a non-ghost constituent from depending on the ghost input.]

25. A global input of a ghost subprogram with side effects shall not be effectively volatile for reading. [This rule says, in effect, that ghost procedural subprograms are subject to the same restrictions as non-ghost nonvolatile functions with respect to reading volatile objects.] A name occurring within a ghost statement shall not denote an object that is effectively volatile for reading. [In other words, a ghost statement is subject to effectively the same restrictions as a ghost subprogram with side effects.]

26. If the Ghost assertion policy in effect at the point of the declaration of a ghost variable or ghost state abstraction is Check, then the Ghost assertion policy in effect at the point of any call to a procedural subprogram for which that variable or state abstraction is a global output shall be Check.

Examples

```
function A_Ghost_Expr_Function (Lo, Hi : Natural) return Natural is
  (if Lo > Integer'Last - Hi then Lo else ((Lo + Hi) / 2))
  with Pre  => Lo <= Hi,
       Post => A_Ghost_Expr_Function'Result in Lo .. Hi,
       Ghost;

function A_Ghost_Function (Lo, Hi : Natural) return Natural
  with Pre  => Lo <= Hi,
       Post => A_Ghost_Function'Result in Lo .. Hi,
       Ghost;
-- The body of the function is declared elsewhere.

function A_Nonexecutable_Ghost_Function (Lo, Hi : Natural) return Natural
  with Pre  => Lo <= Hi,
       Post => A_Nonexecutable_Ghost_Function'Result in Lo .. Hi,
       Ghost,
       Import;
-- The body of the function is not declared elsewhere.
```

6.10 Relaxed Initialization

SPARK defines the Boolean-valued aspect Relaxed_Initialization and the related Boolean-valued ghost attribute, Initialized.

Without the Relaxed_Initialization aspect, the rules that statically prevent reading an uninitialized scalar object are defined with "whole object" granularity. For example, all inputs of a subprogram are required to be fully initialized at the point of a call to the subprogram and all outputs of a subprogram are required to be fully initialized at the point of a return from the subprogram. The Relaxed_Initialization aspect, together with the Initialized attribute, provides a mechanism for safely (i.e., without introducing the possibility of improperly reading an uninitialized scalar) referencing partially initialized Inputs and Outputs.

The Relaxed_Initialization aspect may be specified for a type, for a standalone object, or (at least in effect - see below for details) for a parameter or function result of a subprogram or entry. The prefix of an Initialized attribute reference shall denote an object.

Static Semantics

1. An object is said to *have relaxed initialization* if and only if

   - its Relaxed_Initialization aspect is True; or

- the Relaxed_Initialization aspect of its type is True; or
- it is a subcomponent of an object that has relaxed initialization; or
- it is the return object of a function call and the Relaxed_Initialization aspect of the function's result is True; or
- it is the return object of a call to a predefined concatenation operator and at least one of the operands is a name denoting an object having relaxed initialization; or
- it is the result object of an aggregate having a least one component whose value is that of an object that has relaxed initialization; or
- it is the result of evaluating a value conversion whose operand has relaxed initialization; or
- it is the associated object of an expression (e.g., a view conversion, a qualified expression, or a conditional expression) which has at least one operative constituent (see Ada RM 4.4) which is not the expression itself and whose associated object has relaxed initialization.

A type has relaxed initialization if its Relaxed_Initialization aspect is True. An expression has relaxed initialization if its evaluation yields an object that has relaxed initialization.

2. A Relaxed_Initialization aspect specification for a formal parameter of a callable entity or for a function's result is expressed syntactically as an aspect_specification of the declaration of the enclosing callable entity. [This is expressed this way because Ada does not provide syntax for specifying aspects for subprogram/entry parameters, or for the result of a function.] In the following example, the parameter X1 and the result of F are specified as having relaxed initialization; the parameters X2 and X3 are not:

```
function F (X1 : T1; X2 : T2; X3 : T3) return T4
  with Relaxed_Initialization => (X1 => True, F'Result);
```

More precisely, the Relaxed_Initialization aspect for a subprogram or entry (or a generic subprogram) is specified by an `aspect_specification` where the `aspect_mark` is Relaxed_Initialization and the `aspect_definition` follows the following grammar for `profile_aspect_spec`:

```
profile_aspect_spec ::= ( profile_spec_item {, profile_spec_item} )
profile_spec_item   ::= parameter_name [=> aspect_definition]
                      | function_name'Result [=> aspect_definition]
```

3. Relaxed_Initialization aspect specifications are inherited by a derived type (if the aspect is specified for the ancestor type) and by an inherited subprogram (if the aspect is specified for the corresponding primitive subprogram of the ancestor type).

4. For a prefix *X* that denotes an object which has relaxed initialization, the following attribute is defined:

```
X'Initialized
```

[It follows as a consequence of the other rules of SPARK that if X'Initialized is True, then for every reachable part Y of X whose type is not annotated with the Relaxed_Initialization aspect, Y belongs to its subtype.] An Initialized attribute reference is never a static expression.

Legality Rules

5. The following rules apply to the profile_aspect_spec of a Relaxed_Initialization aspect specification for a subprogram, a generic subprogram, or an entry.

   - Each parameter_name shall name a parameter of the given callable entity and no parameter shall be named more than once. It is not required that every parameter be named.
   - Each aspect_definition within a profile_aspect_spec shall be as for a Boolean aspect.

- The form of profile_spec_item that includes a Result attribute reference shall only be provided if the given callable entity is a function or generic function; in that case, the prefix of the attribute reference shall denote that function or generic function. Such a Result attribute reference is allowed, other language restrictions on the use of Result attribute references notwithstanding (i.e., despite the fact that such a Result attribute reference does not occur within a postcondition expression).
- A parameter or function result named in the aspect_specification shall not be of an elementary type. [It is a bounded error to pass an uninitialized scalar parameter as input for an input parameter or as output for an output parameter or function result, so there is no benefit of marking such a parameter or result as having relaxed initialization. An object of access type is always initialized.]
- A Boolean value of True is implicitly specified if no aspect_definition is provided, as per Ada RM 13.1.1's rules for Boolean-valued aspects. A Boolean value of False is implicitly specified if a given parameter (or, in the case of a function or generic function, the result) is not mentioned in any profile_spec_item.

6. No part of a tagged type, or of a tagged object, shall have relaxed initialization.
7. No part of an effectively volatile type, or of an effectively volatile object, shall have relaxed initialization.
8. No part of an Unchecked_Union type shall have relaxed initialization. No part of the type of the prefix of an Initialized attribute reference shall be of an Unchecked_Union type.
9. A Relaxed_Initialization aspect specification which applies to a declaration occuring in the visible part of a package [(e.g., the declaration of a private type or of a deferred constant)] shall not occur in the private part of that package.
10. A formal parameter of a dispatching operation shall not have relaxed initialization; the result of a dispatching function shall not have relaxed initialization.
11. [Ghost attribute `Initialized` shall only be referenced where a ghost entity would be allowed; see *Ghost Entities* for the statement of this rule.]

Verification Rules

12. At the point of a read of an elementary object X that has relaxed initialization, a verification condition is introduced to ensure that X is initialized. This includes the case where X is a subcomponent of a composite object that is passed as an argument in a call to a predefined relational operator (e.g., "=" or "<"). Such a verification condition is also introduced in the case where X is a reachable part (see *Access Types*) of the [source] expression of an assignment operation and the target of the assignment does not have relaxed initialization, where X is a reachable part of an actual parameter in a call where the corresponding formal parameter is of mode **in** or **in out** and does not have relaxed initialization, upon a call whose precondition implies X'Initialized, and upon return from a call whose postcondition implies X'Initialized.

    [For updates to X that do not involve calls, this check that X is initialized is implemented via flow analysis and no additional annotations are required. Preconditions and postconditions that mention X'Initialized may also be used to communicate information about the initialization status of X across subprogram boundaries.

    These rules statically prevent any of the bounded-error or erroneous execution scenarios associated with reading an uninitialized scalar object described in Ada RM 13.9.1. It may provide useful intuition to think of a subprogram as having (roughly speaking) an implicit precondition of X'Initialized for each of its inputs X that does not have relaxed initialization and an implicit postcondition of Y'Initialized for each of its outputs Y that does not have relaxed initialization; this imprecise description ignores things like volatile objects and state abstractions. For a particular call, this notional precondition is also in effect for a given formal parameter if the corresponding actual parameter does not have relaxed initialization (even if the formal parameter does).

    The verification conditions described here are not needed if X does not have relaxed initialization because the more conservative whole-object-granularity rules that govern that case will ensure that X is initialized whenever it is read.]

13. For any object X, evaluation of X'Initialized includes the evaluation of any subtype predicate applying to X. In addition:

- if X has a composite type, evaluation of X'Initialized includes the evaluation of Y'Initialized for every component Y of X whose type is not annotated with the Relaxed_Initialization aspect,
- if X has unconstrained discriminants, evaluation of X'Initialized includes the evaluation of Y'Initialized for every discriminant Y of X,
- if X has an access-to-object type, evaluation of X'Initialized includes the evaluation X.all'Initialized if X is not null and the designated type of the type of X is not annotated with the Relaxed_Initialization aspect,
- if X has an elementary type, its value must have been written either explicitly or implicitly through default initialization.

Discriminants of out-mode parameters and Output globals of a subprogram are considered to be initialized at the beginning of the subprogram. Other reachable parts are not.

CHAPTER

SEVEN

PACKAGES

Verification Rules

1. The elaboration of a package shall not update, directly or indirectly, a reachable part (see *Access Types*) of a variable that is not declared immediately within the package. [Roughly speaking, this means that the outputs of the notional spec and body elaboration subprograms shall all be objects declared immediately within the package.]
2. The elaboration of a package declaration or body shall not leave any object in the Moved state unless the object was already in the Moved state at the start of that elaboration.

7.1 Package Specifications and Declarations

7.1.1 Abstraction of State

The variables declared within a package but not within a subprogram body or block which does not also enclose the given package constitute the *persistent state* of the package. A package's persistent state is divided into *visible state* and *hidden state*. If a declaration that is part of a package's persistent state is visible outside of the package, then it is a constituent of the package's visible state; otherwise it is a constituent of the package's hidden state.

Though the variables may be hidden they still form part (or all) of the persistent state of the package and the hidden state cannot be ignored. *State abstraction* is the means by which this hidden state is represented and managed. A state abstraction represents one or more declarations which are part of the hidden state of a package.

SPARK extends the concept of state abstraction to provide hierarchical data abstraction whereby the state abstraction declared in a package may contain the persistent state of other packages given certain restrictions described in *Abstract_State, Package Hierarchy and Part_Of*. This provides data refinement similar to the refinement available to types whereby a record may contain fields which are themselves records.

Static Semantics

1. The visible state of a package P consists of:
   - any variables, stand-alone constants of access-to-variable type, or *constants with variable inputs*, declared immediately within the visible part of package P; and
   - the state abstractions declared by the Abstract_State aspect specification (if any) of package P; and
   - the visible state of any packages declared immediately within the visible part of package P.
2. The hidden state of a package P consists of:
   - any variables, stand-alone constants of named access-to-variable type, or *constants with variable inputs*, declared immediately in the private part or body of P; and
   - the visible state of any packages declared immediately within the private part or body of P.

3. The preceding two rules notwithstanding, an object or state abstraction whose Part_Of aspect refers to a task or protected unit is not (directly) part of the visible state or hidden state of any package (see section *Tasks and Synchronization*).

7.1.2 External State

External state is a state abstraction or variable representing something external to a program. For instance, an input or output device, or a communication channel to another subsystem such as another SPARK program.

Updating external state might have some external effect. It could be writing a value to be read by some external device or subsystem which then has a potential effect on that device or subsystem. Similarly the value read from an external state might depend on a value provided by some external device or subsystem.

Ada uses the terms external readers and writers to describe entities external to a program which interact with the program through reading and writing data. Of particular concern to SPARK are external readers and writers which are not strictly under control of the program. It is not known precisely when a value will be written or read by an external reader or writer. These are called *asynchronous readers* and *asynchronous writers* in SPARK.

Each read or update of an external state might be significant, for instance reading or writing a stream of characters to a file, or individual reads or writes might not be significant, for instance reading a temperature from a device or writing the same value to a lamp driver or display. SPARK provides a mechanism to indicate whether a read or write is always significant.

A type is said to be *effectively volatile* if it is either a volatile type, an array type whose Volatile_Components aspect is True, an array type whose component type is effectively volatile, a record type for which all components have an effectively volatile type, a protected type, or a descendant of the type Ada.Synchronous_Task_Control.Suspension_Object.

An *effectively volatile* type is said to be *effectively volatile for reading* if it is either a volatile type with the properties Async_Writers or Effective_Reads set to True (as described below), an array type whose Volatile_Components aspect is True unless the array type has the properties Async_Writers and Effective_Reads set to False (as described below), an array type whose component type is effectively volatile for reading, a record type for which at least one component has an effectively volatile type for reading, a protected type, or a descendant of the type Ada.Synchronous_Task_Control.Suspension_Object.

A nonvolatile protected type is said to be *nonvolatile during a protected action* if none of its subcomponent types are effectively volatile. [In other words, if the only reason that the protected type is effectively volatile is because it is protected.]

An *effectively volatile object* is a volatile object, or an object of an effectively volatile type. An *effectively volatile object for reading* is a volatile object with the properties Async_Writers or Effective_Reads set to True, or an object of an effectively volatile type for reading. [An effectively volatile object for reading is also an effectively volatile object.] There are three exceptions to these rules:

- the current instance of a protected unit whose (protected) type is nonvolatile during a protected action is, by definition, not an effectively volatile object. [This exception reflects the fact that the current instance cannot be referenced in contexts where unsynchronized updates are possible. This means, for example, that the Global aspect of a nonvolatile function which is declared inside of a protected operation may reference the current instance of the protected unit.]
- a constant object associated with the evaluation of a function call, an aggregate, or a type conversion is, by definition, not an effectively volatile object. [See Ada RM 4.6 for the rules about when a type conversion introduces a new object; in cases where it is unspecified whether a new object is created, we assume (for purposes of the rules in this section) that no new object is created].
- the property No_Caching can be specified on a volatile object or on its volatile type, to express that such a variable can be analyzed as not volatile in SPARK, but that the compiler should not cache its value between accesses to the object (e.g. as a defense against fault injection). Such an object is not an effectively volatile object.

External state is an effectively volatile object or a state abstraction which represents one or more effectively volatile objects (or it could be a null state abstraction; see *Abstract_State Aspects*). [The term “external” does not necessarily mean that this state is accessed outside of the SPARK portion of the program (although it might be); it refers to the state being potentially visible to multiple tasks (as well as to the outside world), so that it is externally visible from the perspective of any one task.]

Four Boolean valued *properties* of external states that may be specified are defined:

- Async_Readers - a component of the system external to the program might read/consume a value written to an external state.
- Async_Writers - a component of the system external to the program might update the value of an external state.
- Effective_Writes - every update of the external state is significant.
- Effective_Reads - every read of the external state is significant.

These properties may be specified for an effectively volatile object as Boolean aspects or as external properties of an external state abstraction.

The Boolean aspect Volatile_Function may be specified as part of the (explicit) initial declaration of a function. A function whose Volatile_Function aspect is True is said to be a *volatile function*. Volatile functions can read effectively volatile objects for reading; nonvolatile functions cannot [but they can read other effectively volatile objects]. However note that the rule that a function must not have any output other than its result still applies; in effect this bans a volatile function from reading an object with Effective_Reads => True. As a result, calling a volatile function is considered as having an effect, and such calls are only allowed in certain contexts (see *External State - Variables and Types*). A protected function is also defined to be a *volatile function*, as is an instance of Unchecked_Conversion where one or both of the actual Source and Target types are effectively volatile types for reading. [Unlike nonvolatile functions, two calls to a volatile function with all inputs equal need not return the same result.]

A protected function whose corresponding protected type is nonvolatile during a protected action and whose Volatile_Function aspect is False is said to be *nonvolatile for internal calls*.

Legality Rules

1. If an external state is declared without any of the external properties specified then all of the external properties [i.e. except No_Caching] default to a value of True.
2. If just the name of the property is given then its value defaults to True [for instance Async_Readers defaults to Async_Readers => True].
3. A property may be explicitly given the value False [for instance Async_Readers => False].
4. If any one property is explicitly defined, all undefined properties default to a value of False.
5. The expression defining the Boolean valued property shall be static.
6. Only the following combinations of properties are valid:

Async_Readers	Async_Writers	Effective_Writes	Effective_Reads	No_Caching
True	–	True	–	–
–	True	–	True	–
True	–	–	–	–
–	True	–	–	–
True	True	True	–	–
True	True	–	True	–
True	True	–	–	–
True	True	True	True	–
–	–	–	–	True

[Another way of expressing this rule is that No_Caching is incompatible with the four external properties, that Effective_Reads can only be True if Async_Writers is True and Effective_Writes can only be True if Async_Readers is True.]

Static Semantics

7. Every update of an external state is considered to be read by some external reader if Async_Readers => True.
8. Each successive read of an external state might have a different value [written by some external writer] if Async_Writers => True.
9. If Effective_Writes => True, then every value written to the external state is significant. [For instance writing a sequence of values to a port.]
10. If Effective_Reads => True, then every value read from the external state is significant. [For example a value read from a port might be used in determining how the next value is processed.]
11. Each update of an external state has no external effect if both Async_Readers => False and Effective_Writes => False.
12. Each successive read of an external state will result in the last value explicitly written [by the program] if Async_Writers => False.
13. Every explicit update of an external state might affect the next value read from the external state even if Async_Writers => True.
14. An external state which has the property Async_Writers => True need not be initialized before being read although explicit initialization is permitted. [The external state might be initialized by an external writer.]
15. A subprogram whose Volatile_Function aspect is True shall not override an inherited primitive operation of a tagged type whose Volatile_Function aspect is False. [The reverse is allowed.]
16. A subprogram whose Side_Effects aspect is True shall not override an inherited primitive operation of a tagged type whose Side_Effects aspect is False. [The reverse is allowed.]
17. A protected object has at least the properties Async_Writers => True and Async_Readers => True. If and only if it has at least one Part_Of component with Effective_Writes => True or Effective_Reads => True, then the protected object also carries this property. [This is particularly relevant if a protected object is a constituent of an external state, or if a protected object is an input of a volatile function.]

7.1.3 External State - Variables and Types

In Ada interfacing to an external device or subsystem normally entails using one or more effectively volatile objects to ensure that writes and reads to the device are not optimized by the compiler into internal register reads and writes.

SPARK refines the specification of volatility by introducing four new Boolean aspects which may be applied only to effectively volatile objects or to volatile types. The aspects may be specified in the aspect specification of an object declaration (this effectively excludes volatile objects that are formal parameters, but allows such aspect specifications for generic formal objects) or of a type declaration (including a formal_type_declaration).

The new aspects are:

- Async_Readers - as described in *External State*.
- Async_Writers - as described in *External State*.
- Effective_Reads - as described in *External State*.
- Effective_Writes - as described in *External State*.

These four aspects are said to be the *volatility refinement* aspects. Ada's notion of volatility corresponds to the case where all four aspects are True. Specifying a volatility refinement aspect value of False for an object or type grants permission for the SPARK implementation to make additional assumptions about how the object in question (or, respectively, about how an object of the type in question) is accessed; it is the responsibility of the user to ensure that these assumptions hold. In contrast, specifying a value of True imposes no such obligation on the user.

For example, consider

```
X : Integer with Volatile, Async_Readers => True, Async_Writers => False,
                           Effective_Reads => True, Effective_Writes => True;
...
procedure Proc with ... is
  Y : Integer;
begin
  X := 0;
  Y := X;
  pragma Assert (Y = 0);
end Proc;
```

The verification condition associated with the assertion can be successfully discharged but this success depends on the Async_Writers aspect specification.

The volatility refinement aspects of types (as opposed to those of objects) are type related representation aspects. The value of a given volatility refinement aspect of a volatile type is determined as follows:

- if the aspect's value is explicitly specified, then it is the specified value;
- otherwise, if the type is a derived type whose parent type is volatile then the aspect value is inherited from the parent type;
- otherwise, if at least one other volatility refinement aspect is explicitly specified for the type then the given aspect of the type is implicitly specified to be False;
- otherwise, the given aspect of the type is implicitly specified to be True.

[This is similar to the rules for external state abstractions, except that there is no notion of inheritance in that case.]

The value of a given volatility refinement aspect of an effectively volatile object is determined as follows:

- if the object is a reachable part (see *Access Types*) of a stand-alone object or of a formal parameter but is not itself such an object, then it is the value of the given aspect of that object.
- otherwise, if the object is declared by an object declaration and the given aspect is explicitly specified for the object declaration then it is the specified value;
- otherwise, if the object is declared by an object declaration and then at least one other volatility refinement aspect is explicitly specified for the object declaration then the given aspect of the object is implicitly specified to be False;
- otherwise, it is the value of the given aspect of the type of the object.

Given two entities (each either an object or a type) E1 and E2, E1 is said to be *compatible with respect to volatility* with E2 if

- E1 is not effectively volatile; or
- both E1 and E2 are effectively volatile and each of the four volatility refinement aspects is either False for E1 or True for E2.

Legality Rules

1. Any specified value for a volatility refinement aspect shall be static.

   [If a volatility refinement aspect of a derived type is inherited from an ancestor type and has the boolean value True, the inherited value shall not be overridden to have the value False for the derived type. This follows from the corresponding Ada RM 13.1.1 rule and is stated here only to clarify the point that there is no exception to that rule for volatility refinement aspects. This is consistent with Ada's treatment of the Volatile aspect.]

2. The value of a volatility refinement aspect shall only be specified for an effectively volatile stand-alone object or for an effectively volatile type (which may be a formal type). [A formal parameter is not a stand-alone object; see Ada RM 3.3.1.] If specified for a stand-alone object, the declared object shall be compatible with respect to volatility with its type.

3. The declaration of an effectively volatile stand-alone object or type shall be a library-level declaration. [In particular, it shall not be declared within a subprogram.]

4. A discriminant or a loop parameter shall not be effectively volatile.

5. An effectively volatile type other than a protected type shall not have a discriminated part.

6. A component type of a composite type shall be compatible with respect to volatility with the composite type. Similarly, the [full view of] the designated type of a named nonderived access type shall be compatible with respect to volatility with the access type.

7. A `global_item` of a nonvolatile function, or of a function which is nonvolatile for internal calls, shall not denote either an effectively volatile object for reading or an external state abstraction which has the property Async_Writers => True or Effective_Reads => True.

8. A formal parameter (or result) of a nonvolatile function, or of a function which is nonvolatile for internal calls, shall not be of an effectively volatile type for reading. [For a protected function, this rule does not apply to the notional parameter denoting the current instance of the associated protected unit described in section *Global Aspects*.]

9. Contrary to the general SPARK rule that expression evaluation cannot have side effects, a read of an effectively volatile object for reading is considered to have a side effect. To reconcile this discrepancy, a name denoting such an object shall only occur in a *non-interfering context*. A name occurs in a non-interfering context if it is:

   - the name on the left-hand side of an assignment statement; or
   - the [right-hand side] expression of an assignment statement; or
   - the initialization expression of an object declaration which does not occur inside a declare expression; or
   - the `object_name` of an `object_renaming_declaration`; or
   - the actual parameter in a call to an instance of Unchecked_Conversion whose result is renamed [in an object renaming declaration]; or
   - an actual parameter in a call for which the corresponding formal parameter is of a non-scalar effectively volatile type for reading; or
   - the (protected) prefix of a name denoting a protected operation; or
   - the return expression of a `simple_return_statement` which applies to a volatile function; or
   - the expression of the `extended_return_object_declaration` which applies to a volatile function; or
   - the prefix of a `slice`, `selected_component`, `indexed_component`, or `attribute_reference` which is itself a name occurring in a non-interfering context; or
   - the prefix of an `attribute_reference` whose `attribute_designator` is either Address, Alignment, Component_Size, First, First_Bit, Last, Last_Bit, Length, Position, Size, or Storage_Size; or
   - the expression of a type conversion, a qualified expression or a parenthesized expression occurring in a non-interfering context; or
   - the expression in a `delay_statement`.

   [The attributes listed above all have the property that when their prefix denotes an object, evaluation of the attribute does not involve the evaluation of any part ot the object.]

The same restrictions also apply to a call to a volatile function (except not in the case of an internal call to a protected function which is nonvolatile for internal calls) and to the evaluation of any attribute which is defined to introduce an implicit dependency on a volatile state abstraction [(these are the Callable, Caller, Count, and Terminated attributes; see section *Tasks and Synchronization*)]. [An internal call to a protected function is treated like a call to a nonvolatile function if the function's Volatile_Function aspect is False.]

10. A user-defined primitive equality operation on a record type shall not be a volatile function, unless the record type has only limited views (see *Overloading of Operators*).

    [This avoids the case where such a record type is a component of another composite type, whose predefined equality operation now calls a volatile function through the primitive equality operation on its component.]

Dynamic Semantics

11. There are no dynamic semantics associated with these aspects.

Verification Rules

12. An effectively volatile for reading formal parameter of mode **out** whose Async_Writers aspect is True shall not be read, even after it has been updated.

Examples

```
with System.Storage_Elements;

package Input_Port
  with SPARK_Mode
is
   Sensor : Integer
     with Volatile,
          Async_Writers,
          Address => System.Storage_Elements.To_Address (16#ACECAF0#);
end Input_Port;
```

```
with System.Storage_Elements;

package Output_Port
  with SPARK_Mode
is
   Sensor : Integer
     with Volatile,
          Async_Readers,
          Address => System.Storage_Elements.To_Address (16#ACECAF0#);
end Output_Port;
```

7.1.4 Abstract_State Aspects

State abstraction provides a mechanism for naming, in a package's visible part, state (typically a collection of variables) that will be declared within the package's body (its hidden state). For example, a package declares a visible procedure and we wish to specify the set of global variables that the procedure reads and writes as part of the specification of the subprogram. The variables declared in the package body cannot be named directly in the package specification. Instead, we introduce a state abstraction which is visible in the package specification and later, when the package body is declared, we specify the set of variables that *constitute* or *implement* the state abstraction.

If immediately within a package body, for example, a nested package is declared, then a state abstraction of the inner package may also be part of the implementation of the given state abstraction of the outer package.

The hidden state of a package may be represented by one or more state abstractions, with each pair of state abstractions representing disjoint sets of hidden variables.

If a subprogram P with a Global aspect is declared in the visible part of a package and P reads or updates any of the hidden state of the package then the state abstractions shall be denoted by P. If P has a Depends aspect then the state abstractions shall be denoted as inputs and outputs of P, as appropriate, in the `dependency_relation` of the Depends aspect.

SPARK facilitates the specification of a hierarchy of state abstractions by allowing a single state abstraction to contain visible declarations of package declarations nested immediately within the body of a package, private child or private sibling units and descendants thereof. Each visible state abstraction or variable of a private child or descendant thereof has to be specified as being *part of* a state abstraction of its parent or a public descendant of its parent.

The Abstract_State aspect is introduced by an `aspect_specification` where the `aspect_mark` is Abstract_State and the `aspect_definition` shall follow the grammar of `abstract_state_list` given below.

Syntax

```
abstract_state_list      ::= null
                           | state_name_with_options
                           | ( state_name_with_options { , state_name_with_options } )
state_name_with_options  ::= state_name
                           | ( state_name with option_list )
option_list              ::= option { , option }
option                   ::= simple_option
                           | name_value_option
simple_option            ::= Ghost | Synchronous
name_value_option        ::= Part_Of => abstract_state
                           | External [=> external_property_list]
external_property_list   ::= external_property
                           | ( external_property {, external_property} )
external_property        ::= Async_Readers [=> expression]
                           | Async_Writers [=> expression]
                           | Effective_Writes [=> expression]
                           | Effective_Reads  [=> expression]
                           | others => expression
state_name               ::= defining_identifier
abstract_state           ::= name
```

Legality Rules

1. An `option` shall not be repeated within a single `option_list`.
2. If External is specified in an `option_list` then there shall be at most one occurrence of each of Async_Readers, Async_Writers, Effective_Writes and Effective_Reads.
3. If an `option_list` contains one or more `name_value_option` items then they shall be the final options in the list. [This eliminates the possibility of a positional association following a named association in the property list.]
4. A package_declaration or generic_package_declaration that contains a non-null Abstract_State aspect mentioned in a Part_Of specification shall have a completion (i.e., a body).

   [This rule ensures that the abstract state can have a corresponding state refinement in the body. In cases where the package does not have a completion, the abstract state has no constituents. See *State Refinement*.]

   [Ada RM 7.1's rule defining when a package "requires a completion" is unaffected by the presence of an Abstract_State aspect specification; such an aspect spec does not cause a package to "require a completion". This rule therefore implies that if an Abstract_State aspect specification occurs anywhere within the specification of a library unit package or generic package, then that library unit is going to have to contain a basic_declarative_item that requires a completion (or have an Elaborate_Body pragma) because otherwise it would be impossible to simultaneously satisfy this rule and Ada's rule that a library unit cannot have a package body unless it is required (Ada RM 7.2(4)). One could imagine a simpler rule that an Abstract_State aspect specification causes a package to "require a completion", but we want a SPARK program with its SPARK aspects removed (or ignored) to remain a legal Ada program.]

Static Semantics

5. Each `state_name` occurring in an Abstract_State aspect specification for a given package P introduces an implicit declaration of a state abstraction entity. This implicit declaration occurs at the beginning of the visible part of P. This implicit declaration shall have a completion and is overloadable.

   [The declaration of a state abstraction has the same visibility as any other declaration but a state abstraction shall only be named in contexts where this is explicitly permitted (e.g., as part of a Global aspect specification). A state abstraction is not an object; it does not have a type. The completion of a state abstraction declared in a package `aspect_specification` can only be provided as part of a Refined_State `aspect_specification` within the body of the package.]

6. A **null** `abstract_state_list` specifies that a package contains no hidden state.

7. An External state abstraction is one declared with an `option_list` that includes the External `option` (see *External State*).

8. If a state abstraction which is declared with an `option_list` that includes a Part_Of `name_value_option` whose `name` denote a state abstraction, this indicates that it is a constituent (see *State Refinement*) of the denoted state abstraction. [Alternatively, the name may denote a task or protected unit (see section *Tasks and Synchronization*).]

9. A state abstraction for which the `simple_option` Ghost is specified is said to be a ghost state abstraction. A state abstraction for which the `simple_option` Synchronous is specified is said to be a synchronized state abstraction. [The option name "Synchronous" is used instead of "Synchronized" to avoid unnecessary complications associated with the use of an Ada reserved word.] Every synchronized state abstraction is also (by definition) an external state abstraction. A synchronized state abstraction for which the `simple_option` External is not (explicitly) specified has (by definition) its Async_Readers and Async_Writers aspects specified to be True and its Effective_Writes and Effective_Reads aspects specified to be False.

Dynamic Semantics

There are no dynamic semantics associated with the Abstract_State aspect.

Verification Rules

There are no verification rules associated with the Abstract_State aspect.

Examples

```
package Simple_Abstract_State
  with Abstract_State => State          -- Declaration of abstract state named State
                                        -- representing internal state of the package.
is
   function Is_Ready return Boolean     -- Function checking some property of the State.
     with Global => State;              -- State may be used in a Global aspect.


   procedure Init                       -- Procedure to initialize the internal state of the package.
     with Global => (Output => State),  -- State may be used in a Global aspect.
          Post   => Is_Ready;
```

(continues on next page)

(continued from previous page)

```

   procedure Op (V : Integer)          -- Another procedure providing some operation on State
     with Global => (In_Out => State),
          Pre    => Is_Ready,
          Post   => Is_Ready;

end Simple_Abstract_State;
```

```
package Complex_Abstract_State
  with Abstract_State => (A,
                          B,
                          (C with External => (Async_Writers,
                                               Effective_Reads => False)))
                          --  Three abstract state names are declared A, B & C.
                          --  A and B are internal abstract states.
                          --  C is specified as external state which is an external input.
is
   procedure Init;
end Complex_Abstract_State;
```

7.1.5 Initializes Aspects

The Initializes aspect specifies the visible variables and state abstractions of a package that are initialized by the elaboration of the package. In SPARK a package shall only initialize variables declared immediately within the package.

If the initialization of a variable or state abstraction, *V*, during the elaboration of a package, *P*, is dependent on the value of a visible variable or state abstraction from another package, then this entity shall be denoted in the input list associated with *V* in the Initializes aspect of *P*.

The Initializes aspect is introduced by an `aspect_specification` where the `aspect_mark` is Initializes and the `aspect_definition` shall follow the grammar of `initialization_spec` given below.

Syntax

```
initialization_spec ::= initialization_list
                      | null

initialization_list ::= initialization_item
                      | ( initialization_item { , initialization_item } )

initialization_item ::= name [ => input_list]
```

Legality Rules

1. An Initializes aspect shall only appear in the `aspect_specification` of a `package_specification`.
2. The `name` of each `initialization_item` in the Initializes aspect definition for a package shall denote a state abstraction of the package or an entire object declared immediately within the visible part of the package. [For purposes of this rule, formal parameters of a generic package are not considered to be "declared in the package".]
3. Each `name` in the `input_list` shall denote an object, or a state abstraction but shall not denote an entity declared in the package with the `aspect_specification` containing the Initializes aspect.

4. Each entity in a single `input_list` shall be distinct. Additionally, if an entity is a state abstraction, none of its constituents shall appear as an entity in the same `input_list`.

5. An `initialization_item` with a **null** `input_list` is equivalent to the same `initialization_item` without an `input_list`. [That is Initializes => (A => **null**) is equivalent to Initializes => A.]

Static Semantics

6. The Initializes aspect of a package has visibility of the declarations occurring immediately within the visible part of the package.

7. The Initializes aspect of a package specification asserts which state abstractions and visible variables of the package are initialized by the elaboration of the package, both its specification and body, and any units which have state abstractions or variable declarations that are part (constituents) of a state abstraction declared by the package. [A package with a **null** `initialization_list`, or no Initializes aspect does not initialize any of its state abstractions or variables.]

8. An `initialization_item` shall have an `input_list` if and only if its initialization is dependent on visible variables and state abstractions not declared within the package containing the Initializes aspect. Then the `names` in the `input_list` shall denote variables and state abstractions which are used in determining the initial value of the state abstraction or variable denoted by the `name` of the `initialization_item` but are not constituents of the state abstraction.

Dynamic Semantics

There are no dynamic semantics associated with the Initializes aspect.

Verification Rules

9. If the Initializes aspect is specified for a package, then after the body (which may be implicit if the package has no explicit body) has completed its elaboration, every (entire) variable and state abstraction denoted by a `name` in the Initializes aspect shall be initialized. A state abstraction is said to be initialized if all of its constituents are initialized. An entire variable is initialized if all of its components are initialized. Other parts of the visible state of the package shall not be initialized.

10. If an `initialization_item` has an `input_list` then the variables and state abstractions denoted in the input list shall be used in determining the initialized value of the entity denoted by the `name` of the `initialization_item`.

11. All variables and state abstractions which are not declared within the package but are used in the initialization of an `initialization_item` shall appear in an `input_list` of the `initialization_item`.

12. Any `initialization_item` that is a constant shall be a *constant with variable input*. Any entity in an `input_list` that is a constant shall be a parameter or *constant with variable input*.

13. Where the refinement of a state abstraction is not visible (see *State Refinement*) and a package references one or more of its constituents, the constituents may be represented by a `global_item` that denotes the state abstraction in the `initialization_spec` of the package. [The state abstraction encapsulating a constituent is known from the Part_Of indicator on the declaration of the constituent.]

[Note: these rules allow a variable or state abstraction to be initialized by the elaboration of a package but not be denoted in an Initializes aspect. In such a case the analysis tools will treat the variable or state abstraction as uninitialized when analyzing clients of the package.]

Examples

```
package Q
  with Abstract_State => State,         -- Declaration of abstract state name State
       Initializes    => (State,        -- Indicates that State
                          Visible_Var) -- and Visible_Var will be initialized
                                        -- during the elaboration of Q.
```

(continues on next page)

(continued from previous page)

```
is
   Visible_Var : Integer;
   ...
end Q;
```

```
with Q;
package R
  with Abstract_State => S1,                   -- Declaration of abstract state name S1
       Initializes    => (S1 => Q.State,       -- Indicates that S1 will be initialized
                                               -- dependent on the value of Q.State
                          X  => Q.Visible_Var) -- and X dependent on Q.Visible_Var
                                               -- during the elaboration of R.
is
   X : Integer := Q.Visible_Var;
   ...
end R;
```

```
package Y
  with Abstract_State => (A, B, (C with External => (Async_Writers, Effective_Reads))),
       --  Three abstract state names are declared A, B & C
       Initializes    => A
       --  A is initialized during the elaboration of Y.
       --  C is specified as external state with Async_Writers
       --  and need not be explicitly initialized.
       --  B is not initialized.
is
   ...
end Y;
```

```
package Z
  with Abstract_State => A,
       Initializes    => null
       --  Package Z has an abstract state name A declared but the
       --  elaboration of Z and its private descendants do not
       --  perform any initialization during elaboration.
is
   ...
end Z;
```

7.1.6 Initial_Condition Aspects

The Initial_Condition aspect is introduced by an `aspect_specification` where the `aspect_mark` is Initial_Condition and the `aspect_definition` shall be a *Boolean*_`expression`.

Legality Rules

1. An Initial_Condition aspect shall only be placed in an `aspect_specification` of a `package_specification`.

Static Semantics

2. An Initial_Condition aspect is an assertion and behaves as a postcondition for the elaboration of both the specification and body of a package. If present on a package, then its assertion expression defines properties (a predicate)

of the state of the package which can be assumed to be true immediately following the elaboration of the package. [The expression of the Initial_Condition cannot denote a state abstraction or hidden state. This means that to express properties of hidden state, functions declared in the visible part acting on the state abstractions of the package must be used.]

Dynamic Semantics

3. With respect to dynamic semantics, specifying a given expression as the Initial_Condition aspect of a package is equivalent to specifying that expression as the argument of an Assert pragma occurring at the end of the (possibly implicit) statement list of the (possibly implicit) body of the package. [This equivalence includes all interactions with pragma Assertion_Policy but does not extend to matters of static semantics, such as name resolution.] An Initial_Condition expression does not cause freezing until the point where it is evaluated [, at which point everything that it might freeze has already been frozen].

Verification Rules

4. [The Initial_Condition aspect gives a verification condition to show that the implementation of the `package_specification` and its body satisfy the predicate given in the Initial_Condition aspect.]

5. Each variable or indirectly referenced state abstraction in an Initial_Condition aspect of a package Q which is declared immediately within the visible part of Q shall be initialized during the elaboration of Q and be denoted by a `name` of an `initialization_item` of the Initializes aspect of Q.

Examples

```
package Q
   with Abstract_State    => State,    -- Declaration of abstract state name State
        Initializes       => State,    -- State will be initialized during elaboration
        Initial_Condition => Is_Ready  -- Predicate stating the logical state after
                                       -- initialization.
is
   function Is_Ready return Boolean
      with Global => State;
end Q;
```

```
package X
   with Abstract_State    => A,      -- Declares an abstract state named A
        Initializes       => (A, B), -- A and visible variable B are initialized
                                     -- during package initialization.
        Initial_Condition => A_Is_Ready and B = 0
                                     -- The logical conditions that hold
                                     -- after package elaboration.
is
   ...
   B : Integer;

   function A_Is_Ready return Boolean
      with Global => A;
end X;
```

7.2 Package Bodies

7.2.1 State Refinement

A `state_name` declared by an Abstract_State aspect in the specification of a package shall denote an abstraction representing all or part of its hidden state. If the package has a body, the declaration must be completed in the package body by a Refined_State aspect. The Refined_State aspect defines a *refinement* for each `state_name`. The refinement shall denote the variables and subordinate state abstractions represented by the `state_name` and these are known as its *constituents*.

Constituents of each `state_name` have to be initialized consistently with that of their representative `state_name` as determined by its denotation in the Initializes aspect of the package.

A subprogram may have an *abstract view* and a *refined view*. The abstract view is a subprogram declaration in a package specification of a package where a subprogram may refer to private types and state abstractions whose details are not visible. A refined view of a subprogram is the body or body stub of the subprogram in the package body whose specification declares its abstract view.

In a refined view a subprogram has visibility of the full type declarations of any private types declared by the enclosing package and visibility of the refinements of state abstractions declared by the package. Refined versions of aspects are provided to express the contracts of a refined view of a subprogram.

7.2.2 Refined_State Aspects

The Refined_State aspect is introduced by an `aspect_specification` where the `aspect_mark` is Refined_State and the `aspect_definition` shall follow the grammar of `refinement_list` given below.

Syntax

```
refinement_list   ::= ( refinement_clause { , refinement_clause } )
refinement_clause ::= state_name => constituent_list
constituent_list  ::= null
                    | constituent
                    | ( constituent { , constituent } )
```

where

`constituent ::=` *object*`_name | state_name`

Name Resolution Rules

1. A Refined_State aspect of a `package_body` has visibility extended to the `declarative_part` of the body.

Legality Rules

2. A Refined_State aspect shall only appear in the `aspect_specification` of a `package_body`. [The use of `package_body` rather than package body allows this aspect to be specified for generic package bodies.]
3. If a `package_specification` has a non-null Abstract_State aspect its body shall have a Refined_State aspect.
4. If a `package_specification` does not have an Abstract_State aspect, then the corresponding `package_body` shall not have a Refined_State aspect.
5. Each `constituent` shall be either a variable, a constant, or a state abstraction.
6. An object which is a `constituent` shall be an entire object.

7. A `constituent` of a state abstraction of a package shall denote either an entity with no Part_Of `option` or aspect which is part of the hidden state of the package, or an entity whose declaration has a Part_Of `option` or aspect which denotes this state abstraction (see *Abstract_State, Package Hierarchy and Part_Of*).

8. Each *abstract_*`state_name` declared in the package specification shall be denoted exactly once as the `state_name` of a `refinement_clause` in the Refined_State aspect of the body of the package.

9. Every entity of the hidden state of a package shall be denoted as a `constituent` of exactly one *abstract_*`state_name` in the Refined_State aspect of the package and shall not be denoted more than once. [These `constituents` shall be either objects declared in the private part or body of the package, or the declarations from the visible part of nested packages declared immediately therein.]

10. In a package body where the refinement of a state abstraction is visible the `constituents` of the state abstraction must be denoted in aspect specifications rather than the state abstraction.

11. The legality rules related to a Refined_State aspect given in *Abstract_State, Package Hierarchy and Part_Of* also apply.

12. Each `constituent` of a ghost state abstraction shall be either a ghost variable or a ghost state abstraction. [The reverse situation (i.e., a ghost constituent of a non-ghost state abstraction) is permitted.]

13. A `constituent` of a synchronized state abstraction shall be either a synchronized object or another synchronized state abstraction. A `constituent` of a state abstraction which is neither external nor synchronized shall be not be an effectively volatile object for reading, a synchronized state abstraction, or an external state abstraction.

14. Each `constituent` of a state abstraction shall be declared before the first subprogram, package, task, or protected body, or expression_function_declaration, in the same `declarative_part`.

Static Semantics

15. A Refined_State aspect of a `package_body` completes the declaration of the state abstractions occurring in the corresponding `package_specification` and defines the objects and each subordinate state abstraction that are the `constituents` of the *abstract_*`state_names` declared in the `package_specification`.

16. A **null** `constituent_list` indicates that the named abstract state has no constituents and termed a *null_refinement*. The state abstraction does not represent any actual state at all. [This feature may be useful to minimize changes to Global and Depends aspects if it is believed that a package may have some extra state in the future, or if hidden state is removed.]

Dynamic Semantics

There are no dynamic semantics associated with Refined_State aspect.

Verification Rules

17. Each `constituent` that is a constant shall be a constant *with variable inputs*.

18. If the Async_Writers aspect of a state abstraction is True and the Async_Writers aspect of a constituent of that state abstraction is False, then after the elaboration of the (possibly implicit) body of the package which declares the abstraction, the constituent shall be initialized.

Examples

```
-- Here, we present a package Q that declares two abstract states:
package Q
   with Abstract_State => (A, B),
        Initializes    => (A, B)
is
   ...
end Q;

```

(continues on next page)

(continued from previous page)

```
-- The package body refines
--   A onto three concrete variables declared in the package body
--   B onto the abstract state of a nested package
package body Q
   with Refined_State => (A => (F, G, H),
                          B => R.State)
is
   F, G, H : Integer := 0; -- all initialized as required

   package R
      with Abstract_State => State,
           Initializes    => State -- initialized as required
   is
      ...
   end R;

   ...
end Q;
```

7.2.3 Initialization Issues

Every state abstraction specified as being initialized in the Initializes aspect of a package has to have all of its constituents initialized. This may be achieved by initialization within the package, by assumed pre-initialization (in the case of external state) or, for constituents which reside in another package, initialization by their declaring package.

Verification Rules

1. For each state abstraction denoted by the `name` of an `initialization_item` of an Initializes aspect of a package, all the `constituents` of the state abstraction must be initialized by:

   - initialization within the package; or
   - assumed pre-initialization (in the case of external states); or
   - for constituents which reside in another unit [and have a Part_Of indicator associated with their declaration (see *Abstract_State, Package Hierarchy and Part_Of*)] by their declaring package. [It follows that such constituents will appear in the initialization clause of the declaring unit unless they are external states.]

7.2.4 Refined_Global Aspects

A subprogram declared in the specification of a package may have a Refined_Global aspect applied to its body or body stub. A Refined_Global aspect of a subprogram defines a *refinement* of the Global Aspect of the subprogram; that is, the Refined_Global aspect repeats the Global aspect of the subprogram except that references to state abstractions whose refinements are visible at the point of the subprogram_body are replaced with references to [some or all of the] constituents of those abstractions. References to a state abstraction whose refinement is not visible at the point of the subprogram_body may also be similarly replaced if Part_Of aspect specifications which are visible at the point of the subprogram body identify one or more constituents of the abstraction; such a state abstraction is said to be *optionally refinable* at the point of the subprogram body.

See section *Global Aspects* regarding how the rules given in this section apply to protected operations and to task bodies.

The Refined_Global aspect is introduced by an `aspect_specification` where the `aspect_mark` is Refined_Global and the `aspect_definition` shall follow the grammar of `global_specification` in *Global Aspects*.

Static Semantics

1. The static semantics are as for those of the Global aspect given in *Global Aspects*. [Differences between these two aspects for one subprogram stem from differences in state abstraction visibility between the points where the two aspects are specified.]

Legality Rules

2. A Refined_Global aspect is permitted on a body_stub (if one is present), subprogram body, entry body, or task body if and only if the stub or body is the completion of a declaration occurring in the specification of an enclosing package, the declaration has a Global aspect which denotes a state abstraction declared by the package and either the refinement of the state abstraction is visible or a Part_Of specification specifying a constituent of the state abstraction is visible.

3. A Refined_Global aspect specification shall *refine* the subprogram's Global aspect as follows:

   a. For each `global_item` in the Global aspect which denotes a state abstraction whose non-**null** refinement is visible at the point of the Refined_Global aspect specification, the Refined_Global specification shall include one or more `global_items` which denote `constituents` of that state abstraction.

   b. For each `global_item` in the Global aspect which denotes a state abstraction whose **null** refinement is visible at the point of the Refined_Global aspect specification, there are no corresponding `global_items` in the Refined_Global specification. If this results in a Refined_Global specification with no `global_items`, then the Refined_Global specification shall include a `null_global_specification`.

   c. For each `global_item` in the Global aspect which does not denote a state abstraction whose refinement is visible and does not denote an optionally refinable state abstraction, the Refined_Global specification shall include exactly one `global_item` which denotes the same entity as the `global_item` in the Global aspect.

   d. For each `global_item` in the Global aspect which designates a state abstraction which is optionally refinable, refinement of the abstraction is optional in the following sense: either the reference to the state abstraction may be replaced with references to its constituents (following the rules of case 'a' above) or not (in which case the rules of case 'c' above apply). However, only the latter option is available if the mode of the state abstraction in the Global specification is Output.

   e. No other `global_items` shall be included in the Refined_Global aspect specification.

   f. At least one state abstraction mentioned in the Global aspect specification shall be unmentioned in the Refined_Global aspect specification. [This usually follows as a consequence of other rules, but not in some cases involving optionally refinable state abstractions where the option is declined.]

4. `Global_items` in a Refined_Global `aspect_specification` shall denote distinct entities.

5. The mode of each `global_item` in a Refined_Global aspect shall match that of the corresponding `global_item` in the Global aspect unless that corresponding `global_item` denotes a state abstraction which is not mentioned in the Refined_Global aspect. In that case, the modes of the `global_items` in the Refined_Global aspect which denote (direct or indirect) constituents of that state abstraction collectively determine (as described below) an "effective mode" for the abstraction. If there is at least one such constituent, then that "effective mode" shall match that of the corresponding `global_item` in the Global aspect; it is determined as follows:

   a. If the refinement of the abstraction is visible and every constituent of the abstraction is mentioned in the Refined_Global aspect with a mode of Output, then the effective mode is Output;

   b. Otherwise, if at least one constituent of the abstraction is mentioned in the Refined_Global aspect with a mode of Output or In_Out, then the effective mode is In_Out;

   c. Otherwise, if at least one constituent of the abstraction is mentioned in the Refined_Global aspect with a mode of Input, then the effective mode is Input;

   d. Otherwise, the effective mode is Proof_In.

[If there is no such constituent (e.g., because a *null* refinement is visible) then the mode of the state abstraction in the Global aspect plays no role in determining the legality of the Refined_Global aspect.]

6. The legality rules for *Global Aspects* and External states described in *Refined External States* also apply.

Dynamic Semantics

There are no dynamic semantics associated with a Refined_Global aspect.

Verification Rules

7. If a subprogram has a Refined_Global aspect it is used in the analysis of the subprogram body rather than its Global aspect.

8. The verification rules given for *Global Aspects* also apply.

Examples

```
package Refined_Global_Examples
  with SPARK_Mode,
       Abstract_State => State
is
   procedure P1_1 (I : in Integer)
     with Global => (In_Out => State);

   procedure P1_2 (I : in Integer)
     with Global => (In_Out => State);

   procedure P1_3 (Result : out Integer)
     with Global => (Input => State);

   procedure P1_4 (I : in Integer)
     with Global => (Output => State);

end Refined_Global_Examples;
```

```
package body Refined_Global_Examples
  with SPARK_Mode,
       Refined_State => (State => (A, B))
is
   A : Integer;  --  The constituents of State
   B : Integer;

   procedure P1_1 (I : in Integer)
     with Refined_Global => (In_Out => A,  --  Refined onto constituents of State
                             Output => B)  --  B is Output but A is In_Out and
                                           --  so Global State is also In_Out
   is
   begin
      B := A;
      A := I;
   end P1_1;

   procedure P1_2 (I : in Integer)
     with Refined_Global => (Output => A)  --  Not all of the constituents of
                                           --  State are updated and so the Global
                                           --  State must In_Out
```

(continues on next page)

(continued from previous page)

```
   is
   begin
      A := I;
   end P1_2;

   procedure P1_3 (Result : out Integer)
     with Refined_Global => (Input => B)  --  Not all of the constituents of State
                                          --  are read but none of them are
                                          --  updated so the Global State is Input
   is
   begin
      Result := B;
   end P1_3;

   procedure P1_4 (I : in Integer)
     with Refined_Global => (Output => (A, B))  --  The constituents of State are
                                                --  not read but they are all
                                                --  updated and so the mode
                                                --  selector of State is Output
   is
   begin
      A := I;
      B := A;
   end P1_4;

end Refined_Global_Examples;
```

7.2.5 Refined_Depends Aspects

A subprogram declared in the specification of a package may have a Refined_Depends aspect applied to its body or body stub. A Refined_Depends aspect of a subprogram defines a *refinement* of the Depends aspect of the subprogram; that is, the Refined_Depends aspect repeats the Depends aspect of the subprogram except that references to state abstractions, whose refinements are visible at the point of the subprogram_body, are replaced with references to [some or all of the] constituents of those abstractions.

See section *Global Aspects* regarding how the rules given in this section apply to protected operations and to task bodies.

The Refined_Depends aspect is introduced by an `aspect_specification` where the `aspect_mark` is Refined_Depends and the `aspect_definition` shall follow the grammar of `dependency_relation` in *Depends Aspects*.

Static Semantics

1. The static semantics are as for those of the Depends aspect given in *Depends Aspects*. [Differences between these two aspects for one subprogram stem from differences in state abstraction visibility between the points where the two aspects are specified.]

Legality Rules

2. A Refined_Depends aspect is permitted on a body_stub (if one is present), subprogram body, entry body, or task body if and only if the stub or body is the completion of a declaration in the specification of an enclosing package and the declaration has a Depends aspect which denotes a state abstraction declared by the package and the refinement of the state abstraction is visible.

3. A Refined_Depends aspect specification is, in effect, a copy of the corresponding Depends aspect specification except that any references in the Depends aspect to a state abstraction, whose refinement is visible at the point of the Refined_Depends specification, are replaced with references to zero or more direct or indirect constituents of that state abstraction. A Refined_Depends aspect shall have a `dependency_relation` which is derivable from the original given in the Depends aspect as follows:

   a. A *partially refined dependency relation* is created by first copying, from the Depends aspect, each `output` that is not state abstraction whose refinement is visible at the point of the Refined_Depends aspect, along with its `input_list`, to the partially refined dependency relation as an `output` denoting the same entity with an `input_list` denoting the same entities as the original. [The order of the `outputs` and the order of `inputs` within the `input_list` is insignificant.]

   b. The partially refined dependency relation is then extended by replacing each `output` in the Depends aspect that is a state abstraction, whose refinement is visible at the point of the Refined_Depends, by zero or more `outputs` in the partially refined dependency relation. It shall be zero only for a **null** refinement, otherwise all of the `outputs` shall denote a `constituent` of the state abstraction.

   c. If the `output` in the Depends aspect denotes a state abstraction which is not also an `input`, then each `constituent` of the state abstraction shall be denoted as an `output` of the partially refined dependency relation.

   d. These rules may, for each `output` in the Depends aspect, introduce more than one `output` in the partially refined dependency relation. Each of these `outputs` has an `input_list` that has zero or more of the `inputs` from the `input_list` of the original `output`. The union of these `inputs` and the original state abstraction, if it is an `input` in the `input_list`, shall denote the same `inputs` that appear in the `input_list` of the original `output`.

   e. If the Depends aspect has a `null_dependency_clause`, then the partially refined dependency relation has a `null_dependency_clause` added with an `input_list` denoting the same `inputs` as the original.

   f. The partially refined dependency relation is completed by replacing each `input` which is a state abstraction, whose refinement is visible at the point of the Refined_Depends aspect, by zero or more `inputs` which are its constituents.

   g. If a state abstraction is denoted in an `input_list` of a `dependency_clause` of the original Depends aspect and its refinement is visible at the point of the Refined_Depends aspect (derived via the process described in the rules 3a - 3f above), then:

      - at least one of its `constituents` shall be denoted as an `input` in at least one of the `dependency_clauses` of the Refined_Depends aspect corresponding to the original `dependency_clause` in the Depends aspect; or
      - at least one of its `constituents` shall be denoted in the `input_list` of a `null_dependency_clause`; or
      - the state abstraction is both an `input` and an `output` and not every `constituent` of the state abstraction is an `output` of the Refined_Depends aspect. [This rule does not exclude denoting a `constituent` of such a state abstraction in an `input_list`.]

4. These rules result in omitting each state abstraction whose **null** refinement is visible at the point of the Refined_Depends. If and only if required by the syntax, the state abstraction shall be replaced by a **null** symbol rather than being omitted.

5. No other `outputs` or `inputs` shall be included in the Refined_Depends aspect specification. `Outputs` in the Refined_Depends aspect specification shall denote distinct entities. `Inputs` in an `input_list` shall denote distinct entities.

6. [The above rules may be viewed from the perspective of checking the consistency of a Refined_Depends aspect with its corresponding Depends aspect. In this view, each `input` in the Refined_Depends aspect that is a

`constituent` of a state abstraction, whose refinement is visible at the point of the Refined_Depends aspect, is replaced by its representative state abstraction with duplicate `inputs` removed.

Each `output` in the Refined_Depends aspect, which is a `constituent` of the same state abstraction whose refinement is visible at the point of the Refined_Depends aspect, is merged along with its `input_list` into a single `dependency_clause` whose `output` denotes the state abstraction and `input_list` is the union of all of the `inputs` replaced by their encapsulating state abstraction, as described above, and the state abstraction itself if not every `constituent` of the state abstraction appears as an `output` in the Refined_Depends aspect.]

7. The rules for *Depends Aspects* also apply.

Dynamic Semantics

There are no dynamic semantics associated with a Refined_Depends aspect as it is used purely for static analysis purposes and is not executed.

Verification Rules

8. If a subprogram has a Refined_Depends aspect it is used in the analysis of the subprogram body rather than its Depends aspect.

9. The verification rules given for *Depends Aspects* also apply.

Examples

```
package Refined_Depends_Examples
  with SPARK_Mode,
       Abstract_State => State
is
   procedure P1_1 (I : in Integer)
     with Global  => (In_Out => State),
          Depends => (State =>+ I);

   procedure P1_2 (I : in Integer)
     with Global  => (In_Out => State),
          Depends => (State =>+ I);

   procedure P1_3 (Result : out Integer)
     with Global  => (Input => State),
          Depends => (Result => State);

   procedure P1_4 (I : in Integer)
     with Global  => (Output => State),
          Depends => (State => I);

end Refined_Depends_Examples;
```

```
package body Refined_Depends_Examples
  with SPARK_Mode,
       Refined_State => (State => (A, B))
is
   A : Integer;  --  The constituents of State
   B : Integer;

   procedure P1_1 (I : in Integer)
     with Refined_Global  => (In_Out => A,
                              Output => B),
```

(continues on next page)

(continued from previous page)

```
         Refined_Depends => (A => I,  --  A and B are constituents of State and
                                      --  both are outputs.
                             B => A)  --  A is dependent on I but A is also an
                                      --  input and B depends on A. Hence the
                                      --  Depends => (State =>+ I).
   is
   begin
      B := A;
      A := I;
   end P1_1;

   procedure P1_2 (I : in Integer)
     with Refined_Global  => (Output => A),
          Refined_Depends => (A => I)  --  One but not all of the constituents
                                       --  of State is updated hence the
                                       --  Depends => (State =>+ I)
   is
   begin
      A := I;
   end P1_2;

   procedure P1_3 (Result : out Integer)
     with Refined_Global  => (Input => B),
          Refined_Depends => (Result => B)  --  Not all of the constituents of
                                            --  State are read but none of them
                                            --  are updated, hence
                                            --  Depends => (Result => State)
   is
   begin
      Result := B;
   end P1_3;

   procedure P1_4 (I : in Integer)
     with Refined_Global  => (Output => (A, B)),
          Refined_Depends => ((A, B) => I)  --  The constituents of State are not
                                            --  inputs but all constituents of
                                            --  State are updated, hence,
                                            --  Depends => (State => I)
   is
   begin
      A := I;
      B := I;
   end P1_4;

end Refined_Depends_Examples;
```

7.2.6 Abstract_State, Package Hierarchy and Part_Of

In order to avoid aliasing-related problems (see *Anti-Aliasing*), SPARK must ensure that if a given piece of state (either an object or a state abstraction) is going to be a constituent of a given state abstraction, that relationship must be known at the point where the constituent is declared.

For a variable declared immediately within a package body, this is not a problem. The state refinement in which the variable is specified as a constituent precedes the declaration of the variable, and so there is no *window* between the introduction of the variable and its identification as a constituent. Similarly for a variable or state abstraction that is part of the visible state of a package that is declared immediately within the given package body.

For variable declared immediately within the private part of a package, such an unwanted window does exist (and similarly for a variable or state abstraction that is part of the visible state of a package that is declared immediately within the given private part).

In order to cope with this situation, the Part_Of aspect provides a mechanism for specifying at the point of a constituent's declaration the state abstraction to which it belongs, thereby closing the window. The state abstraction's refinement will eventually confirm this relationship. The Part_Of aspect, in effect, makes visible a preview of (some of) the state refinement that will eventually be provided in the package body.

This mechanism is also used in the case of the visible state of a private child unit (or a public descendant thereof).

The Part_Of aspect can also be used in a different way to indicate that an object or state abstraction is to be treated as though it were declared within a protected unit or task unit (see section *Tasks and Synchronization*).

Static Semantics

1. A *Part_Of indicator* is a Part_Of `option` of a state abstraction declaration in an Abstract_State aspect, a Part_Of aspect specification applied to a variable declaration or a Part_Of specification aspect applied to a generic package instantiation. The Part_Of indicator shall denote the *encapsulating* state abstraction of which the declaration is a constituent, or shall denote a task or protected unit (see section *Tasks and Synchronization*).

Legality Rules

2. A variable declared immediately within the private part of a given package or a variable or state abstraction that is part of the visible state of a package that is declared immediately within the private part of the given package shall have its Part_Of indicator specified; the Part_Of indicator shall denote a state abstraction declared by the given package.
3. A variable or state abstraction which is part of the visible state of a non-generic private child unit (or a public descendant thereof) shall have its Part_Of indicator specified; the Part_Of indicator shall denote a state abstraction declared by either the parent unit of the private unit or by a public descendant of that parent unit.
4. A Part_Of aspect specification for a package instantiation applies to each part of the visible state of the instantiation. More specifically, explicitly specifying the Part_Of aspect of a package instantiation implicitly specifies the Part_Of aspect of each part of the visible state of that instantiation. The legality rules for such an implicit specification are the same as for an explicit specification.
5. No other declarations shall have a Part_Of indicator which denotes a state abstraction. [Other declarations may have a Part_Of indicator which denotes a task or protected unit (see section *Tasks and Synchronization*).]
6. The refinement of a state abstraction denoted in a Part_Of indicator shall denote as `constituents` all of the declarations that have a Part_Of indicator denoting the state abstraction. [This might be performed once the package body has been processed.]
7. A state abstraction and a constituent (direct or indirect) thereof shall not both be denoted in one Global, Depends, Initializes, Refined_Global or Refined_Depends aspect specification. The denotation must be consistent between the Global and Depends or between Refined_Global and Refined_Depends aspects of a single subprogram.

Verification Rules

8. For flow analysis, where a state abstraction is visible as well as one or more of its `constituents`, its refinement is not visible and the Global and or Depends aspects of a subprogram denote the state abstraction, then in the implementation of the subprogram a direct or indirect

   - read of a `constituent` of the state abstraction shall be treated as a read of the encapsulating state abstraction of the `constituent`; or
   - update of a `constituent` of the state abstraction shall be treated as an update of the encapsulating state abstraction of the `constituent`. An update of such a `constituent` is regarded as updating its encapsulating state abstraction with a self dependency as it is unknown what other `constituents` the state abstraction encapsulates.

Examples

```
package P
   --  P has no state abstraction
is
   ...
end P;

--  P.Pub is the public package that declares the state abstraction
package P.Pub --  public unit
  with Abstract_State => (R, S)
is
   ...
end P.Pub;

--  State abstractions of P.Priv, A and B, plus the concrete variable X,
--  are split up among two state abstractions within P.Pub, R and S.
with P.Pub;
private package P.Priv --  private unit
  with Abstract_State => ((A with Part_Of => P.Pub.R),
                          (B with Part_Of => P.Pub.S))
is
   X : T  --  visible variable which is a constituent of P.Pub.R.
     with Part_Of => P.Pub.R;
end P.Priv;

with P.Priv; --  P.Priv has to be with'd because its state is part of
             --  the refined state.
package body P.Pub
  with Refined_State => (R => (P.Priv.A, P.Priv.X, Y),
                         S => (P.Priv.B, Z))
is
   Y : T2;  --  hidden state
   Z : T3;  --  hidden state
   ...
end P.Pub;
```

7.2.7 Refined Postcondition Aspects

A subprogram declared in the specification of a package may have a Refined_Post aspect applied to its body or body stub. The Refined_Post aspect may be used to restate a postcondition given on the declaration of a subprogram in terms of the full view of a private type or the `constituents` of a refined `state_name`.

The Refined_Post aspect is introduced by an `aspect_specification` where the `aspect_mark` is "Refined_Post" and the `aspect_definition` shall be a Boolean `expression`.

Legality Rules

1. A Refined_Post aspect may only appear on a body_stub (if one is present) or the body (if no stub is present) of a subprogram or entry which is declared in the specification of a package, its abstract view. If the initial declaration in the visible part has no explicit postcondition, a postcondition of True is assumed for the abstract view.
2. A Refined_Post aspect is an assertion. The same legality rules apply to a Refined_Post aspect as for a postcondition (a Post aspect).

Static Semantics

3. [A Refined Postcondition of a subprogram defines a *refinement* of the postcondition of the subprogram and is intended for use by callers who can see the body of the subprogram.]
4. [Logically, the Refined Postcondition of a subprogram must imply its postcondition. This means that it is perfectly logical for the declaration not to have a postcondition (which in its absence defaults to True) but for the body or body stub to have a Refined Postcondition. It also means that a caller who sees the Refined Postcondition of a subprogram will always be able to prove at least as much about the results of the call as if the usual precondition were used instead.]
5. The static semantics are otherwise as for a postcondition.

Dynamic Semantics

6. When a subprogram or entry with a Refined Postcondition is called, the Refined Postcondition is evaluated immediately before the evaluation of the postcondition or, if there is no postcondition, immediately before the point at which a postcondition would have been evaluated. If the Refined Postcondition evaluates to False, then the exception Assertion.Assertion_Error is raised. Otherwise, the postcondition is then evaluated and checked as described in the Ada RM.

Verification Rules

7. If a subprogram has both a Refined_Post aspect and a Post (and/or Post'Class) aspect, then the verification condition associated with postcondition checking is discharged in two steps.

   First, the success of the Refined_Post run-time check must be proven as usual (i.e., just like any other run-time check).

   Next, an additional proof obligation is generated which relates the Refined_Post to the Post (and Post'Class) aspects of the subprogram according to a "wrapper" model. Imagine two subprograms with the same parameter profile and Global and Depends aspects, but with different postconditions P1 and P2 (neither of these two subprograms has a Refined_Post aspect). Suppose further that the first subprogram is a "wrapper" for the second; that is, its implementation consists of nothing but a call to the second subprogram (for functions, the call would occur in a return statement). Consider the proof obligation generated for the postcondition check of that "wrapper" subprogram; roughly speaking, it is a check that P1 is implied by P2. In that sense of the word "implied", a verification condition is generated that any Post/Post'Class condition for a subprogram is implied by its Refined_Post condition. In particular, knowledge about the internals of the subprogram that was available in proving the Refined_Post condition is not available in proving this implication (just as, in the "wrapper" illustration, the internal details of the second subprogram are not available in proving the postcondition of the first).
8. If a Refined_Post aspect specification is visible at the point of a call to the subprogram, then the Refined_Post is used instead of the Postcondition aspect for purposes of formal analysis of the call. Similarly for using the

Refined_Global aspect instead of the Global aspect and the Refined_Depends aspect instead of the Depends aspect. [Roughly speaking, the "contract" associated with a call is defined by using the Refined_* aspects of the callee instead of the corresponding non-refined aspects in the case where Refined_* aspect specifications are visible.]

7.2.8 Refined External States

External state which is a state abstraction requires a refinement as does any state abstraction. There are rules which govern refinement of a state abstraction on to external states which are given in this section.

Legality Rules

1. A state abstraction that is not specified as External shall not have `constituents` which are External states.
2. An External state abstraction shall have each of the properties set to True which are True for any of its `constituents`.
3. Refined_Global aspects must respect the rules related to external properties of constituents which are external states given in *External State* and *External State - Variables and Types*.
4. All other rules for Refined_State, Refined_Global and Refined_Depends aspect also apply.

Examples

```
package Externals
  with SPARK_Mode,
       Abstract_State => ((Combined_Inputs with External => Async_Writers),
                          (Displays with External => Async_Readers),
                          (Complex_Device with External => (Async_Readers,
                                                            Effective_Writes,
                                                            Async_Writers))),
       Initializes    => Complex_Device,
       Always_Terminates
is
   procedure Read (Combined_Value : out Integer)
     with Global    => Combined_Inputs,  -- Combined_Inputs is an Input;
                                         -- it does not have Effective_Reads and
                                         -- may be an specified just as an
                                         -- Input in Global and Depends aspects.
          Depends   => (Combined_Value => Combined_Inputs);

   procedure Display (D_Main, D_Secondary : in String)
     with Global    => (Output => Displays), -- Displays is an Output and may
                                            -- be specified just as an
                                            -- Output in Global and Depends
                                            -- aspects.
          Depends   => (Displays => (D_Main, D_Secondary));

   function Last_Value_Sent return Integer
     with Volatile_Function,
          Global => Complex_Device;  -- Complex_Device is an External state.
                                     -- It does not have Effective_Reads and
                                     -- may be an specified as a global_item of
                                     -- a volatile function.

```

(continues on next page)

(continued from previous page)

```
   procedure Output_Value (Value : in Integer)
     with Global  => (In_Out => Complex_Device),
          Depends => (Complex_Device => (Complex_Device, Value));
   -- Output_Value only sends out a value if it is not the same
   -- as the last value sent.  When a value is sent it updates
   -- the saved value and has to check a status port.
   -- The subprogram must be a procedure.

end Externals;
```

```
private package Externals.Temperature
  with SPARK_Mode,
       Abstract_State => (State with External => Async_Writers,
                                     Part_Of  => Externals.Combined_Inputs),
       Always_Terminates
is
   procedure Read (Temp : out Integer)
     with Global  => State,
          Depends => (Temp => State);
end Externals.Temperature;
```

```
private package Externals.Pressure
  with SPARK_Mode,
       Abstract_State => (State with External => Async_Writers,
                                     Part_Of  => Externals.Combined_Inputs),
       Always_Terminates
is
   procedure Read (Press : out Integer)
     with Global  => State,
          Depends => (Press => State);
end Externals.Pressure;
```

```
private package Externals.Main_Display
  with SPARK_Mode,
       Abstract_State => (State with External => Async_Readers,
                                     Part_Of  => Externals.Displays),
       Always_Terminates
is
   procedure Display (Text: in String)
     with Global => (Output => State),
          Depends => (State => Text);
end Externals.Main_Display;
```

```
private package Externals.Secondary_Display
  with SPARK_Mode,
       Abstract_State => (State with External => Async_Readers,
                                     Part_Of  => Externals.Displays),
       Always_Terminates
is
   procedure Display (Text: in String)
     with Global => (Output => State),
```

(continues on next page)

(continued from previous page)

```
        Depends => (State => Text);
end Externals.Secondary_Display;
```

```
with System.Storage_Elements,
     Externals.Temperature,
     Externals.Pressure,
     Externals.Main_Display,
     Externals.Secondary_Display;

package body Externals
  with SPARK_Mode,
       Refined_State => (Combined_Inputs => (Externals.Temperature.State,
                                             Externals.Pressure.State),
                         -- Both Temperature and
                         -- Pressure are inputs only.

                         Displays => (Externals.Main_Display.State,
                                      Externals.Secondary_Display.State),
                         -- Both Main_Display and
                         -- Secondary_Display are outputs only.

                         Complex_Device => (Saved_Value,
                                            Out_Reg,
                                            In_Reg))
                         -- Complex_Device is a mixture of inputs, outputs and
                         -- non-volatile constituents.
is
   Saved_Value : Integer := 0;  -- Initialized as required.

   Out_Reg : Integer
     with Volatile,
          Async_Readers,
          Effective_Writes, -- Every value written to the port is significant.
          Address  => System.Storage_Elements.To_Address (16#ACECAFE0#);

   In_Reg : Integer
     with Volatile,
          Async_Writers,
          Address  => System.Storage_Elements.To_Address (16#A11CAFE0#);
```

7.3 Private Types and Private Extensions

No extensions or restrictions.

7.3.1 Private Operations

No extensions or restrictions.

7.3.2 Type Invariants

[Type invariants are supported in SPARK, but are subject to restrictions which imply that if a type invariant is specified for a type T, then any new verification conditions which this introduces outside of the package which defines T are trivially satisified. These restrictions ensure that any object or value of type T (or a descendant thereof) which can be named outside of that package will satisfy the invariant and so, for example, could not fail the runtime check associated with passing that object or value as a parameter in call to a procedure for which Ada requires runtime checking of the invariant (which, in turn, means that the verification condition corresponding to that runtime check is trivially satisfied). In order to accomplish this goal, verification conditions for type invariants are introduced in several contexts where Ada does not define corresponding runtime checks.]

[As a consequence of this approach, adding or deleting a type invariant for a private type should have little or no impact on users outside of the package defining the private type; on the other hand, such a change could have a great deal of impact on the verification conditions generated for the implementation of the private type and its operations.]

[Just as a reminder to the reader, text enclosed in square brackets is non-normative expository text. This is true everywhere in the SPARK RM, but there is a lot of such expository text in this section and we don't want anyone to be confused about what is strictly part of the language definition and what is not.]

Static Semantics

1. For a given type-invariant bearing type T, a *boundary subprogram* is a subprogram which is declared inside the immediate scope of type T, and visible outside the immediate scope of T.

   The point at which a generic is declared plays no role in determining whether a subprogram declared as or within an instantiation of that generic is a boundary subprogram.

Legality Rules

2. The aspect Type_Invariant may be specified in SPARK, but only for the completion of a private type. [In other words, the Type_Invariant aspect shall not be specified for a partial view of a type, nor for the completion of a private extension.] The aspect Type_Invariant'Class is not in SPARK.
3. [A Type_Invariant expression shall not have a variable input; see *Expressions* for the statement of this rule.]
4. A Type_Invariant shall not apply to an effectively volatile type for reading.

Verification Rules

In Ada RM 7.3.2, Ada defines the points at which runtime checking of type invariants is performed. In SPARK, these rules (or, more precisely, the verification conditions corresponding to these Ada dynamic semantics rules) are extended in several ways. In effect, verification conditions are generated as if Ada defined additional dynamic type invariant checking at several points (described below) where, in fact, Ada defines no such checks. [This means that when we talk below about extending invariant checks, we are only talking about generating additional verification conditions; we are not talking about any changes in a program's behavior at run-time.]

5. The type invariant expression for a type T shall not include a call to a boundary function for type T, if that boundary function has an input with a part of type T. [This often means that a type invariant expression cannot contain calls to functions declared in the visible part of the package in question.]

Ramification: It is a consequence of other rules that upon entry to a boundary subprogram for a type T, every part of every input that is of type T can be assumed to satisfy T's invariant.

6. Upon returning from a boundary subprogram for a type T, a verification condition is introduced for every part of every output that is of type T (or a descendant thereof), to ensure that this part satisfies T's invariant.
7. For every subprogram declared inside the immediate scope of type T, the preceding rule [and ramification] also apply to [any parts of] any global input or output and to [any parts of] any tagged subprogram parameter.
8. When calling a boundary subprogram for a type T or a subprogram declared outside of the immediate scope of T, a verification condition is introduced for every part of every input that is of type T (or a descendant thereof), to ensure that this part satisfies T's invariant. [This verification condition is trivially satisfied if the caller is outside of the immediate scope of T, or if the input in question is subject to rule 5 and constant for the caller. The idea here is to prevent invariant-violating values from "leaking out".]

Ramification: It is a consequence of other rules that upon return from a boundary subprogram for a type T or a subprogram declared outside of the immediate scope of T, every part of every output that is of type T (or a descendant thereof) can be assumed to satisfy T's invariant.

9. For every subprogram, the preceding rule [and ramification] also apply to [any parts of] any global input or output and to [any parts of] any tagged subprogram parameter. [The verification condition of rule 6 is trivially satisfied if the caller is outside of the immediate scope of T, or if the input in question is subject to rule 4 and constant for the caller.]
10. At the end of the elaboration of a package (i.e., at the point where the Initial_Condition, if any, is checked) a verification condition is introduced for the objects (both variables and constants) declared within the package. [If one chooses to think of package elaboration as being performed by a notional parameterless "elaboration" subprogram, then this rule (very roughly speaking) says that the global outputs of this notional subprogram follow much the same rules as for other subprograms.]
11. A Type_Invariant expression shall always terminate.

Ramification: In determining whether a dispatching call is a call to a boundary subprogram or to a subprogram declared outside of the immediate scope of T, the statically named callee is used.

Ramification: It is possible that the underlying tag of a tagged object (at runtime) may differ from the tag of its nominal (compile time) type. Suppose that an object X is (statically) of type T1 (or T1'Class) but has T2'Tag as its underlying tag, and that T2 has one or more components which are not components of T1. Ada does not define runtime checking of type invariants for such "hidden" components of parameters. The rules about tagged inputs and outputs in rules 6 and 8 are introduced in order to deal with technical difficulties that would otherwise arise in the treatment of these hidden components.

7.3.3 Default Initial Conditions

The Default_Initial_Condition aspect may be specified only as part of the aspect_specification of a `private_type_declaration`. The `aspect_definition`, if any, of such an aspect specification shall be either a null literal or a *Boolean*`_expression`.

The `aspect_definition` may be omitted; this is semantically equivalent to specifying a static *Boolean*`_expression` having the value True.

An aspect specification of "null" indicates that the partial view of the type does not define full default initialization (see *Declarations*). [The full view of the type might or might not define full default initialization.] This case has no associated dynamic semantics.

Conversely, an aspect specification of a *Boolean*`_expression` indicates that, in the partial view of the type, every part whose type is not annotated with the Relaxed_Initialization aspect defines full default initialization. This case also has dynamic semantics.

7.4 Deferred Constants

No extensions or restrictions.

7.5 Limited Types

No extensions or restrictions.

7.6 Assignment and Finalization

Legality Rules

1. Controlled types are not permitted in SPARK.

7.7 Elaboration Issues

SPARK imposes a set of restrictions which ensure that a call to a subprogram cannot occur before the body of the subprogram has been elaborated. The success of the runtime elaboration check associated with a call is guaranteed by these restrictions and so the verification condition associated with such a check is trivially discharged. Similar restrictions are imposed to prevent the reading of uninitialized library-level variables during library unit elaboration, and to prevent instantiation of a generic before its body has been elaborated. Finally, restrictions are imposed in order to ensure that the Initial_Condition (and Initializes aspect) of a library-level package can be meaningfully used.

These restrictions are described in this section. Because all of these elaboration-related issues are treated similarly, they are discussed together in one section.

Note that throughout this section an implicit call (e.g., one associated with default initialization of an object or with a defaulted parameter in a call) is treated in the same way as an explicit call, and an explicit call which is unevaluated at the point where it (textually) occurs is ignored at that point (but is not ignored later at a point where it is evaluated). This is similar to the treatment of expression evaluation in Ada's freezing rules. This same principle applies to the rules about reading global variables discussed later in this section.

Static Semantics

1. A call which occurs within the same compilation_unit as the subprogram_body of the callee is said to be an *intra-compilation_unit call*.

2. A construct (specifically, a call to a subprogram or a read or write of a variable) which occurs in elaboration code for a library-level package is said to be *executable during elaboration*. If a subprogram call is executable during elaboration and the callee's body occurs in the same compilation_unit as the call, then any constructs occurring within that body are also executable during elaboration. [If a construct is executable during elaboration, this means that it could be executed during the elaboration of the enclosing library unit and is subject to certain restrictions described below.]

   For a given library unit L1 and a given distinct library unit's spec or body L2 depending on L1 through a chain of `with_clauses`, the elaboration of the body of L1 is said to be *known to precede* the elaboration of L2 if either:

   a. L2 references L1 in an Elaborate or Elaborate_All pragma; or

   b. L1's Elaborate_Body aspect is True; or

   c. L1 does not require a body (the terminology is a little odd in this case because L1 has no body); or

   d. L1 is preelaborated and L2's library unit is not; or

e. L2 semantically depends on some library_item L3 such that the elaboration of the body of L1 is known to precede the elaboration of L3. [See Ada RM 10.1.1 for definition of semantic dependence.]

Legality Rules

3. SPARK requires that an intra-compilation_unit call which is executable during elaboration shall occur after a certain point in the unit (described below) where the subprogram's completion is known to have been elaborated. The portion of the unit following this point and extending to the start of the completion of the subprogram is defined to be the *early call region* for the subprogram. An intra-compilation_unit call which is executable during elaboration and which occurs (statically) before the start of the completion of the callee shall occur within the early call region of the callee.

4. The start of the early call region is obtained by starting at the subprogram's completion (typically a subprogram_body) and then traversing the preceding constructs in reverse elaboration order until a non-preelaborable statement/declarative_item/pragma is encountered. The early call region starts immediately after this non-preelaborable construct (or at the beginning of the enclosing block (or library unit package spec or body) if no such non-preelaborable construct is found).

   [The idea here is that once elaboration reaches the start of the early call region, there will be no further expression evaluation or statement execution (and, in particular, no further calls) before the subprogram_body has been elaborated because all elaborable constructs that will be elaborated in that interval will be preelaborable. Hence, any calls that occur statically after this point cannot occur dynamically before the elaboration of the subprogram body.]

   [These rules allow this example

```
package body Pkg is
   ...
   procedure P;
   procedure Q;
   X : Integer := Some_Function_Call; -- not preelaborable
   procedure P is ... if Blap then Q; end if; ... end P;
   procedure Q is ... if Blaq then P; end if; ... end Q;
begin
   P;
end;
```

   even though the call to Q precedes the body of Q. The early call region for either P or Q begins immediately after the declaration of X. Note that because the call to P is executable during elaboration, so is the call to Q.]

5. For purposes of the above rules, a subprogram completed by a renaming-as-body is treated as though it were a wrapper which calls the renamed subprogram (as described in Ada RM 8.5.4(7.1/1)). [The notional "call" occuring in this wrapper is then subject to the above rules, like any other call.]

6. If an instance of a generic occurs in the same compilation_unit as the body of the generic, the body must precede the instance.

   [If this rule were only needed in order to avoid elaboration check failures, a similar rule to the rule for calls could be defined. This stricter rule is used in order to avoid having to cope with use-before-definition, as in

```
generic
package G is
   ...
end G;

procedure Proc is
   package I is new G; -- expansion of I includes references to X
begin ... ; end;
```

(continues on next page)

(continued from previous page)

```
X : Integer;

package body G is
   ... <uses of X> ...
end G;
```

This stricter rule applies even if the declaration of the instantiation is not “executable during elaboration”].

7. In the case of a dispatching call, the subprogram_body mentioned in the above rules is that (if any) of the statically denoted callee.

8. The first freezing point of a tagged type shall occur within the early call region of each of its overriding primitive operations.

   [This rule is needed to prevent a dispatching call before the body of the (dynamic, not static) callee has been elaborated. The idea here is that after the freezing point it would be possible to declare an object of the type and then use it as a controlling operand in a dispatching call to a primitive operation of an ancestor type. No analysis is performed to identify scenarios where this is not the case, so conservative rules are adopted.]

   [Ada ensures that the freezing point of a tagged type will always occur after both the completion of the type and the declarations of each of its primitive subprograms; the freezing point of any type will occur before the declaration of any objects of the type or the evaluation of any expressions of the type. This is typically all that one needs to know about freezing points in order to understand how the above rule applies to a particular example.]

9. For purposes of defining the early call region, the specification and body of a library unit package whose Elaborate_Body aspect is True are treated as if they both belonged to some enclosing declaration list with the body immediately following the specification. This means that the early call region in which a call is permitted can span the specification/body boundary.

   This is important for tagged type declarations.

10. For each call that is executable during elaboration for a given library unit package spec or body, there are two cases: it is (statically) a call to a subprogram whose completion is in the current compilation_unit (or in a preelaborated unit), or it is not. In the latter case, an Elaborate_All pragma shall be provided to ensure that the given library unit spec or body will not be elaborated until after the complete semantic closure of the unit in which the (statically denoted) callee is declared.

11. For an instantiation of a generic package (excluding a bodiless generic package) which does not occur in the same compilation unit as the generic body, the same rules apply as described above for a call (i.e., an Elaborate_All pragma is required). For an instantiation of a generic subprogram which does not occur in the same compilation unit as the generic body, the same rules also apply except that only an Elaborate (as opposed to an Elaborate_All) pragma is required.

12. An implementation is permitted to accept constructs which violate the preceding rules in this section (e.g., an implementation might choose to behave, for purposes of defining an early call region, as though some non-preelaborable construct is preelaborable), but only if the implementation is able to statically ensure that accepting these constructs does not introduce the possibility of failing an elaboration check (either for a call or for an instantiation), reading an uninitialized variable, or unsafe reliance on a package’s Initial_Condition. [If an implementation chooses to take advantage of this permission, then the burden is entirely on the implementation to “get it right”.]

[These rules correctly prohibit the following example:

```
package P is
   function F return Boolean;
```

(continues on next page)

(continued from previous page)

```
   Flag : Boolean := F; -- would fail elaboration checks
end; --]
```

Examples

```
function Times_2 (X : Integer) return Integer is
begin
   return 2 * X;
end Times_2;
```

```
with Times_2;

package Intra_Unit_Elaboration_Order_Examples
  with Initializes => (X, Y)
is
   pragma Elaborate_Body;  --  Ensures body of package is elaborated
                           --  immediately after its declaration
   procedure P (I : in out Integer); --  P and hence Q are executable during
   procedure Q (J : in out Integer); --  elaboration as P is called in the
                                     --  package body

   X : Integer := Times_2 (10);  --  Not preelaborable
                                 --  The early call region begins here
                                 --  and extends into the package body because
                                 --  of the Elaborate_Body pragma.

   Y : Integer;

   procedure R (Z : in out Integer)
     with Post => Z = G (Z'Old); --  The call to G is allowed here as it is in
                                 --  the early call region

   procedure S (A : in out Integer)
     with Global => Y;           --  Global Y needs to be initialized.

   function F (I : Integer) return Integer;
   function G (J : Integer) return Integer is (2 * F (J));
   --  The call to F is allowed here as it is in
   --  early call region.
end Intra_Unit_Elaboration_Order_Examples;
```

```
package body Intra_Unit_Elaboration_Order_Examples is

   function F (I : Integer) return Integer is (I + 1);
   --  The early call region for F ends here as the body has been
   --  declared. It can now be called using normal visibility rules.

   procedure P (I : in out Integer) is
   begin
      if I > 10 then
         Q (I);  --  Q is still in the early call region and so this call is
                 --  allowed
```

(continues on next page)

(continued from previous page)

```
      end if;
   end P;
   --  The early call region for P ends here as the body has been
   --  declared. It can now be called using normal visibility rules.

   procedure Q (J : in out Integer) is
   begin
      if J > 20 then
         J := J - 10;
         P (J);  --  P can be called as its body is declared.
      end if;
   end Q;
   --  The early call region for Q ends here as the body has been
   --  declared. It can now be called using normal visibility rules.

   procedure R (Z : in out Integer) is
   begin
      Z := G (Z); --  The expression function G has been declared and
                  --  so can be called
   end R;

   procedure S (A : in out Integer) is
   begin
      A := A + Y; --  Reference to Y is ok because it is in the early call
                  --  region and the Elaborate_Body pragma ensures it is
                  --  initialized before it is used.
   end S;

begin
   Y := 42;
   P (X);   --  Call to P and hence Q during the elaboration of the package.
end Intra_Unit_Elaboration_Order_Examples;
```

```
package Inter_1 is
   function F (I : Integer) return Integer;
end Inter_1;
```

```
package body Inter_1 is
   function F (I : Integer) return Integer is (I);
end Inter_1;
```

```
package Inter_2 is
   function G (I : Integer) return Integer;
end Inter_2;
```

```
package body Inter_2 is
   function G (I : Integer) return Integer is (I);
end Inter_2;
```

```
with Inter_1;
pragma Elaborate_All (Inter_1);  --  Ensure the body of the called function F
```

(continues on next page)

(continued from previous page)

```
                                  --  has been elaborated.

package Inter_Unit_Elaboration_Examples with Elaborate_Body is
   X : Integer := Inter_1.F (10);  --  The call to F is ok because its body is
                                   --  sure to have been elaborated.
   Y : Integer;

   procedure P (I : in out Integer);  --  P is declared so that the package
                                      --  requires a body for this example.
end Inter_Unit_Elaboration_Examples;
```

```
with Inter_2;
pragma Elaborate_All (Inter_2);  --  Ensure body of called function G has
                                 --  been elaborated.

package body Inter_Unit_Elaboration_Examples is
   procedure P (I : in out Integer) is
   begin
      I := 2 * I;
   end P;
begin
   Y := Inter_2.G (20);  --  Call to G is ok because the body of
                         --  G is sure to have been elaborated.
end Inter_Unit_Elaboration_Examples;
```

7.7.1 Use of Initial_Condition and Initializes Aspects

Static Semantics

To ensure the correct semantics of the Initializes and Initial_Condition aspects, when applied to library units, language restrictions (described below) are imposed in SPARK which have the following consequences:

1. During the elaboration of a library unit package (spec or body), library-level variables declared outside of that package cannot be modified and library-level variables declared outside of that package can only be read if

   a. the variable (or its state abstraction) is mentioned in the Initializes aspect of its enclosing package (from *Initializes Aspects*); and

   b. either the variable is declared and initialized during the elaboration of the specification of its enclosing library unit package or the elaboration of the body of that library unit is known to precede the elaboration of the spec or body which reads the variable.

2. From the end of the elaboration of a library package's body to the invocation of the main program (i.e., during subsequent library unit elaboration), variables declared in the package (and constituents of state abstractions declared in the package) remain unchanged. The Initial_Condition aspect is an assertion which is checked at the end of the elaboration of a package body (but occurs textually in the package spec; see *Initial_Condition Aspects*). The initial condition of a library-level package will remain true from this point until the invocation of the main subprogram (because none of the inputs used in computing the condition can change during this interval). This means that a package's initial condition can be assumed to be true both upon entry to the main subprogram itself and during elaboration of any other unit (spec or body) whose elaboration is known to follow that of the body of the package (see preceding definition of "known to precede"; *known to follow* is, by definition, the inverse relationship). An Initial_Condition which depends on no variable inputs can also be assumed to be true throughout the execution of the main subprogram.

3. If a package's Initializes aspect mentions a state abstraction whose refinement includes constituents declared outside of that package, then the elaboration of bodies of the enclosing packages of those constituents will precede the elaboration of the body of the package declaring the abstraction (as a consequence of the rules given in *Elaboration Issues*). The idea here is that all constituents of a state abstraction whose initialization has been promised are in fact initialized by the end of the elaboration of the body of the abstraction's unit - we don't have to wait for the elaboration of other units (e.g., private children) which contribute to the abstraction.

Verification Rules

4. If a read of a variable (or state abstraction, in the case of a call to a subprogram which takes an abstraction as an input) declared in another library unit is executable during elaboration (as defined above), then either

   - the entity being read shall be a variable (i.e., not a state abstraction) and shall be initialized (perhaps by default) during the elaboration of its enclosing library unit specification; or
   - the elaboration of the compilation unit which performs the read shall be known to follow that of the body of the unit declaring the variable or state abstraction.

   In either case, the variable or state abstraction shall be specified as being initialized in the Initializes aspect of the declaring package. [This is needed to ensure that the variable has been initialized at the time of the read.]

5. If a variable is declared (immediately or not) within a library unit package specification, and if that variable is initialized (perhaps by default) during the elaboration of that specification, and if any part of that variable is also assigned to during the elaboration of the corresponding library unit package body, then that library unit's Elaborate_Body aspect shall be True. [This is needed to ensure that the variable remains unread between the elaboration of the specification and of the body of its enclosing library unit.]

6. The elaboration of a package's specification and body shall not write to a variable (or state abstraction, in the case of a call to a procedure which takes an abstraction as an output) declared outside of the package. The output associated with a read of an external state with the property Effective_Reads is permitted. [This rule applies to all packages: library-level or not, instantiations or not.] The inputs and outputs of a package's elaboration (including the elaboration of any private descendants of a library unit package) shall be as described in the Initializes aspect of the package.

Legality Rules

7. The elaboration of a package body shall be known to follow the elaboration of the body of each of the library units [(typically private children)] which provide constituents for a state abstraction denoted in the Initializes aspect of the given package.

Examples

```
package P
  with Initializes => VP
is
   pragma Elaborate_Body;  --  Needed because VP is
   VP : Integer;           --  Initialized in the body
end P;
```

```
with P;
pragma Elaborate_All (P);  --  P.VP is used in initialization of V

package Initialization_And_Elaboration
  with Abstract_State    => State,
       Initializes       => (State,
                             V => P.VP),  -- Initializing V depends on P.VP
       Initial_Condition => V = P.VP and Get_It = 0
is
```

(continues on next page)

(continued from previous page)

```
   V : Integer := P.VP;

   procedure Do_It (I : in Integer)
     with Global => (In_Out => State);

   function Get_It return Integer
     with Global => State;
end Initialization_And_Elaboration;
```

```
private package Initialization_And_Elaboration.Private_Child
  with Abstract_State    => (State with Part_Of =>
                               Initialization_And_Elaboration.State),
       Initializes       => State,
       Initial_Condition => Get_Something = 0
is
   procedure Do_Something (I : in Integer)
     with Global   => (In_Out => State),
          Always_Terminates;

   function Get_Something return Integer
     with Global => State;
end Initialization_And_Elaboration.Private_Child;
```

```
with Initialization_And_Elaboration.Private_Child;
pragma Elaborate (Initialization_And_Elaboration.Private_Child);
--  pragma Elaborate for the private child is required because it is a
--  constituent of the state abstraction
--  Initialization_And_Elaboration.State, which is mentioned in the
--  Initializes aspect of the package.

package body Initialization_And_Elaboration
  with Refined_State => (State => Private_Child.State)
  --  State is initialized
  --  Private child must be elaborated.
is
   procedure Do_It (I : in Integer)
     with Refined_Global => (In_Out => Private_Child.State)
   is
   begin
      Private_Child.Do_Something (I);
   end Do_It;

   function Get_It return Integer
     with Refined_Global => Private_Child.State
   is
   begin
      return Private_Child.Get_Something;
   end Get_It;
end Initialization_And_Elaboration;
```

CHAPTER

EIGHT

VISIBILITY RULES

8.1 Declarative Region

No extensions or restrictions.

8.2 Scope of Declarations

No extensions or restrictions.

8.3 Visibility

No extensions or restrictions.

8.3.1 Overriding Indicators

No extensions or restrictions.

8.4 Use Clauses

Legality Rules

1. Use clauses are always in SPARK, even if the unit mentioned is not completely in SPARK.

8.5 Renaming Declarations

8.5.1 Object Renaming Declarations

Legality Rules

1. [An expression or range occurring as part of an `object_renaming_declaration` shall not have a variable input; similarly, the access-valued prefix of a dereference occurring as part of an object_renaming declaration shall not have a variable input. See *Expressions* for the statement of this rule.] [The first part of this rule can apply to an index of an indexed_component and the range of a slice.]

8.5.2 Exception Renaming Declarations

No extensions or restrictions.

8.5.3 Package Renaming Declarations

No extensions or restrictions.

8.5.4 Subprogram Renaming Declarations

From the point of view of both static and dynamic verification, a *renaming-as-body* is treated as a one-line subprogram that "calls through" to the renamed unit.

Legality Rules

1. The `aspect_specification` on a `subprogram_renaming_declaration` shall not include any of the SPARK-defined aspects introduced in this document.

8.5.5 Generic Renaming Declarations

No extensions or restrictions.

8.6 The Context of Overload Resolution

No extensions or restrictions.

CHAPTER

NINE

TASKS AND SYNCHRONIZATION

Tasks and protected types are in SPARK, but are subject to the restrictions of the Ravenscar profile or the more permissive Jorvik profile (see Ada RM D.13). In particular, task entry declarations are never in SPARK.

Tasks may communicate with each other via synchronized objects; these include protected objects, suspension objects, atomic objects, constants, and "constant after elaboration" objects (described later).

Other objects are said to be unsynchronized and may only be referenced (directly or via intermediate calls) by a single task (including the environment task) or by the protected operations of a single protected object.

These rules statically eliminate the possibility of erroneous concurrent access to shared data (i.e., "data races").

Tagged task types, tagged protected types, and the various forms of synchronized interface types are in SPARK. Subject to the restrictions of Ravenscar or Jorvik, delay statements and protected procedure handlers are in SPARK. The attributes Callable, Caller, Identity and Terminated are in SPARK.

Static Semantics

1. A type is said to *yield synchronized objects* if it is

   - a task type; or
   - a protected type; or
   - a synchronized interface type; or
   - an array type whose element type yields synchronized objects; or
   - a record type or type extension whose discriminants, if any, lack default values, which has at least one nondiscriminant component (possibly inherited), and all of whose nondiscriminant component types yield synchronized objects; or
   - a descendant of the type Ada.Synchronous_Task_Control.Suspension_Object; or
   - a private type whose completion yields synchronized objects.

   An object is said to be *synchronized* if it is

   - of a type which yields synchronized objects; or
   - an atomic object whose Async_Writers aspect is True; or
   - a variable which is "constant after elaboration" (see section *Object Declarations*); or
   - a constant not of access-to-variable type.

   [Synchronized objects may be referenced by multiple tasks without causing erroneous execution. The declaration of a synchronized stand-alone variable shall be a library-level declaration.]

Legality Rules

2. Task and protected units are in SPARK, but their use requires the Ravenscar or Jorvik profile. [In other words, a task or protected unit is not in SPARK if neither the Ravenscar profile nor the Jorvik profile apply to the enclosing compilation unit.] Similarly, the use of task or protected units also requires a Partition_Elaboration_Policy of Sequential. [This is to prevent data races during library unit elaboration.] Similarly, the use of any subprogram which references the predefined state abstraction Ada.Task_Identification.Tasking_State (described below) as a global requires the Ravenscar or Jorvik profile.

3. If the declaration of a variable or a package which declares a state abstraction follows (within the same immediately enclosing declarative region) a `single_task_declaration` or a `single_protected_declaration`, then the Part_Of aspect of the variable or state abstraction may denote the task or protected unit. This indicates that the object or state abstraction is not part of the visible state or private state of its enclosing package. [Loosely speaking, flow analysis will treat the object as though it were declared within its "owner". This can be useful if, for example, a protected object's operations need to reference an object whose Address aspect is specified. The protected (as opposed to task) case corresponds to the previous notion of "virtual protected elements" in RavenSPARK.]

   An object or state abstraction which "belongs" to a task unit in this way is treated as a local object of the task (e.g., it cannot be named in a Global aspect specification occurring outside of the body of the task unit, just as an object declared immediately within the task body could not be). An object or state abstraction which "belongs" to a protected unit in this way is treated as a component of the (anonymous) protected type (e.g., it can never be named in any Global aspect specification, just as a protected component could not be). [There is one obscure exception to these rules, described in the next paragraph: a subprogram which is declared within the statement list of the body of the immediately enclosing package (this is possible via a block statement).]

   Any name denoting such an object or state abstraction shall occur within either

   - the body of the "owning" task or protected unit; or
   - the statement list of the object's immediately enclosing package; or
   - an Initializes or Initial_Condition aspect specification for the object's immediately enclosing package.

   [Roughly speaking, such an object can only be referenced from within the "owning" unit or during the execution of the statement list of its enclosing package].

   The notional equivalences described above break down in the case of package elaboration. The presence or absence of such a Part_Of aspect specification is ignored in determining the legality of an Initializes or Initial_Condition aspect specification. [Very roughly speaking, the restrictions implied by such a Part_Of aspect specification are not really "in effect" during library unit elaboration; or at least that's one way to view it. For example such an object can be accessed from within the elaboration code of its immediately enclosing package. On the other hand, it could not be accessed from within a subprogram unless the subprogram is declared within either the task unit body in question (in the task case) or within the statement list of the body of the immediately enclosing package (in either the task or the protected case).]

4. A protected type shall define full default initialization. A variable whose Part_Of aspect specifies a task unit or protected unit shall be of a type which defines full default initialization, or shall be declared with an initial value expression, or shall be imported.

5. A type which does not yield synchronized objects shall not have a component type which yields synchronized objects. [Roughly speaking, no mixing of synchronized and unsynchronized component types.]

6. A constituent of a synchronized state abstraction shall be a synchronized object or a synchronized state abstraction.

7. [The expression of a Priority aspect specification shall not have a variable input; see *Expressions* for the statement of this rule.]

Verification Rules

8. A `global_item` occurring in a Global aspect specification of a task unit or of a protected operation shall not denote an object or state abstraction which is not synchronized.

9. A `global_item` occurring in the Global aspect specification of the main subprogram shall not denote an object or state abstraction whose Part_Of aspect denotes a task or protected unit. [In other words, the environment task cannot reference objects which "belong" to other tasks.]

10. A state abstraction whose Part_Of aspect specifies a task unit or protected unit shall be named in the Initializes aspect of its enclosing package.

11. The precondition of a protected operation shall not reference a global variable, unless it is *constant after elaboration*.

12. The Ravenscar profile includes "Max_Entry_Queue_Length => 1" and "Max_Protected_Entries => 1" restrictions. The Jorvik profile does not, but does allow use of pragma Max_Queue_Length to specify the maximum entry queue length for a particular entry. If the maximum queue length for some given entry of some given protected object is specified (via either mechanism) to have the value N, then at most N distinct tasks (including the environment task) shall ever call (directly or via intermediate calls) the given entry of the given protected object. [Roughly speaking, each such protected entry can be statically identified with a set of at most N "caller tasks" and no task outside that set shall call the entry. This rule is enforced via (potentially conservative) flow analysis, as opposed to by introducing verification conditions.]

    For purposes of this rule, Ada.Synchronous_Task_Control.Suspension_Object is assumed to be a protected type having one entry and the procedure Suspend_Until_True is assumed to contain a call to the entry of its parameter. [This rule discharges the verification condition associated with the Ada rule that two tasks cannot simultaneously suspend on one suspension object (see Ada RM D.10(10)).]

13. The verification condition associated with the Ada rule that it is a bounded error to invoke an operation that is potentially blocking (including due to cyclic locking) during a protected action (see Ada RM 9.5.1(8)) is discharged via (potentially conservative) flow analysis, as opposed to by introducing verification conditions. [Support for the "Potentially_Blocking" aspect discussed in AI12-0064 may be incorporated into SPARK at some point in the future.]

    The verification condition associated with the Ada rule that it is a bounded error to call the Current_Task function from an entry_body, or an interrupt handler (see Ada RM C.7.1(17/3)) is discharged similarly.

    The verification condition associated with the Ada rule that the active priority of a caller of a protected operation is not higher than the ceiling of the corresponding protected object (see Ada RM D.3(13)) is dependent on (potentially conservative) flow analysis. This flow analysis is used to determine which tasks potentially call (directly or indirectly) a protected operation of which protected objects, and similarly which protected objects have protected operations that potentially perform calls (directly or indirectly) on the operations of other protected objects. A verification condition is created for each combination of potential (task or protected object) caller and called protected object to ensure that the (task or ceiling) priority of the potential caller is no greater than the ceiling priority of the called protected object.

14. The end of a task body shall not be reachable. [This follows from from Ravenscar's or Jorvik's No_Task_Termination restriction.]

15. A nonvolatile function shall not be potentially blocking. [Strictly speaking this rule is already implied by other rules of SPARK, notably the rule that a nonvolatile function cannot depend on a volatile input.] [A dispatching call which statically denotes a primitive subprogram of a tagged type T is a potentially blocking operation if the corresponding primitive operation of any descendant of T is potentially blocking.]

16. The package Ada.Task_Identification declares (and initializes) a synchronized external state abstraction named Tasking_State. The packages Ada.Real_Time and Ada.Calendar declare (and initialize) synchronized external state abstractions named Clock_Time. The Async_Readers and Async_Writers aspects of all those state abstractions are True, and their Effective_Reads and Effective_Writes aspects are False. Each is listed in the Initializes aspect of its respective package. For each of the following language-defined functions, the Volatile_Function aspect of the function is defined to be True and the Global aspect of the function specifies that one of these two state abstractions is referenced as an Input global:

    - Ada.Real_Time.Clock references Ada.Real_Time.Clock_Time;

- Ada.Execution_Time.Clock references Ada.Real_Time.Clock_Time;
- Ada.Execution_Time.Clock_For_Interrupts references Ada.Real_Time.Clock_Time;
- Ada.Execution_Time.Interrupts.Clock references Ada.Real_Time.Clock_Time;
- Ada.Calendar.Clock (which is excluded by the Ravenscar profile but not by the Jorvik profile) references Ada.Calendar.Clock_Time;
- Ada.Task_Identification.Current_Task references Ada.Task_Identification.Tasking_State;
- Ada.Task_Identification.Is_Terminated references Ada.Task_Identification.Tasking_State;
- Ada.Task_Identification.Is_Callable references Ada.Task_Identification.Tasking_State;
- Ada.Task_Identification.Activation_Is_Complete references Ada.Task_Identification.Tasking_State;
- Ada.Dispatching.EDF.Get_Deadline references Ada.Task_Identification.Tasking_State;
- Ada.Interrupts.Is_Reserved references Ada.Task_Identification.Tasking_State;
- Ada.Interrupts.Is_Attached references Ada.Task_Identification.Tasking_State;
- Ada.Interrupts.Detach_Handler references Ada.Task_Identification.Tasking_State;
- Ada.Interrupts.Get_CPU references Ada.Task_Identification.Tasking_State;
- Ada.Synchronous_Task_Control.Current_State references Ada.Task_Identification.Tasking_State.

[Functions excluded by the Jorvik profile (and therefore also by the Ravenscar profile) are not on this list.]

17. For each of the following language-defined procedures, the Global aspect of the procedure specifies that the state abstraction Ada.Task_Identification.Tasking_State is referenced as an In_Out global:

- Ada.Interrupts.Detach_Handler;
- Ada.Dispatching.Yield.

18. For purposes of determining global inputs and outputs, a delay statement is considered to reference the state abstraction Ada.Real_Time.Clock_Time as an input. [In other words, a delay statement can be treated like a call to a procedure which takes the delay expression as an actual parameter and references the Clock_Time state abstraction as an Input global.]

19. For purposes of determining global inputs and outputs, a use of any of the Callable, Caller, Count, or Terminated attributes is considered to reference the state abstraction Ada.Task_Identification.Tasking_State as an Input. [In other words, evaluation of one of these attributes can be treated like a call to a volatile function which takes the attribute prefix as a parameter (in the case where the prefix denotes an object or value) and references the Tasking_State state abstraction as an Input global.] [On the other hand, use of the Identity or Storage_Size attributes introduces no such dependency.]

20. Preconditions are added to subprogram specifications as needed in order to avoid the failure of language-defined runtime checks for the following subprograms:

- for Ada.Execution_Time.Clock, T does not equal Task_Identification.Null_Task_Id.
- for Ada.Execution_Time.Clock_For_Interrupts, Interrupt_Clocks_Supported is True.
- for Ada.Execution_Time.Interrupts.Clock, Separate_Interrupt_Clocks_Supported is True.
- for Ada.Execution_Time's arithmetic and conversion operators (including Time_Of), preconditions are defined to ensure that the result belongs to the result type.
- for Ada.Real_Time's arithmetic and conversion operators (including Time_Of), preconditions are defined to ensure that the result belongs to the result type.

21. All procedures declared in the visible part of Ada.Synchronous_Task_Control have a dependency "(S => null)" despite the fact that S has mode **in out**.

CHAPTER

TEN

PROGRAM STRUCTURE AND COMPILATION ISSUES

SPARK supports constructive, modular analysis. This means that analysis may be performed before a program is complete based on unit interfaces. For instance, to analyze a subprogram which calls another all that is required is a specification of the called subprogram including, at least, its `global_specification` and if formal verification of the calling program is to be performed, then the Pre and Postcondition of the called subprogram need to be provided. The body of the called subprogram does not need to be implemented to analyze the caller. The body of the called subprogram is checked to be conformant with its specification when its implementation code is available and analyzed.

The separate compilation of Ada `compilation_units` is consistent with SPARK modular analysis except where noted in the following subsections but, particularly with respect to incomplete programs, analysis does not involve the execution of the program.

10.1 Separate Compilation

Legality Rules

1. A program unit cannot be a task unit, a protected unit or a protected entry.

10.1.1 Compilation Units - Library Units

No restrictions or extensions.

10.1.2 Context Clauses - With Clauses

Legality Rules

1. With clauses are always in SPARK, even if the unit mentioned is not completely in SPARK.

Abstract Views

State abstractions are visible in the limited view of packages in SPARK. The notion of an *abstract view* of an object declaration is also introduced, and the limited view of a package includes the abstract view of any objects declared in the visible part of that package. The only allowed uses of an abstract view of an object are where the use of a state abstraction would be allowed (for example, in a Global `aspect_specification`).

Legality Rules

2. A name denoting the abstract view of an object shall occur only:

   a. as a `global_item` in a Global or Refined_Global aspect specification; or

b. as an `input` or `output` in a Depends or Refined_Depends aspect specification; or

c. in an `input_list` of an Initializes aspect.

Static Semantics

3. Any state abstractions declared within a given package are present in the limited view of the package. [This means that, for example, a Global `aspect_specification` for a subprogram declared in a library unit package *P1* could refer to a state abstraction declared in a package *P2* if *P1* has a limited with of *P2*.]

4. For every object declared by an `object_declaration` occurring immediately within the visible part of a given package, the limited view of the package contains an *abstract view* of the object.

10.1.3 Subunits of Compilation Units

No restrictions or extensions.

10.1.4 The Compilation Process

The analysis process in SPARK is similar to the compilation process in Ada except that the `compilation_units` are analyzed, that is flow analysis and formal verification is performed, rather than compiled.

10.1.5 Pragmas and Program Units

No restrictions or extensions.

10.1.6 Environment-Level Visibility Rules

No restrictions or extensions.

10.2 Program Execution

SPARK analyses do not involve program execution. However, SPARK programs are executable including those new language defined aspects and pragmas where they have dynamic semantics given.

10.2.1 Elaboration Control

No extensions or restrictions.

CHAPTER

ELEVEN

EXCEPTIONS

11.1 Exception Declarations

No additions or restrictions

11.2 Exception Handlers

Exception handlers are supported in SPARK, but the verification rules associated to language mandated checks and contracts make it so that only exceptions raised in actual raise statements can be handled.

Legality Rules

1. Exception handlers shall not have a choice parameter.

11.3 Raise Statements and Raise Expressions

Raise statements and raise expressions are in SPARK. An exception is said to be *expected* if it is covered by a choice of an exception handler in an enclosing handled sequence of statements, or if its enclosing entity is a procedure body and the exception is covered by a choice in its Exceptional_Cases aspect whose associated consequence is not statically False.

As described below, all raise expressions must be provably never executed. The same holds true for raise statements if they raise unexpected exceptions.

Verification Rules

1. A `raise_expression` introduces an obligation to prove that the expression will not be evaluated, much like the verification condition associated with

   ```
   pragma Assert (False);
   ```

   [In other words, the verification conditions introduced for a raise expression are the same as those introduced for a run-time check which fails unconditionally.]

2. A `raise_statement` introduces an obligation to prove that the exception raised is expected. [For raise statements with an exception name which is unexpected, this amounts to proving that the statement will not be executed.]

11.4 Exception Handling

No additions or restrictions.

11.4.1 The Package Exceptions

11.4.2 Pragmas Assert and Assertion_Policy

Legality Rules

1. The pragmas `Assertion_Policy`, `Suppress`, and `Unsuppress` are allowed in SPARK, but have no effect on the generation of verification conditions. [For example, an array index value must be shown to be in bounds regardless of whether Index_Check is suppressed at the point of the array indexing.]
2. The following SPARK defined aspects and pragmas are assertions and their *Boolean*`_expressions` are assertion expressions:
   - Assert_And_Cut;
   - Assume;
   - Contract_Cases;
   - Default_Initial_Condition;
   - Initial_Condition;
   - Loop_Invariant;
   - Loop_Variant; and
   - Refined_Post.

   There is an *assertion*`_aspect_mark` for each of these aspects and pragmas with the same identifier as the corresponding aspect or pragma. In addition, Ghost is a SPARK defined *assertion*`_aspect_mark`.

   An implementation may introduce further implementation defined *assertion*`_aspect_marks` some of which may apply to groups of these assertions.

CHAPTER

TWELVE

GENERIC UNITS

Enforcement of SPARK's rules within a generic unit is not guaranteed. Violations might not be reported until an instance of the generic unit is analyzed. If an instance of a generic unit occurs within another generic unit, this principle is applied recursively.

12.1 Generic Instantiation

Legality Rules

1. An instantiation of a generic is or is not in SPARK depending on whether the instance declaration and the instance body (described in section 12.3 of the Ada reference manual) are in SPARK [(i.e., when considered as a package (or, in the case of an instance of a generic subprogram, as a subprogram)].
2. [A generic actual parameter corresponding to a generic formal object having mode **in** shall not have a variable input; see *Expressions* for the statement of this rule.]

[For example, a generic which takes a formal limited private type would be in SPARK. An instantiation which passes in a tagged type with subcomponents of an access type as the actual type would not be in SPARK; another instantiation of the same generic which passes in, for example, Standard.Integer, might be in SPARK.]

[Ada has a rule that legality rules are not enforced in an instance body (and, in some cases, in the private part of an instance of a generic package). No such rule applies to the restrictions defining which Ada constructs are in SPARK. For example, a backward goto statement in an instance body would cause the instantiation to not be in SPARK.]

[Consider the problem of correctly specifying the Global and Depends aspects of a subprogram declared within an instance body which contains a call to a generic formal subprogram (more strictly speaking, to the corresponding actual subprogram of the instantiation in question). These aspects are simply copied from the corresponding aspect specification in the generic, so this implies that we have to "get them right" in the generic (where "right" means "right for all instantiations"). One way to do this is to assume that a generic formal subprogram references no globals (or, more generally, references any fixed set of globals) and to only instantiate the generic with actual subprograms that meet this requirement.]

CHAPTER

THIRTEEN

REPRESENTATION ISSUES

13.1 Operational and Representation Aspects

SPARK defines several Boolean-valued aspects. These include the Async_Readers, Async_Writers, Constant_After_Elaboration, Effective_Reads, Effective_Writes, Extensions_Visible, Ghost, Side_Effects and Volatile_Function aspects. [Note that this list does not include expression-valued aspects, such as Default_Initial_Condition or Initial_Condition.]

The following rules apply to each of these aspects unless specified otherwise for a particular aspect:

1. In the absence of an aspect specification (explicit or inherited), the default value of the given aspect is False.
2. If the given aspect is specified via an aspect_specification [(as opposed to via a pragma)] then the aspect_definition (if any) shall be a static Boolean expression. [Omitting the aspect_definition in an aspect_specification is equivalent to specifying a value of True as described in Ada RM 13.1.1(15).]
3. The usage names in an aspect_definition for the given aspect are resolved at the point of the associated declaration. [This supersedes the name resolution rule given in Ada RM 13.1.1 that states that such names are resolved at the end of the enclosing declaration list.]

[One case where the "unless specified otherwise" clause applies is illustrated by

> X : Integer with Volatile;

where the Async_Readers aspect of X is True, not False.]

Ada allows aspect specifications for package declarations and package bodies but does not define any aspects which can be specified in this way. SPARK defines, for example, the Initial_Condition and Refined_State aspects (the former can be specified for a package declaration; the latter for a package body). Ada's usual rule that

> The usage names in an aspect_definition [are not resolved at the point of the associated declaration, but rather] are resolved at the end of the immediately enclosing declaration list.

is applied for such aspects as though "the immediately enclosing declaration list" is that of the visible part (in the former case) or of the body (in the latter case). [For example, the Initial_Condition expression of a package which declares a variable in its visible part can (directly) name that variable. Simlarly, the Refined_State aspect specification for a package body can name variables declared within the package body.]

13.2 Packed Types

No restrictions or additions.

13.3 Operational and Representation Attributes

No restrictions or additions.

13.4 Enumeration Representation Clauses

No restrictions or additions.

13.5 Record Layout

13.6 Change of Representation

No restrictions or additions.

13.7 The Package System

Direct manipulation of addresses is restricted in SPARK. In particular, the use of address clauses or aspects to define the address of an object in memory is restricted in SPARK. If the address of an object `X` is specified to be the address of another object `Y`, using an address clause of the form `with Address => Y'Address`, then `X` is said to be overlaid on `Y`. Both `X` and `Y` are said to be overlaid objects. The verification rules below impose restrictions on overlaid objects in SPARK. Other address clauses and aspects are not restricted; the onus is on the user to ensure that this is correct with respect to the program semantics of SPARK.

Legality Rules

1. The use of the operators defined for type Address are not permitted in SPARK except for use within representation clauses.

Verification Rules

2. If an object `X` is overlaid on an object `Y`, then the sizes of `X` and `Y` shall be known at compile-time and shall be equal.
3. If an object `X` is overlaid on an object `Y`, then the alignment of `Y` shall be an integral multiple of the alignment of `X`.
4. The type of an overlaid object shall be suitable as the target of an unchecked conversion (see *Unchecked Type Conversions*);
5. If the address clause of an object `X` is not of the form `with Address => Y'Address` for some object `Y`, then `X` shall be volatile.
6. If the address of an object `Y` is taken other than in an address clause of the form `with Address => Y'Address`, then `Y` shall be volatile.
7. If an object `X` overlays an object `Y`, then neither `X` nor `Y` shall be constituents of an abstract state.

13.8 Machine Code Insertions

Legality Rules

1. Machine code insertions are not in SPARK.

13.9 Unchecked Type Conversions

A subtype S is said to be *suitable for unchecked conversion* if:

- S is not of a tagged type, of an access type, of an immutably limited type, of a type with discriminants, or of a private type whose completion fails to meet these requirements.
- if S is a floating-point type, its Size is not greater than the Size of the largest floating-point type on the target.
- if S is a scalar type that is not a floating-point type, its Size is not greater than the Size of the largest integer type on the target.
- if S is a composite type, the Size N of S is the sum of the Size of the components of S, and all components of S are also suitable for unchecked conversion.

[Limits on the Size of scalar types are meant to allow the compiler to zero out extra bits not used in the representation of the scalar value, when writing a value of the type (as GNAT ensures).]

A subtype S is said to be *suitable as the target of an unchecked conversion* if it is suitable for unchecked conversion, and, in addition:

- S is not of a subtype that is subject to a predicate, or of a type that is subject to a type invariant.
- Given the Size N of S in bits, there exist exactly 2**N distinct valid values that belong to S and contain no invalid scalar parts. [In other words, every possible assignment of values to the bits representing an object of subtype S represents a distinct value of S.]
- If S is a composite type, all parts of S are also suitable as the target of an unchecked conversion.

[Note that floating-point types are not suitable as the target of an unchecked conversion, because NaN is not considered to be a valid value.]

Unchecked type conversions are in SPARK, with some restrictions described below. Although it is not mandated by Ada standard, the compiler should ensure that it does not return the result of unchecked conversion by reference if it could be misaligned (as GNAT ensures).

Verification Rules

1. The source and target subtypes of an instance of `Unchecked_Conversion` shall have the same Size.
2. The source and target subtypes shall be suitable for unchecked conversion and the target subtype should be suitable as the target of an unchecked conversion.

13.9.1 Data Validity

SPARK rules ensure the only possible cases of invalid data in a SPARK program come from interfacing with the external world, either through the hardware-software or Operating Systems integration, or through interactions with non-SPARK code in the same program. In particular, it is up to users to ensure that data read from external sources are valid.

Validity can be ensured by using a type for the target of the data read from an external source (or an unchecked type conversion when used to read data from external source) which is sufficient to encompass all possible values of the source. Alternatively the X'Valid (or X'Valid_Scalars for composite types) may be used to help determine the validity of an object.

The use of invalid values in a program (other than in a Valid, or Valid_Scalars attribute) may invalidate any proofs performed on the program.

13.10 Unchecked Access Value Creation

Legality Rules

1. The Unchecked_Access attribute is not in SPARK.

13.11 Storage Management

Legality Rules

1. Aspect specifications for the Storage_Pool and Storage_Size aspects are not in SPARK, nor are uses of the corresponding attributes. The predefined unit System.Storage_Pools is not in SPARK, nor is any other predefined unit that semantically depends on it. The pragma Default_Storage_Pool is not in SPARK.

13.12 Pragma Restrictions and Pragma Profile

Restrictions and Profiles will be available with SPARK to provide profiles suitable for different application environments.

13.13 Streams

Legality Rules

1. Stream types and operations are not in SPARK.

13.14 Freezing Rules

No restrictions or additions.

CHAPTER

FOURTEEN

PREDEFINED LANGUAGE ENVIRONMENT (ANNEX A)

This chapter describes how SPARK treats the Ada predefined language environment and standard libraries, corresponding to appendices A through H of the Ada RM.

SPARK programs are able to use much of the Ada predefined language environment and standard libraries. The standard libraries are not necessarily mathematically, formally proven in any way, unless specifically stated, and should be treated as tested code.

In addition many standard library subprograms have checks on the consistency of the actual parameters when they are called. If they are inconsistent in some way they will raise an exception. It is strongly recommended that each call of a standard library subprogram which may raise an exception due to incorrect actual parameters should be immediately preceded by a pragma Assert to check that the actual parameters meet the requirements of the called subprogram. Alternatively the called subprogram may be wrapped in a user defined subprogram with a suitable precondition. Examples of these approaches are given in *The Package Strings.Maps (A.4.2)*.

No checks or warnings are given that this protocol is followed. The onus is on the user to ensure that a library subprogram is called with consistent actual parameters.

14.1 The Package Standard (A.1)

SPARK supports all of the types, subtypes and operators declared in package Standard. The predefined exceptions are considered to be declared in Standard, but their use is constrained by other language restrictions.

14.2 The Package Ada (A.2)

No additions or restrictions.

14.3 Character Handling (A.3)

14.3.1 The Packages Characters, Wide_Characters, and Wide_Wide_Characters (A.3.1)

No additions or restrictions. As in Ada, the wide character sets provided are SPARK tool, compiler and platform dependent.

14.3.2 The Package Characters.Handling (A.3.2)

No additions or restrictions.

14.3.3 The Package Characters.Latin_1 (A.3.3)

No additions or restrictions.

14.3.4 The Package Characters.Conversions (A.3.4)

No Additions or restrictions.

14.3.5 The Package Wide_Characters.Handling (A.3.5)

No additions or restrictions.

14.3.6 The Package Wide_Wide_Characters.Handling (A.3.6)

No additions or restrictions.

14.4 String Handling (A.4)

No additions or restrictions.

14.4.1 The Package Strings (A.4.1)

No additions or restrictions.

The predefined exceptions are considered to be declared in Strings, but their use is constrained by other language restrictions.

14.4.2 The Package Strings.Maps (A.4.2)

Preconditions and postconditions are added to subprograms. Preconditions prevent all exceptions specified in the corresponding part of the Ada RM to be raised.

14.4.3 Fixed-Length String Handling (A.4.3)

Preconditions and postconditions are added to subprograms. Preconditions on subprograms prevent all exceptions specified in the corresponding part of the Ada RM to be raised, except for procedures Move, Replace_Slice, Insert, Overwrite, Head, and Tail. These have incomplete contracts and may raise an exception if they are called with inconsistent actual parameters. Each call of these procedures should be preceded with a pragma Assert to check that the actual parameters are consistent. Postconditions on subprograms, when present, fully detail their effect.

14.4.4 Bounded-Length String Handling (A.4.4)

Global, preconditions and postconditions are added to subprograms. Preconditions prevent all exceptions specified in the corresponding part of the Ada RM to be raised.

14.4.5 Unbounded-Length String Handling (A.4.5)

1. The type String_Access and the procedure Free are not in SPARK as they require non-owning access types and cannot be denoted in SPARK program text.

Global, preconditions and postconditions are added to subprograms. Preconditions prevent all exceptions specified in the corresponding part of the Ada RM to be raised.

14.4.6 String-Handling Sets and Mappings (A.4.6)

No additions or restrictions.

14.4.7 Wide_String Handling (A.4.7)

1. The types Wide_String_Access and Wide_Character_Mapping_Function are not in SPARK nor are the subprograms which have formal parameters of these types and cannot be denoted in SPARK program texts.

Each call of a subprogram which may raise an exception if it is called with inconsistent actual parameters should be immediately preceded by a pragma Assert checking the consistency of the actual parameters.

14.4.8 Wide_Wide_String Handling (A.4.8)

1. The types Wide_Wide_String_Access and Wide_Wide_Character_Mapping_Function are not in SPARK nor are the subprograms which have formal parameters of these types and cannot be denoted in SPARK program texts.

Each call of a subprogram which may raise an exception if it is called with inconsistent actual parameters should be immediately preceded by a pragma Assert checking the consistency of the actual parameters.

14.4.9 String Hashing (A.4.9)

No additions or restrictions.

14.4.10 String Comparison (A.4.10)

No additions or restrictions.

14.4.11 String Encoding (A.4.11)

The subprograms of this package are callable from SPARK but those that may raise an exception due to inconsistent parameters should have a pragma Assert confirming that the actual parameters are consistent immediately preceding each call of such a subprogram.

14.5 The Numerics Packages (A.5)

No additions or restrictions

14.5.1 Elementary Functions (A.5.1)

All functions are annotated with preconditions that guard against exceptions being raised. The following functions may produce infinite results for some inputs which satisfy their preconditions (if any). For SPARK, this is just as bad as propagating an exception. Both are events that can invalidate SPARK proofs because proofs may rely on an assumption that these events do not occur. Thus, the onus is on the user to avoid such inputs:

- function Exp returns +infinite on large values of argument X
- function ** returns +infinite on large values of arguments Left and Right
- functions Cot of one argument, as well as functions Tan and Cot with arguments X and Cycle, may return an infinite on values of X that are close to their singularity points
- functions Sinh and Cosh return an infinite on larges values of argument X
- function Coth returns an infinite on small values of argument X close to zero
- functions Arctanh and Arccoth return an infinite on values of argument X close to one

Interestingly, function Tan of one argument never returns an infinite result for any input value, both in 32-bits and 64-bits floating-points. This is due to all floating-point approximations of its singularity points being too far from the singularity (all values that are a multiple of π away from $\pi/2$).

14.5.2 Random Number Generation (A.5.2)

The package Ada.Numerics.Float_Random and an instantiation of package Ada.Numerics.Discrete_Random is ostensibly in SPARK but the functions have side effects and should not be called from SPARK text.

14.6 Input-Output (A.6)

No additions or restrictions.

14.7 External Files and File Objects (A.7)

No additions or restrictions.

14.8 Sequential and Direct Files (A.8)

No additions or restrictions.

14.8.1 The Generic Package Sequential_IO (A.8.1)

An instantiation of Sequential_IO will ostensibly be in SPARK but in use it may give rise to flow-errors as the effect of reads and writes is not captured in the subprogram contracts. Calls to its subprograms may raise IO_Exceptions based on external events.

14.8.2 File Management (A.8.2)

No additions or restrictions.

14.8.3 Sequential Input-Output Operations (A.8.3)

No additions or restrictions.

14.8.4 The Generic Package Direct_IO (A.8.4)

An instantiation of Direct_IO will ostensibly be in SPARK but in use it may give rise to flow-errors as the effect of reads and writes is not captured in the subprogram contracts. Calls to its subprograms may raise IO_Exceptions based on external events.

14.8.5 Direct Input-Output Operations (A.8.5)

No additions or restrictions.

14.9 The Generic Package Storage_IO (A.9)

An instantiation of Storage_IO will ostensibly be in SPARK but in use it may give rise to flow-errors as the effect of reads and writes is not captured in the subprogram contracts. Calls to its subprograms may raise IO_Exceptions based on external events.

14.10 Text Input-Output (A.10)

No additions or restrictions.

14.10.1 The Package Text_IO (A.10.1)

Ada.Text_IO is ostensibly in SPARK except for the type File_Access, a generalized access type, thus preventing Ada.Text_IO from being declared with SPARK_Mode On explicitly in the visible part. The following subprograms are explicitly marked as SPARK_Mode Off:

- The functions Current_Input, Current_Output, Current_Error, Standard_Input, Standard_Output and Standard_Error because they create aliasing, by returning the corresponding file.
- The procedures Set_Input, Set_Output and Set_Error because they also create aliasing, by assigning a File_Type variable to respectively Current_Input, Current_Output or Current_Error.
- Functions Get_Line because they have a side effect of reading data from a file and updating its file pointers.

The abstract state File_System declared in Ada.Text_IO is used to model the memory on the system and the file handles (Line_Length, Col, etc.). This is made necessary by the fact that almost every procedure in Text_IO that actually modifies attributes of its File_Type parameter takes it as an **in** parameter.

All functions and procedures are annotated with Global, and Pre/Post when possible. The Global contracts are typically In_Out for File_System, even in Put or Get procedures that update the current column and/or line. Functions have an Input global contract. The only functions with Global => null are the functions Get and Put in the generic packages that have the same behavior as sprintf and sscanf.

Preconditions are not always complete, as not all conditions leading to run-time exceptions can be effectively modelled in SPARK:

- Status_Error (due to a file already open/not open) is fully modelled
- Mode_Error (due to a violation of the internal state machine) is fully modelled
- Layout_Error is partially modelled
- Use_Error is not modelled (it is related to the external environment)
- Name_Error is not modelled (it would require checking availability on disk beforehand)
- End_Error is not modelled (it is raised when a file terminator is read while running the procedure)

In the exceptional cases that are not fully modelled, it is possible that SPARK tools do not issue a possible precondition failure message on a call, yet an exception can be raised at run-time. See the spec files for the exact contracts.

14.10.2 Text File Management (A.10.2)

The possibility of errors related to the actual content or limitations of the file system are not modelled (e.g. when trying to create an already existing file, or open a file that does not exist).

Preconditions and postconditions are added to describe other constraints.

14.10.3 Default Input, Output and Error Files (A.10.3)

Apart from procedure Flush, all other subprograms are explicitly marked as SPARK_Mode Off, as described above, because they create aliasing.

14.10.4 Specification of Line and Page Lengths (A.10.4)

Global, preconditions and postconditions are added to subprograms.

14.10.5 Operations on Columns, Lines and Pages (A.10.5)

Global, preconditions and postconditions are added to subprograms.

14.10.6 Get and Put Procedures (A.10.6)

Global, preconditions and postconditions are added to subprograms.

14.10.7 Input-Output of Characters and Strings (A.10.7)

Functions Get_Line are explicitly marked as SPARK_Mode Off, as described above, because they have side effects.

Global, preconditions and postconditions are added to other subprograms.

14.10.8 Input-Output for Integer Types (A.10.8)

Global, preconditions and postconditions are added to subprograms.

14.10.9 Input-Output for Real Types (A.10.9)

Global, preconditions and postconditions are added to subprograms.

14.10.10 Input-Output for Enumeration Types (A.10.10)

Global, preconditions and postconditions are added to subprograms.

14.10.11 Input-Output for Bounded Strings (A.10.11)

An instantiation of Bounded_IO will ostensibly be in SPARK but in use it may give rise to flow-errors as the effect of reads and writes is not captured in the subprogram contracts. Calls to its subprograms may raise IO_Exceptions based on external events.

14.10.12 Input-Output of Unbounded Strings (A.10.12)

Ada.Text_IO.Unbounded_IO is ostensibly in SPARK but in use it may give rise to flow-errors as the effect of reads and writes is not captured in the subprogram contracts. Calls to its subprograms may raise IO_Exceptions based on external events.

The functions Ada.Text_IO.Unbounded_IO.Get_Line should not be called from SPARK program text as the functions have a side effect of reading from a file.

14.11 Wide Text Input-Output and Wide Wide Text Input-Output (A.11)

These packages have the same constraints as was discussed for Ada.Text_IO.

14.12 Stream Input-Output (A.12)

Stream input and output is not supported by SPARK and the use of the package Ada.Streams.Stream_IO and the child packages of Ada.Text_IO concerned with streams is not permitted in SPARK program text.

14.13 Exceptions in Input-Output (A.13)

The exceptions declared in package Ada.IO_Exceptions which are raised by the Ada input-output subprograms are in SPARK but the exceptions cannot be handled in SPARK program text.

14.14 File Sharing (A.14)

File sharing is not permitted in SPARK, since it may introduce an alias.

14.15 The Package Command_Line (A.15)

The package Command_Line is in SPARK except that the function Argument may propagate Constraint_Error. To avoid this exception each call to Argument should be immediately preceded by the assertion:

```
pragma Assert (Number <= Argument_Count);
```

where Number represents the actual parameter to the function Argument.

14.16 The Package Directories (A.16)

The package Directories is ostensibly in SPARK but in use it may give rise to flow-errors as the effect of reads and writes is not captured in the subprogram contracts. Calls to its subprograms may raise IO_Exceptions based on external events.

14.17 The Package Environment_Variables (A.17)

The package Environment_Variables is ostensibly mostly in SPARK but in use it may give rise to flow-errors as the effect of reads and writes is not captured in the subprogram contracts. Calls to its subprograms may raise IO_Exceptions based on external events.

The procedure Iterate is not in SPARK.

14.18 Containers (A.18)

The standard Ada container libraries are not supported in SPARK.

An implementation may choose to provide alternative container libraries whose specifications are in SPARK and are intended to support formal verification.

14.19 The Package Locales (A.19)

No additions or restrictions.

14.20 Interface to Other Languages (Annex B)

This section describes features for mixed-language programming in SPARK, covering facilities offered by Ada's Annex B.

Package `Interfaces` can be used in SPARK, including its intrinsic "Shift" and "Rotate" functions.

Other packages are not directly supported.

14.21 Systems Programming (Annex C)

This section describes features for systems programming in SPARK, covering facilities offered by Ada's Annex C.

Almost all of the facilities offered by this Annex are out of scope for SPARK and so are not supported.

14.21.1 Pragma Discard_Names (C.5)

Pragma Discard_Names is not permitted in SPARK, since its use can lead to implementation defined behaviour at run time.

14.21.2 Shared Variable Control (C.6)

The following restrictions are applied to the declaration of volatile types and objects in SPARK:

Legality Rules

1. A volatile representation aspect may only be applied to an `object_declaration` or a `full_type_declaration`.
2. A type which is not effectively volatile shall not have a volatile subcomponent.
3. A discriminant shall not be volatile.
4. Neither a discriminated type nor an object of such a type shall be volatile.
5. Neither a tagged type nor an object of such a type shall be volatile.
6. An effectively volatile object shall only be declared at library-level.

14.22 Real-Time Systems (Annex D)

SPARK supports the parts of the real-time systems annex that comply with the Ravenscar or Jorvik profiles (see Ada RM D.13). See section *Tasks and Synchronization*.

14.23 Distributed Systems (Annex E)

SPARK does not support the distributed systems annex.

14.24 Information Systems (Annex F)

The `Machine_Radix` aspect and attribute are permitted in SPARK.

The package `Ada.Decimal` may be used, although it declares constants whose values are implementation defined.

The packages `Ada.Text_IO.Editing` and its "Wide" variants are not directly supported in SPARK.

14.25 Numerics (Annex G)

This section describes features for numerical programming in SPARK, covering facilities offered by Ada's Annex G.

Packages declared in this Annex are usable in SPARK, although many details are implementation defined.

Implementations (both compilers and verification tools) should document how both *strict mode* and *relaxed mode* are implemented and their effect on verification and performance.

14.26 High Integrity Systems (Annex H)

SPARK fully supports the requirements of Ada's Annex H.

CHAPTER

FIFTEEN

LANGUAGE-DEFINED ASPECTS AND ATTRIBUTES (ANNEX K)

15.1 Language-Defined Aspects

1. Ada language aspects are permitted as shown in the following table:

Aspect	Allowed in SPARK	Comment
Address	Yes	
Alignment (object)	Yes	
Alignment (subtype)	Yes	
All_Calls_Remote	No	
Asynchronous	No	
Atomic	Yes	
Atomic_Components	Yes	
Attach_Handler	Yes	
Bit_Order	Yes	
Coding	Yes	
Component_Size	Yes	
Constant_Indexing	No	
Convention	Yes	
CPU	Yes	
Default_Component_Value	Yes	
Default_Iterator	No	
Default_Storage_Pool	No	
Default_Value	Yes	
Default_Storage_Pool	No	Restricted access types
Dispatching_Domain	No	Ravenscar
Dynamic_Predicate	Yes	
Elaborate_Body	Yes	
Export	Yes	
External_Name	Yes	
External_Tag	No	No tags
Implicit_Dereference	No	Restricted access types
Import	Yes	
Independent	Yes	
Independent_Components	Yes	
Inline	Yes	
Interrupt_Handler	Yes	
Interrupt_Priority	Yes	
Iterator_Element	No	

continues on next page

Table 1 – continued from previous page

Aspect	Allowed in SPARK	Comment
Layout (record)	Yes	
Link_Name	Yes	
Machine_Radix	Yes	
No_Return	Yes	
Output	No	No streams
Pack	Yes	
Pre	Yes	
Pre'Class	Yes	
Post	Yes	
Post'Class	Yes	
Predicate_Failure	Yes	
Preelaborate	Yes	
Priority	Yes	No variable input
Pure	Yes	
Relative_Deadline	Yes	
Remote_Call_Interface	No	
Remote_Types	No	
Shared_Passive	No	
Size (object)	Yes	
Size (subtype)	Yes	
Small	Yes	
Static_Predicate	Yes	
Storage_Pool	No	Restricted access types
Storage_Size (access)	No	Restricted access types
Storage_Size (task)	Yes	
Stream_Size	No	No streams
Synchronization	Yes	
Type_Invariant	Yes	
Type_Invariant'Class	No	
Unchecked_Union	Yes	
Variable_Indexing	No	
Volatile	Yes	
Volatile_Components	Yes	
Write	No	No streams

2. SPARK defines the following aspects:

Aspect	Allowed in SPARK	Comment
Abstract_State	Yes	
Always_Terminates	Yes	
Async_Readers	Yes	
Async_Writers	Yes	
Constant_After_Elaboration	Yes	
Contract_Cases	Yes	
Default_Initial_Condition	Yes	
Depends	Yes	
Effective_Reads	Yes	
Effective_Writes	Yes	
Exceptional_Cases	Yes	
Extensions_Visible	Yes	
Ghost	Yes	
Global	Yes	
Initial_Condition	Yes	
Initializes	Yes	
No_Caching	Yes	
Part_Of	Yes	
Refined_Depends	Yes	
Refined_Global	Yes	
Refined_Post	Yes	
Refined_State	Yes	
Side_Effects	Yes	
SPARK_Mode	Yes	Language defined but implementation dependent
Volatile_Function	Yes	

15.2 Language-Defined Attributes

1. Ada language attributes are permitted as shown in the following table:

Attribute	Allowed in SPARK	Comment
P'Access	No	Restricted access types
X'Access	Yes	
X'Address	No	Only allowed in representation clauses
S'Adjacent	Yes	Only supported with static attribute expressions; implicit precondition (A
S'Aft	Yes	
S'Alignment	Warn	Warning in pedantic mode
X'Alignment	Warn	Warning in pedantic mode
S'Base	Yes	
S'Bit_Order	Warn	Warning in pedantic mode
P'Body_Version	No	
T'Callable	Yes	
E'Caller	Yes	
S'Ceiling	Yes	
S'Class	Yes	
X'Component_Size	Warn	Warning in pedantic mode
S'Compose	Yes	Only supported with static attribute expressions

contin

Table 2 – continued from previous page

Attribute	Allowed in SPARK	Comment
A'Constrained	Yes	
S'Copy_Sign	Yes	
E'Count	No	
S'Definite	Yes	
S'Delta	Yes	
S'Denorm	Yes	
S'Digits	Yes	
S'Exponent	Yes	Only supported with static attribute expressions
S'External_Tag	No	No tags
A'First	Yes	
S'First	Yes	
A'First(N)	Yes	
R.C'First_Bit	Warn	Warning in Pedantic mode
S'First_Valid	Yes	
S'Floor	Yes	
S'Fore	Yes	
S'Fraction	Yes	Only supported with static attribute expressions
X'Has_Same_Storage	No	
E'Identity	No	
T'Identity	Yes	
X'Image	Yes	Same as S'Image(X) (Ada RM 3.5(55.4/4))
S'Image	Yes	
S'Class'Input	No	No streams
S'Input	No	No streams
A'Last	Yes	
S'Last	Yes	
A'Last(N)	Yes	
R.C'Last_Bit	Warn	Warning in pedantic mode
S'Last_Valid	Yes	
S'Leading_Part	Yes	Only supported with static attribute expressions
A'Length	Yes	
A'Length(N)	Yes	
S'Machine	Yes	Only supported with static attribute expressions
S'Machine_Emax	Yes	
S'Machine_Emin	Yes	
S'Machine_Mantissa	Yes	
S'Machine_Overflows	Yes	
S'Machine_Radix	Yes	
S'Machine_Rounding	Yes	
S'Machine_Rounds	Yes	
S'Max	Yes	
S'Max_Alignment_For_Allocation	No	Restricted access types
S'Max_Size_In_Storage_Elements	No	Restricted access types
S'Min	Yes	
S'Mod	Yes	
S'Model	Yes	Only supported with static attribute expressions
S'Model_Emin	Yes	
S'Model_Epsilon	Yes	
S'Model_Mantissa	Yes	
S'Model_Small	Yes	

continu

Table 2 – continued from previous page

Attribute	Allowed in SPARK	Comment
S'Modulus	Yes	
X'Old	Yes	
S'Class'Output	No	No streams
S'Output	No	No streams
X'Overlaps_Storage	No	
D'Partition_Id	Yes	
S'Pos	Yes	
R.C'Position	Warn	Warning in pedantic mode
S'Pred	Yes	Implicit precondition (Ada RM 3.5(27))
P'Priority	No	Ravenscar
A'Range	Yes	
S'Range	Yes	
A'Range(N)	Yes	
S'Class'Read	No	No streams
S'Read	No	No streams
S'Remainder	Yes	
F'Result	Yes	
S'Round	Yes	
S'Rounding	Yes	
S'Safe_First	Yes	
S'Safe_Last	Yes	
S'Scale	Yes	
S'Scaling	Yes	Only supported with static attribute expressions
S'Signed_Zeros	Yes	
S'Size	Warn	Warning in pedantic
X'Size	Warn	Warning in pedantic
S'Small	Yes	
S'Storage_Pool	No	Restricted access types
S'Storage_Size	No	Restricted access types
T'Storage_Size	Yes	
S'Stream_Size	No	No streams
S'Succ	Yes	Implicit precondition (Ada RM 3.5(24))
S'Tag	No	No tags
X'Tag	No	No tags
T'Terminated	Yes	
System'To_Address	Yes	
S'Truncation	Yes	
S'Truncation	Yes	
S'Unbiased_Rounding	Yes	Only supported with static attribute expressions
X'Unchecked_Access	No	
X'Update	Yes	
S'Val	Yes	Implicit precondition (Ada RM 3.5.5(7))
X'Valid	Yes	Assumed to be True at present
S'Value	Yes	Implicit precondition (Ada RM 3.5(55/3))
P'Version	No	
S'Wide_Image	Yes	
S'Wide_Value	Yes	Implicit precondition (Ada RM 3.5(43/3))
S'Wide_Wide_Image	Yes	
S'Wide_Wide_Value	Yes	Implicit precondition (Ada RM 3.5(39.12/3))
S'Wide_Wide_Width	Yes	

contin

Table 2 – continued from previous page

Attribute	Allowed in SPARK	Comment
S'Wide_Width	Yes	
S'Width	Yes	
S'Class'Write	No	No streams
S'Write	No	No streams

2. SPARK defines the following attributes:

Attribute	Allowed in SPARK	Comment
X'Initialized	Yes	Only allowed in ghost code
X'Loop_Entry	Yes	

15.3 GNAT Implementation-Defined Attributes

The following GNAT implementation-defined attributes are permitted in SPARK:

Attribute	Allowed in SPARK	Comment
X'Img	Yes	Same as X'Image (Ada RM 3.5(55.4/4))

CHAPTER

SIXTEEN

LANGUAGE-DEFINED PRAGMAS (ANNEX L)

16.1 Ada Language-Defined Pragmas

The following Ada language-defined pragmas are supported as follows:

Pragma	Allowed in SPARK	Comment
All_Calls_Remote	No	
Assert	Yes	
Assertion_Policy	Yes	No effect on provability (see section “Assertion Pragmas” in the SPARK User
Asynchronous	No	
Atomic	Yes	
Atomic_Components	Yes	
Attach_Handler	Yes	
Convention	Yes	
CPU	Yes	
Default_Storage_Pool	No	Restricted access types
Detect_Blocking	Yes	
Discard_Names	No	
Dispatching_Domain	No	Ravenscar
Elaborate	Yes	
Elaborate_All	Yes	
Elaborate_Body	Yes	
Export	Yes	
Import	Yes	
Independent	Yes	
Independent_Components	Yes	
Inline	Yes	
Inspection_Point	Yes	
Interrupt_Handler	Yes	
Interrupt_Priority	Yes	
Linker_Options	Yes	
List	Yes	
Locking_Policy	Yes	
No_Return	Yes	
Normalize_Scalars	Yes	
Optimize	Yes	
Pack	Yes	
Page	Yes	
Partition_Elaboration_Policy	Yes	Ravenscar

continues on n

Table 1 – continued from previous page

Pragma	Allowed in SPARK	Comment
Preelaborable_Initialization	Yes	
Preelaborate	Yes	
Priority	Yes	
Priority_Specific_Dispatching	No	Ravenscar
Profile	Yes	
Pure	Yes	
Queuing_Policy	Yes	Ravenscar
Relative_Deadline	Yes	
Remote_Call_Interface	No	Distributed systems
Remote_Types	No	Distributed systems
Restrictions	Yes	
Reviewable	Yes	
Shared_Passive	No	Distributed systems
Storage_Size	Yes/No	tasks, not access types
Suppress	Yes	
Task_Dispatching_Policy	No	Ravenscar
Unchecked_Union	Yes	
Unsuppress	Yes	
Volatile	Yes	
Volatile_Components	Yes	

16.2 SPARK Language-Defined Pragmas

The following SPARK language-defined pragmas are defined:

Pragma	Allowed in SPARK	Comment
Abstract_State	Yes	
Assert_And_Cut	Yes	
Assume	Yes	
Async_Readers	Yes	
Async_Writers	Yes	
Constant_After_Elaboration	Yes	
Contract_Cases	Yes	
Default_Initial_Condition	Yes	
Depends	Yes	
Effective_Reads	Yes	
Effective_Writes	Yes	
Extensions_Visible	Yes	
Ghost	Yes	
Global	Yes	
Initial_Condition	Yes	
Initializes	Yes	
Loop_Invariant	Yes	
Loop_Variant	Yes	
No_Caching	Yes	
Part_Of	Yes	
Refined_Depends	Yes	
Refined_Global	Yes	
Refined_Post	Yes	
Refined_State	Yes	
SPARK_Mode	Yes	Language defined but implementation dependent
Unevaluated_Use_Of_Old	Yes	
Volatile_Function	Yes	

16.3 GNAT Implementation-Defined Pragmas

The following GNAT implementation-defined pragmas are permitted in SPARK:

Pragma	Allowed in SPARK	Comment
Ada_83	Yes	
Ada_95	Yes	
Ada_05	Yes	
Ada_12	Yes	
Ada_2005	Yes	
Ada_2012	Yes	
Ada_2020	Yes	
Annotate	Yes	
Check	Yes	
Check_Policy	Yes	No effect on provability (see section “Assertion Pragmas” in the SPARK Use
Compile_Time_Error	Yes	Ignored (replaced by null statement)
Compile_Time_Warning	Yes	Ignored (replaced by null statement)
Debug	Yes	Ignored (replaced by null statement)
Default_Scalar_Storage_Order	Yes	

continues on n

Table 2 – continued from previous page

Pragma	Allowed in SPARK	Comment
Export_Function	Yes	
Export_Procedure	Yes	
Ignore_Pragma	Yes	
Inline_Always	Yes	
Invariant	Yes	
Linker_Section	Yes	
Max_Queue_Length	Yes	Extended Ravenscar
No_Elaboration_Code_All	Yes	
No_Heap_Finalization	Yes	
No_Inline	Yes	No effect on contextual analysis of subprograms
No_Tagged_Streams	Yes	
Overflow_Mode	Yes	
Post	Yes	
Postcondition	Yes	
Post_Class	Yes	
Pre	Yes	
Precondition	Yes	
Pre_Class	Yes	
Predicate	Yes	
Predicate_Failure	Yes	
Provide_Shift_Operators	Yes	
Pure_Function	Yes	
Restriction_Warnings	Yes	
Secondary_Stack_Size	Yes	
Style_Checks	Yes	
Test_Case	Yes	
Type_Invariant	Yes	
Type_Invariant_Class	Yes	
Unmodified	Yes	
Unreferenced	Yes	
Unused	Yes	
Validity_Checks	Yes	
Volatile_Full_Access	Yes	
Warnings	Yes	
Weak_External	Yes	

CHAPTER

SEVENTEEN

GLOSSARY

The SPARK Reference Manual uses a number of technical terms to describe its features and rules. Some of these terms are well known others are less well known or have been defined within this document. In the glossary given here the less well known terms and those defined by SPARK are listed with a brief explanation to their meaning.

- Data-flow analysis is the process of collecting information about the way the variables are used and defined in the program. In particular, in SPARK it is used to detect the use of uninitialized variables and state abstractions.
- Executable semantics is the definition of what it means for a construct to be executed at run-time. In SPARK, most contracts have executable semantics, which means in particular that they can halt execution by raising an exception if some error condition occurs.
- Flow analysis is a term used to cover both data-flow and information-flow analysis.
- Formal Verification, in the context of hardware and software systems, is the act of proving or disproving the correctness of intended algorithms underlying a system with respect to a certain formal specification or property, using formal methods of mathematics. In SPARK this entails proving the implementation of a subprogram against its specification given its precondition using an automatic theorem prover (which may be part of the SPARK toolset). The specification may be given by a postcondition or assertions or may be implicit from the definition of the program when proving absence of run-time exceptions (robustness property).
- Information-flow analysis in an information theoretical context is the transfer of information from a variable x to a variable y in a given process, that is y depends on x. Not all flows may be desirable. For example, perhaps the behavior of one part of a system is intended to be completely independent of the state of another part so that information flow from the latter part to the former would indicate a design error. Or a system shouldn't leak any secret information to public observers. In SPARK information-flow analysis is used to detect useless statements and check that the implementation of a subprogram satisfies its Global aspect and Depends aspect (if they are present). It may also be used for security analysis in SPARK.

APPENDIX A

SPARK 2005 TO SPARK 2014 MAPPING SPECIFICATION

This appendix defines the mapping between SPARK 2005 and SPARK. It is intended as both a completeness check for the SPARK language design, and as a guide for projects upgrading from SPARK 2005 to SPARK 2014.

A.1 SPARK 2005 Features and SPARK 2014 Alternatives

Nearly every SPARK 2005 feature has a SPARK 2014 equivalent or there is an alternative way of providing the same feature in SPARK 2014. The only SPARK 2005 (not including RavenSPARK) features that do not have a direct alternative are:

- the 'Always_Valid attribute;
- the ability to add pre and postconditions to an instantiation of a generic subprogram, e.g., Unchecked_Conversion; and
- a precondition on the body of a subprogram refining the one on the specification - this is not usually required in SPARK 2014, it is normally replaced by the use of expression functions.

At the moment the first two features have to be accomplished using pragma Assume.

The following subsections of this appendix demonstrate how many SPARK 2005 idioms map into SPARK 2014. As a quick reference the table below shows, for each SPARK 2005 annotation or SPARK 2005 specific feature, a reference to the equivalent or alternative in SPARK 2014. In the table headings 2014 RM is the SPARK Reference Manual and Mapping is this appendix, the SPARK 2005 to SPARK 2014 mapping specification.

SPARK 2005	SPARK 2014	2014 RM	Mapping
~ in post	'Old attribute - see Ada RM 6.1.1		*A.2.2*
~ in body	'Loop_Entry attribute	*5.5.3*	*A.7*
<->	=		
A -> B	(if A then B) - see Ada RM 4.5.7		*A.2.2*
%	not needed		*A.7*
always_valid	not supported		*A.4.1*
assert	pragma Assert_And_Cut	*5.9*	*A.4.2*
assert in loop	pragma Loop_Invariant	*5.5.3*	*A.4.1*
assume	pragma Assume	*5.9*	*A.4.1*
check	pragma Assert - see Ada RM 11.4.2		*A.4.1*
derives on spec	Depends aspect	*6.1.5*	*A.2.1*
derives on body	No separate spec - Depends aspect		
derives on body	Separate spec - Refined_Depends aspect	*7.2.5*	*A.3.2*
for all	quantified_expression - see Ada RM 4.5.8		*A.2.3*
for some	quantified_expression - See Ada RM 4.5.8		*A.4.1*

continues on next page

Table 1 – continued from previous page

SPARK 2005	SPARK 2014	2014 RM	Mapping
global on spec	Global aspect	*6.1.4*	*A.2.1*
global on body	No separate spec - Global aspect		
global on body	Separate spec - Refined_Global aspect	*7.2.4*	*A.2.4*
hide	pragma SPARK_Mode - see User Guide		
inherit	not needed		*A.3.4*
initializes	Initializes aspect	*7.1.5*	*A.2.4*
main_program	not needed		
object assertions	rule declarations are not needed		*A.5.3*
own on spec	Abstract_State aspect	*7.1.4*	*A.3.2*
own on body	Refined_State aspect	*7.2.2*	*A.3.2*
post on spec	postcondition - see Ada RM 6.1.1	*6.1.1*	*A.2.2*
post on body	No separate spec - postcondition		
post on body	Separate spec - Refined_Post aspect	*7.2.7*	
pre	precondition - see Ada RM 6.1.1	*6.1.1*	
proof functions	Ghost functions	*6.9*	*A.5.3*
proof types	Ada types		*A.5.5*
return	‘Result attribute - see Ada RM 6.1.1		*A.2.2*
update	delta aggregate		*A.6*

A.2 Subprogram patterns

A.2.1 Global and Derives

This example demonstrates how global variables can be accessed through procedures/functions and presents how the SPARK 2005 *derives* annotation maps over to *depends* in SPARK 2014. The example consists of one procedure (*Swap*) and one function (*Add*). *Swap* accesses two global variables and swaps their contents while *Add* returns their sum.

Specification in SPARK 2005:

```
package Swap_Add_05
--# own X, Y: Integer;
is
   X, Y: Integer;

   procedure Swap;
   --# global in out X, Y;
   --# derives X from Y &
   --#         Y from X;

   function Add return Integer;
   --# global in X, Y;

end Swap_Add_05;
```

body in SPARK 2005:

```
package body Swap_Add_05
is
   procedure Swap
```

(continues on next page)

(continued from previous page)

```
   is
      Temporary: Integer;
   begin
      Temporary := X;
      X         := Y;
      Y         := Temporary;
   end Swap;

   function Add return Integer
   is
   begin
      return X + Y;
   end Add;

end Swap_Add_05;
```

Specification in SPARK 2014:

```
package Swap_Add_14
  with SPARK_Mode
is
   -- Visible variables are not state abstractions.
   X, Y: Integer;

   procedure Swap
     with Global  => (In_Out => (X, Y)),
          Depends => (X => Y,   -- to be read as "X depends on Y"
                      Y => X);  -- to be read as "Y depends on X"

   function Add return Integer
     with Global  => (Input => (X, Y));
end Swap_Add_14;
```

Body in SPARK 2014:

```
package body Swap_Add_14
  with SPARK_Mode
is
   procedure Swap is
      Temporary: Integer;
   begin
      Temporary := X;
      X         := Y;
      Y         := Temporary;
   end Swap;

   function Add return Integer is (X + Y);
end Swap_Add_14;
```

A.2.2 Pre/Post/Return contracts

This example demonstrates how the *Pre/Post/Return* contracts are restructured and how they map from SPARK 2005 to SPARK 2014. Procedure *Swap* and function *Add* perform the same task as in the previous example, but the global variables have been replaced by parameters (this is not necessarry for proof) and they have been augmented by pre and post annotations. Two additional functions (*Max* and *Divide*) and one additional procedure (*Swap_Array_Elements*) have also been included in this example in order to demonstrate further features. *Max* returns the maximum of the two parameters. *Divide* returns the division of the two parameters after having ensured that the divisor is not equal to zero. The *Swap_Array_Elements* procedure swaps the contents of two elements of an array.

Specification in SPARK 2005:

```
package Swap_Add_Max_05 is

   subtype Index      is Integer range 1..100;
   type    Array_Type is array (Index) of Integer;

   procedure Swap (X, Y : in out Integer);
   --# post X = Y~ and Y = X~;

   function Add (X, Y : Integer) return Integer;
   --# pre ((X >= 0 and Y >= 0) -> (X + Y <= Integer'Last)) and
   --#     ((X <  0 and Y <  0) -> (X + Y >= Integer'First));
   --# return X + Y;

   function Max (X, Y : Integer) return Integer;
   --# return Z => (X >= Y -> Z = X) and
   --#              (Y >  X -> Z = Y);

   function Divide (X, Y : Integer) return Integer;
   --# pre Y /= 0 and X > Integer'First;
   --# return X / Y;

   procedure Swap_Array_Elements(I, J : Index; A: in out Array_Type);
   --# post A = A~[I => A~(J); J => A~(I)];

end Swap_Add_Max_05;
```

Body in SPARK 2005:

```
package body Swap_Add_Max_05
is
   procedure Swap (X, Y: in out Integer)
   is
      Temporary: Integer;
   begin
      Temporary := X;
      X         := Y;
      Y         := Temporary;
   end Swap;

   function Add (X, Y : Integer) return Integer
   is
   begin
```

(continues on next page)

(continued from previous page)

```
      return X + Y;
   end Add;

   function Max (X, Y : Integer) return Integer
   is
      Result: Integer;
   begin
      if X >= Y then
         Result := X;
      else
         Result := Y;
      end if;
      return Result;
   end Max;

   function Divide (X, Y : Integer) return Integer
   is
   begin
      return X / Y;
   end Divide;

   procedure Swap_Array_Elements(I, J : Index; A: in out Array_Type)
   is
      Temporary: Integer;
   begin
      Temporary := A(I);
      A(I)      := A(J);
      A(J)      := Temporary;
   end Swap_Array_Elements;

end Swap_Add_Max_05;
```

Specification in SPARK 2014:

```
package Swap_Add_Max_14
  with SPARK_Mode
is
   subtype Index      is Integer range 1..100;
   type    Array_Type is array (Index) of Integer;

   procedure Swap (X, Y : in out Integer)
     with Post => (X = Y'Old and Y = X'Old);

   function Add (X, Y : Integer) return Integer
     with Pre  => (if X >= 0 and Y >= 0 then X <= Integer'Last - Y
                  elsif X < 0 and Y < 0 then X >= Integer'First - Y),
          -- The precondition may be written as X + Y in Integer if
          -- an extended arithmetic mode is selected
          Post => Add'Result = X + Y;

   function Max (X, Y : Integer) return Integer
     with Post => Max'Result = (if X >= Y then X else Y);
```

(continues on next page)

(continued from previous page)

```

   function Divide (X, Y : Integer) return Integer
     with Pre  => Y /= 0 and X > Integer'First,
          Post => Divide'Result = X / Y;

   procedure Swap_Array_Elements(I, J : Index; A: in out Array_Type)
     with Post => A = A'Old'Update (I => A'Old (J),
                                    J => A'Old (I));
end Swap_Add_Max_14;
```

Body in SPARK 2014:

```
package body Swap_Add_Max_14
  with SPARK_Mode
is
   procedure Swap (X, Y : in out Integer) is
      Temporary: Integer;
   begin
      Temporary := X;
      X         := Y;
      Y         := Temporary;
   end Swap;

   function Add (X, Y : Integer) return Integer is (X + Y);

   function Max (X, Y : Integer) return Integer is
     (if X >= Y then X
      else Y);

   function Divide (X, Y : Integer) return Integer is (X / Y);

   procedure Swap_Array_Elements(I, J : Index; A: in out Array_Type) is
      Temporary: Integer;
   begin
      Temporary := A(I);
      A(I)      := A(J);
      A(J)      := Temporary;
   end Swap_Array_Elements;
end Swap_Add_Max_14;
```

A.2.3 Attributes of unconstrained out parameter in precondition

The following example illustrates the fact that the attributes of an unconstrained formal array parameter of mode “out” are permitted to appear in a precondition. The flow analyzer also needs to be smart about this, since it knows that X’First and X’Last are well-defined in the body, even though the content of X is not.

Specification in SPARK 2005:

```
package P
is
   type A is array (Positive range <>) of Integer;

```

(continues on next page)

(continued from previous page)

```
   -- Shows that X'First and X'Last _can_ be used in
   -- precondition here, even though X is mode "out"...
   procedure Init (X : out A);
   --# pre X'First = 1  and
   --#     X'Last  >= 20;
   --# post for all I in Positive range X'Range =>
   --#      ((I /= 20 -> (X (I) = 0)) and
   --#         (I = 1 -> (X (I) = X'Last)) and
   --#         (I = 20 -> (X (I) = -1)));

end P;
```

Body in SPARK 2005:

```
package body P is

   procedure Init (X : out A) is
   begin
      X := (others => 0);
      X (1) := X'Last;
      X (20) := -1;
  end Init;

end P;
```

Specification in SPARK 2014:

```
package P
  with SPARK_Mode
is
   type A is array (Positive range <>) of Integer;

   -- Shows that X'First, X'Last and X'Length _can_ be used
   -- in precondition here, even though X is mode "out"...
   procedure Init (X : out A)
     with Pre  => X'First = 1 and X'Last >= 20,
          Post => (for all I in X'Range =>
                     (if I = 1 then X (I) = X'Last
                      elsif I = 20 then X (I) = -1
                      else X (I) = 0));
end P;
```

Body in SPARK 2014:

```
package body P
  with SPARK_Mode
is
   procedure Init (X : out A) is
   begin
      X := (1 => X'Last, 20 => -1, others => 0);
   end Init;
end P;
```

A.2.4 Data Abstraction, Refinement and Initialization

This example demonstrates data abstraction and refinement. It also shows how abstract data is shown to be initialized during package elaboration (it need not be - it could be initialized through an explicit subprogram call, in which case the Initalizes annotation should not be given). There is also a demonstration of how procedures and functions can be nested within other procedures and functions. Furthermore, it illustrates how global variable refinement can be performed.

Specification in SPARK 2005:

```
package Nesting_Refinement_05
--# own State;
--# initializes State;
is
   procedure Operate_On_State;
   --# global in out State;
end Nesting_Refinement_05;
```

Body in SPARK 2005:

```
package body Nesting_Refinement_05
--# own State is X, Y;     -- Refined State
is
   X, Y: Integer;

   procedure Operate_On_State
   --# global in out X;    -- Refined Global
   --#           out Y;
   is
      Z: Integer;

      procedure Add_Z_To_X
      --# global in out X;
      --#        in     Z;
      is
      begin
         X := X + Z;
      end Add_Z_To_X;

      procedure Overwrite_Y_With_Z
      --# global    out Y;
      --#        in     Z;
      is
      begin
         Y := Z;
      end Overwrite_Y_With_Z;
   begin
      Z := 5;
      Add_Z_To_X;
      Overwrite_Y_With_Z;
   end Operate_On_State;

begin -- Promised to initialize State
      -- (which consists of X and Y)
   X := 10;
```

(continues on next page)

(continued from previous page)

```
   Y := 20;
end Nesting_Refinement_05;
```

Specification in SPARK 2014:

```
package Nesting_Refinement_14
  with SPARK_Mode,
       Abstract_State => State,
       Initializes    => State
is
   procedure Operate_On_State
     with Global   => (In_Out => State);
end Nesting_Refinement_14;
```

Body in SPARK 2014:

```
package body Nesting_Refinement_14
  -- State is refined onto two concrete variables X and Y
  with SPARK_Mode,
       Refined_State => (State => (X, Y))
is
   X, Y: Integer;

   procedure Operate_On_State
     with Refined_Global => (In_Out => X,
                             Output => Y)
   is
      Z: Integer;

      procedure Add_Z_To_X
        with Global => (In_Out => X,
                        Input  => Z)
      is
      begin
         X := X + Z;
      end Add_Z_To_X;

      procedure Overwrite_Y_With_Z
        with Global => (Output => Y,
                        Input  => Z)
      is
      begin
         Y := Z;
      end Overwrite_Y_With_Z;
   begin
      Z := 5;
      Add_Z_To_X;
      Overwrite_Y_With_Z;
   end Operate_On_State;

begin
   -- Promised to initialize State
```

(continues on next page)

(continued from previous page)

```
   -- (which consists of X and Y)
   X := 10;
   Y := 20;
end Nesting_Refinement_14;
```

A.3 Package patterns

A.3.1 Abstract Data Types (ADTs)

Visible type

The following example adds no mapping information. The SPARK 2005 and SPARK 2014 versions of the code are identical. Only the specification of the SPARK 2005 code will be presented. The reason why this code is being provided is to allow for a comparison between a package that is purely public and an equivalent one that also has private elements.

Specification in SPARK 2005:

```
package Stacks_05 is
   Stack_Size : constant := 100;
   type Pointer_Range is range 0 .. Stack_Size;
   subtype Index_Range is Pointer_Range range 1 .. Stack_Size;
   type Vector is array(Index_Range) of Integer;

   type Stack is
      record
         Stack_Vector : Vector;
         Stack_Pointer : Pointer_Range;
      end record;

   function Is_Empty(S : Stack) return Boolean;
   function Is_Full(S : Stack) return Boolean;

   procedure Clear(S : out Stack);
   procedure Push(S : in out Stack; X : in Integer);
   procedure Pop(S : in out Stack; X : out Integer);
end Stacks_05;
```

Private type

Similarly to the previous example, this one does not contain any annotations either. Due to this, the SPARK 2005 and SPARK 2014 versions are exactly the same. Only the specification of the 2005 version shall be presented.

Specification in SPARK 2005:

```
package Stacks_05 is

   type Stack is private;

   function Is_Empty(S : Stack) return Boolean;
   function Is_Full(S : Stack) return Boolean;
```

(continues on next page)

(continued from previous page)

```

   procedure Clear(S : out Stack);
   procedure Push(S : in out Stack; X : in Integer);
   procedure Pop(S : in out Stack; X : out Integer);

private
   Stack_Size : constant := 100;
   type Pointer_Range is range 0 .. Stack_Size;
   subtype Index_Range is Pointer_Range range 1 .. Stack_Size;
   type Vector is array(Index_Range) of Integer;

   type Stack is
      record
         Stack_Vector : Vector;
         Stack_Pointer : Pointer_Range;
      end record;
end Stacks_05;
```

Private type with pre/post contracts

This example demonstrates how *pre* and *post* conditions of subprograms may be specified in terms of functions declared in the same package specification. The function declarations are completed in the body and the postconditions of the completed functions are used to prove the implementations of the other subprograms. In SPARK 2014 explicit postconditions do not have to be specified on the bodies of the functions as they are implemented as expression functions and the expression, E, of the function acts as a default refined postcondition, i.e., F'Result = E. Note that the SPARK 2014 version is proven entirely automatically whereas the SPARK 2005 version requires user defined proof rules.

Specification in SPARK 2005:

```
package Stacks_05
is

   type Stack is private;

   function Is_Empty(S : Stack) return Boolean;
   function Is_Full(S : Stack) return Boolean;

   procedure Clear(S : in out Stack);
   --# post Is_Empty(S);
   procedure Push(S : in out Stack; X : in Integer);
   --# pre  not Is_Full(S);
   --# post not Is_Empty(S);
   procedure Pop(S : in out Stack; X : out Integer);
   --# pre  not Is_Empty(S);
   --# post not Is_Full(S);

private
   Stack_Size : constant := 100;
   type    Pointer_Range is range 0 .. Stack_Size;
   subtype Index_Range   is Pointer_Range range 1 .. Stack_Size;
   type    Vector        is array(Index_Range) of Integer;
```

(continues on next page)

(continued from previous page)

```

   type Stack is
      record
         Stack_Vector  : Vector;
         Stack_Pointer : Pointer_Range;
      end record;
end Stacks_05;
```

Body in SPARK 2005:

```
package body Stacks_05 is

   function Is_Empty (S : Stack) return Boolean
   --# return S.Stack_Pointer = 0;
   is
   begin
      return S.Stack_Pointer = 0;
   end Is_Empty;

   function Is_Full (S : Stack) return Boolean
   --# return S.Stack_Pointer = Stack_Size;
   is
   begin
      return S.Stack_Pointer = Stack_Size;
   end Is_Full;

   procedure Clear (S : in out Stack)
   --# post Is_Empty(S);
   is
   begin
      S.Stack_Pointer := 0;
   end Clear;

   procedure Push (S : in out Stack; X : in Integer)
   is
   begin
      S.Stack_Pointer := S.Stack_Pointer + 1;
      S.Stack_Vector (S.Stack_Pointer) := X;
   end Push;

   procedure Pop (S : in out Stack; X : out Integer)
   is
   begin
      X := S.Stack_Vector (S.Stack_Pointer);
      S.Stack_Pointer := S.Stack_Pointer - 1;
   end Pop;
end Stacks_05;
```

Specification in SPARK 2014:

```
package Stacks_14
  with SPARK_Mode
```

(continues on next page)

(continued from previous page)

```
is
   type Stack is private;

   function Is_Empty(S : Stack) return Boolean;
   function Is_Full(S : Stack) return Boolean;

   procedure Clear(S : in out Stack)
     with Post => Is_Empty(S);

   procedure Push(S : in out Stack; X : in Integer)
     with Pre  => not Is_Full(S),
          Post => not Is_Empty(S);

   procedure Pop(S : in out Stack; X : out Integer)
     with Pre  => not Is_Empty(S),
          Post => not Is_Full(S);

private
   Stack_Size : constant := 100;
   type    Pointer_Range is range 0 .. Stack_Size;
   subtype Index_Range   is Pointer_Range range 1 .. Stack_Size;
   type    Vector        is array(Index_Range) of Integer;

   type Stack is record
      Stack_Vector  : Vector;
      Stack_Pointer : Pointer_Range;
   end record;
end Stacks_14;
```

Body in SPARK 2014:

```
package body Stacks_14
  with SPARK_Mode
is
   -- Expression function has default refined postcondition of
   -- Is_Empty'Result = (S.Stack_Pointer = 0)
   function Is_Empty(S : Stack) return Boolean is (S.Stack_Pointer = 0);

   -- Expression function has default refined postcondition of
   -- Is_Empty'Result = (S.Stack_Pointer = Stack_Size)
   function Is_Full(S : Stack) return Boolean is (S.Stack_Pointer = Stack_Size);

   procedure Clear(S : in out Stack) is
   begin
      S.Stack_Pointer := 0;
   end Clear;

   procedure Push(S : in out Stack; X : in Integer) is
   begin
      S.Stack_Pointer := S.Stack_Pointer + 1;
      S.Stack_Vector(S.Stack_Pointer) := X;
   end Push;
```

(continues on next page)

(continued from previous page)

```

   procedure Pop(S : in out Stack; X : out Integer) is
   begin
      X := S.Stack_Vector(S.Stack_Pointer);
      S.Stack_Pointer := S.Stack_Pointer - 1;
   end Pop;
end Stacks_14;
```

Private/Public child visibility

The following example demonstrates visibility rules that apply between public children, private children and their parent in SPARK 2005. More specifically, it shows that:

- Private children are able to see their private siblings but not their public siblings.
- Public children are able to see their public siblings but not their private siblings.
- All children have access to their parent but the parent can only access private children.

Applying the SPARK tools on the following files will produce certain errors. This was intentionally done in order to illustrate both legal and illegal access attempts.

SPARK 2014 shares Ada's visibility rules. No restrictions have been applied in terms of visibility. Note that SPARK 2014 code does not require Inherit annotations.

Specification of parent in SPARK 2005:

```
package Parent_05
is
   function F (X : Integer) return Integer;
   function G (X : Integer) return Integer;
end Parent_05;
```

Specification of private child A in SPARK 2005:

```
--#inherit Parent_05; -- OK
private package Parent_05.Private_Child_A_05
is
   function F (X : Integer) return Integer;
end Parent_05.Private_Child_A_05;
```

Specification of private child B in SPARK 2005:

```
--#inherit Parent_05.Private_Child_A_05, -- OK
--#        Parent_05.Public_Child_A_05;  -- error, public sibling
private package Parent_05.Private_Child_B_05
is
   function H (X : Integer) return Integer;
end Parent_05.Private_Child_B_05;
```

Specification of public child A in SPARK 2005:

```
--#inherit Parent_05,                    -- OK
--#        Parent_05.Private_Child_A_05;  -- error, private sibling
package Parent_05.Public_Child_A_05
```

(continues on next page)

(continued from previous page)

```
is
  function G (X : Integer) return Integer;
end Parent_05.Public_Child_A_05;
```

Specification of public child B in SPARK 2005:

```
--#inherit Parent_05.Public_Child_A_05; -- OK
package Parent_05.Public_Child_B_05
is
   function H (X : Integer) return Integer;
end Parent_05.Public_Child_B_05;
```

Body of parent in SPARK 2005:

```
with Parent_05.Private_Child_A_05,   -- OK
     Parent_05.Public_Child_A_05;    -- error, public children not visible
package body Parent_05
is
   function F (X : Integer) return Integer is
   begin
      return Private_Child_A_05.F (X);
   end F;

   function G (X : Integer) return Integer is
   begin
      return Public_Child_A_05.G (X);
   end G;

end Parent_05;
```

Body of public child A in SPARK 2005:

```
package body Parent_05.Public_Child_A_05
is
   function G (X : Integer) return Integer is
      Result : Integer;
   begin
      if X <= 0 then
         Result := 0;
      else
         Result := Parent_05.F (X);  -- OK
      end if;
      return Result;
   end G;
end Parent_05.Public_Child_A_05;
```

Body of public child B in SPARK 2005:

```
with Parent_05.Private_Child_B_05;
package body Parent_05.Public_Child_B_05
is
   function H (X : Integer) return Integer is
   begin
```

(continues on next page)

(continued from previous page)

```
      return Parent_05.Private_Child_B_05.H (X);
   end H;
end Parent_05.Public_Child_B_05;
```

Body of private child B in SPARK 2005:

```
package body Parent_05.Private_Child_B_05
is
   function H (X : Integer) return Integer is
      Result : Integer;
   begin
      if X <= 10 then
         Result := 10;
      else
         Result := Parent_05.F (X);  -- Illegal in SPARK 2005
      end if;
      return Result;
   end H;
end Parent_05.Private_Child_B_05;
```

Specification of parent in SPARK 2014:

```
package Parent_14
  with SPARK_Mode
is
   function F (X : Integer) return Integer;
   function G (X : Integer) return Integer;
end Parent_14;
```

Specification of private child A in SPARK 2014:

```
private package Parent_14.Private_Child_A_14
  with SPARK_Mode
is
   function F (X : Integer) return Integer
     with Global   => null;
end Parent_14.Private_Child_A_14;
```

Specification of private child B in SPARK 2014:

```
private package Parent_14.Private_Child_B_14
  with SPARK_Mode
is
   function H (X : Integer) return Integer;
end Parent_14.Private_Child_B_14;
```

Specification of public child A in SPARK 2014:

```
package Parent_14.Public_Child_A_14
  with SPARK_Mode
is
  function G (X : Integer) return Integer;
end Parent_14.Public_Child_A_14;
```

Specification of public child B in SPARK 2014:

```
package Parent_14.Public_Child_B_14
  with SPARK_Mode
is
   function H (X : Integer) return Integer;
end Parent_14.Public_Child_B_14;
```

Body of parent in SPARK 2014:

```
with Parent_14.Private_Child_A_14,   -- OK
     Parent_14.Public_Child_A_14;    -- OK

package body Parent_14
  with SPARK_Mode
is
   function F (X : Integer) return Integer is (Private_Child_A_14.F (X));

   function G (X : Integer) return Integer is (Public_Child_A_14.G (X));
end Parent_14;
```

Body of public child A in SPARK 2014:

```
package body Parent_14.Public_Child_A_14
  with SPARK_Mode
is
   function G (X : Integer) return Integer is
     (if X <= 0 then 0
      else Parent_14.F (X));  -- OK
end Parent_14.Public_Child_A_14;
```

Body of public child B in SPARK 2014:

```
with Parent_14.Private_Child_B_14;

package body Parent_14.Public_Child_B_14
  with SPARK_Mode
is
   function H (X : Integer) return Integer is
     (Parent_14.Private_Child_B_14.H (X));
end Parent_14.Public_Child_B_14;
```

Body of private child B in SPARK 2014:

```
package body Parent_14.Private_Child_B_14
  with SPARK_Mode
is
   function H (X : Integer) return Integer is
     (if X <= 10 then 10
      else Parent_14.F (X));  -- Legal in SPARK 2014
end Parent_14.Private_Child_B_14;
```

A.3.2 Abstract State Machines (ASMs)

Visible, concrete state

Initialized by declaration

The example that follows presents a way in SPARK 2005 of initializing a concrete own variable (a state that is not refined) at the point of the declaration of the variables that compose it. Generally it is not good practice to declare several concrete own variables, data abstraction should be used but here we are doing it for the point of illustration.

In SPARK 2014 the client's view of package state is either visible (declared in the visible part of the package) or a state abstraction representing hidden state. A variable cannot overload the name of a state abstraction and therefore a state abstraction must be completed by a refinement given in the body of the package - there is no concept of a concrete state abstraction. The constituents of a state abstraction may be initialized at their declaration.

Specification in SPARK 2005:

```
package Stack_05
--# own S, Pointer;    -- concrete state
--# initializes S, Pointer;
is
   procedure Push(X : in Integer);
   --# global in out S, Pointer;

   procedure Pop(X : out Integer);
   --# global in S; in out Pointer;
end Stack_05;
```

Body in SPARK 2005:

```
package body Stack_05
is
   Stack_Size : constant := 100;
   type    Pointer_Range is range 0 .. Stack_Size;
   subtype Index_Range   is Pointer_Range range 1..Stack_Size;
   type    Vector        is array(Index_Range) of Integer;

   S : Vector := Vector'(Index_Range => 0);  -- Initialization of S
   Pointer : Pointer_Range := 0;             -- Initialization of Pointer

   procedure Push(X : in Integer)
   is
   begin
      Pointer := Pointer + 1;
      S(Pointer) := X;
   end Push;

   procedure Pop(X : out Integer)
   is
   begin
      X := S(Pointer);
      Pointer := Pointer - 1;
   end Pop;
```

(continues on next page)

(continued from previous page)

```

end Stack_05;
```

Specification in SPARK 2014:

```
package Stack_14
  with SPARK_Mode,
       Abstract_State => (S_State, Pointer_State),
       Initializes    => (S_State, Pointer_State)
is
   procedure Push(X : in Integer)
     with Global => (In_Out => (S_State, Pointer_State));

   procedure Pop(X : out Integer)
     with Global => (Input   => S_State,
                     In_Out => Pointer_State);
end Stack_14;
```

Body in SPARK 2014:

```
package body Stack_14
  with SPARK_Mode,
       Refined_State => (S_State       => S,
                         Pointer_State => Pointer)
is
   Stack_Size : constant := 100;
   type    Pointer_Range is range 0 .. Stack_Size;
   subtype Index_Range   is Pointer_Range range 1..Stack_Size;
   type    Vector        is array(Index_Range) of Integer;

   S : Vector := Vector'(Index_Range => 0);  -- Initialization of S
   Pointer : Pointer_Range := 0;             -- Initialization of Pointer

   procedure Push (X : in Integer)
     with Refined_Global => (In_Out => (S, Pointer))
   is
   begin
      Pointer := Pointer + 1;
      S (Pointer) := X;
   end Push;

   procedure Pop(X : out Integer)
     with Refined_Global => (Input  => S,
                             In_Out => Pointer)
   is
   begin
      X := S (Pointer);
      Pointer := Pointer - 1;
   end Pop;
end Stack_14;
```

Initialized by elaboration

The following example presents how a package's concrete state can be initialized at the statements section of the body. The specifications of both SPARK 2005 and SPARK 2014 are not presented since they are identical to the specifications of the previous example.

Body in SPARK 2005:

```
package body Stack_05
is
   Stack_Size : constant := 100;
   type    Pointer_Range is range 0 .. Stack_Size;
   subtype Index_Range   is Pointer_Range range 1..Stack_Size;
   type    Vector        is array(Index_Range) of Integer;

   S : Vector;
   Pointer : Pointer_Range;

   procedure Push(X : in Integer)
   is
   begin
      Pointer := Pointer + 1;
      S(Pointer) := X;
   end Push;

   procedure Pop(X : out Integer)
   is
   begin
      X := S(Pointer);
      Pointer := Pointer - 1;
   end Pop;

begin  -- initialization
   Pointer := 0;
   S := Vector'(Index_Range => 0);
end Stack_05;
```

Body in SPARK 2014:

```
package body Stack_14
  with SPARK_Mode,
       Refined_State => (S_State       => S,
                         Pointer_State => Pointer)
is
   Stack_Size : constant := 100;
   type    Pointer_Range is range 0 .. Stack_Size;
   subtype Index_Range   is Pointer_Range range 1..Stack_Size;
   type    Vector        is array(Index_Range) of Integer;

   S : Vector;
   Pointer : Pointer_Range;

   procedure Push (X : in Integer)
     with Refined_Global => (In_Out => (S, Pointer))
```

(continues on next page)

(continued from previous page)

```
   is
   begin
      Pointer := Pointer + 1;
      S (Pointer) := X;
   end Push;

   procedure Pop(X : out Integer)
     with Refined_Global => (Input  => S,
                             In_Out => Pointer)
   is
   begin
      X := S (Pointer);
      Pointer := Pointer - 1;
   end Pop;
begin
   -- initialization
   Pointer := 0;
   S := Vector'(Index_Range => 0);
end Stack_14;
```

Private, concrete state

In SPARK 2005 variables declared in the private part of a package are considered to be concrete own variables. In SPARK 2014 they are hidden state and must be constituents of a state abstraction.

The SPARK 2005 body has not been included since it does not contain any annotations.

Specification in SPARK 2005:

```
package Stack_05
--# own S, Pointer;
is
   procedure Push(X : in Integer);
   --# global in out S, Pointer;

   procedure Pop(X : out Integer);
   --# global in     S;
   --#        in out Pointer;
private
   Stack_Size : constant := 100;
   type    Pointer_Range is range 0 .. Stack_Size;
   subtype Index_Range   is Pointer_Range range 1..Stack_Size;
   type    Vector        is array(Index_Range) of Integer;

   S : Vector;
   Pointer : Pointer_Range;
end Stack_05;
```

Specification in SPARK 2014:

```
package Stack_14
  with SPARK_Mode,
```

(continues on next page)

(continued from previous page)

```
      Abstract_State => (S_State, Pointer_State)
is
   procedure Push(X : in Integer)
     with Global => (In_Out => (S_State, Pointer_State));

   procedure Pop(X : out Integer)
     with Global => (Input  => S_State,
                     In_Out => Pointer_State);

private
   Stack_Size : constant := 100;
   type    Pointer_Range is range 0 .. Stack_Size;
   subtype Index_Range   is Pointer_Range range 1..Stack_Size;
   type    Vector        is array(Index_Range) of Integer;

   S       : Vector with Part_Of => S_State;
   Pointer : Pointer_Range with Part_Of => Pointer_State;
end Stack_14;
```

Body in SPARK 2014:

```
package body Stack_14
  with SPARK_Mode,
      Refined_State => (S_State       => S,
                        Pointer_State => Pointer)
is
   procedure Push(X : in Integer)
     with Refined_Global => (In_Out => (S, Pointer))
   is
   begin
      Pointer := Pointer + 1;
      S (Pointer) := X;
   end Push;

   procedure Pop (X : out Integer)
     with Refined_Global => (Input  => S,
                             In_Out => Pointer)
   is
   begin
      X := S (Pointer);
      Pointer := Pointer - 1;
   end Pop;
end Stack_14;
```

Private, abstract state, refining onto concrete states in body

Initialized by procedure call

In this example, the abstract state declared at the specification is refined at the body. Procedure *Init* can be invoked by users of the package, in order to initialize the state.

Specification in SPARK 2005:

```
package Stack_05
--# own State;
is
   procedure Push(X : in Integer);
   --# global in out State;

   procedure Pop(X : out Integer);
   --# global in out State;

   procedure Init;
   --# global    out State;

end Stack_05;
```

Body in SPARK 2005:

```
package body Stack_05
--# own State is S, Pointer;
is
   Stack_Size : constant := 100;
   type    Pointer_Range is range 0 .. Stack_Size;
   subtype Index_Range   is Pointer_Range range 1..Stack_Size;
   type    Vector        is array(Index_Range) of Integer;

   Pointer : Pointer_Range;
   S       : Vector;

   procedure Push(X : in Integer)
   --# global in out Pointer, S;
   is
   begin
      Pointer := Pointer + 1;
      S(Pointer) := X;
   end Push;

   procedure Pop(X : out Integer)
   --# global in     S;
   --#        in out Pointer;
   is
   begin
      X := S(Pointer);
      Pointer := Pointer - 1;
   end Pop;

   procedure Init
```

(continues on next page)

(continued from previous page)

```
   --# global    out Pointer, S;
   is
   begin
      Pointer := 0;
      S := Vector'(Index_Range => 0);
   end Init;
end Stack_05;
```

Specification in SPARK 2014:

```
package Stack_14
  with SPARK_Mode,
       Abstract_State => State
is
   procedure Push(X : in Integer)
     with Global => (In_Out => State);

   procedure Pop(X : out Integer)
     with Global => (In_Out => State);

   procedure Init
     with Global => (Output => State);
end Stack_14;
```

Body in SPARK 2014:

```
package body Stack_14
  with SPARK_Mode,
       Refined_State => (State => (Pointer, S))
is
   Stack_Size : constant := 100;
   type    Pointer_Range is range 0 .. Stack_Size;
   subtype Index_Range   is Pointer_Range range 1..Stack_Size;
   type    Vector        is array(Index_Range) of Integer;

   Pointer : Pointer_Range;
   S       : Vector;

   procedure Push(X : in Integer)
     with Refined_Global => (In_Out => (Pointer, S))
   is
   begin
      Pointer := Pointer + 1;
      S (Pointer) := X;
   end Push;

   procedure Pop (X : out Integer)
     with Refined_Global => (In_Out => Pointer,
                             Input  => S)
   is
   begin
      X := S (Pointer);
```

(continues on next page)

(continued from previous page)

```
      Pointer := Pointer - 1;
   end Pop;

   procedure Init
     with Refined_Global => (Output => (Pointer, S))
   is
   begin
      Pointer := 0;
      S := (Index_Range => 0);
   end Init;
end Stack_14;
```

Initialized by elaboration of declaration

The example that follows introduces an abstract state at the specification and refines it at the body. The constituents of the abstract state are initialized at declaration.

Specification in SPARK 2005:

```
package Stack_05
--# own State;
--# initializes State;
is
   procedure Push(X : in Integer);
   --# global in out State;

   procedure Pop(X : out Integer);
   --# global in out State;

end Stack_05;
```

Body in SPARK 2005:

```
package body Stack_05
--# own State is Pointer, S; -- refinement of state
is
   Stack_Size : constant := 100;
   type    Pointer_Range is range 0 .. Stack_Size;
   subtype Index_Range   is Pointer_Range range 1..Stack_Size;
   type    Vector        is array(Index_Range) of Integer;

   S : Vector := Vector'(others => 0);
   Pointer : Pointer_Range := 0;
   -- initialization by elaboration of declaration

   procedure Push(X : in Integer)
   --# global in out Pointer, S;
   is
   begin
      Pointer := Pointer + 1;
      S(Pointer) := X;
```

(continues on next page)

(continued from previous page)

```
   end Push;

   procedure Pop(X : out Integer)
   --# global in     S;
   --#        in out Pointer;
   is
   begin
      X := S(Pointer);
      Pointer := Pointer - 1;
   end Pop;
end Stack_05;
```

Specification in SPARK 2014:

```
package Stack_14
  with SPARK_Mode,
       Abstract_State => State,
       Initializes    => State
is
   procedure Push(X : in Integer)
     with Global => (In_Out => State);

   procedure Pop(X : out Integer)
     with Global => (In_Out => State);
end Stack_14;
```

Body in SPARK 2014:

```
package body Stack_14
  with SPARK_Mode,
       Refined_State => (State => (Pointer, S)) -- refinement of state
is
   Stack_Size : constant := 100;
   type    Pointer_Range is range 0 .. Stack_Size;
   subtype Index_Range   is Pointer_Range range 1..Stack_Size;
   type    Vector        is array(Index_Range) of Integer;

   S : Vector := (others => 0);
   Pointer : Pointer_Range := 0;
   -- initialization by elaboration of declaration

   procedure Push(X : in Integer)
     with Refined_Global => (In_Out => (Pointer, S))
   is
   begin
      Pointer := Pointer + 1;
      S (Pointer) := X;
   end Push;

   procedure Pop(X : out Integer)
     with Refined_Global => (In_Out => Pointer,
                             Input  => S)
```

(continues on next page)

(continued from previous page)

```
   is
   begin
      X := S (Pointer);
      Pointer := Pointer - 1;
   end Pop;
end Stack_14;
```

Initialized by package body statements

This example introduces an abstract state at the specification and refines it at the body. The constituents of the abstract state are initialized at the statements part of the body. The specifications of the SPARK 2005 and SPARK 2014 versions of the code are as in the previous example and have thus not been included.

Body in SPARK 2005:

```
package body Stack_05
--# own State is Pointer, S;  -- refinement of state
is
   Stack_Size : constant := 100;
   type    Pointer_Range is range 0 .. Stack_Size;
   subtype Index_Range   is Pointer_Range range 1..Stack_Size;
   type    Vector        is array(Index_Range) of Integer;

   S : Vector;
   Pointer : Pointer_Range;

   procedure Push(X : in Integer)
   --# global in out Pointer, S;
   is
   begin
      Pointer := Pointer + 1;
      S(Pointer) := X;
   end Push;

   procedure Pop(X : out Integer)
   --# global in out Pointer;
   --#        in     S;
   is
   begin
      X := S(Pointer);
      Pointer := Pointer - 1;
   end Pop;
begin  -- initialized by package body statements
   Pointer := 0;
   S := Vector'(Index_Range => 0);
end Stack_05;
```

Body in SPARK 2014:

```
package body Stack_14
  with SPARK_Mode,
```

(continues on next page)

(continued from previous page)

```
      Refined_State => (State => (Pointer, S))  -- refinement of state
is
   Stack_Size : constant := 100;
   type    Pointer_Range is range 0 .. Stack_Size;
   subtype Index_Range   is Pointer_Range range 1..Stack_Size;
   type    Vector        is array(Index_Range) of Integer;

   S       : Vector;
   Pointer : Pointer_Range;

   procedure Push(X : in Integer)
     with Refined_Global => (In_Out => (Pointer, S))
   is
   begin
      Pointer := Pointer + 1;
      S (Pointer) := X;
   end Push;

   procedure Pop(X : out Integer)
     with Refined_Global => (In_Out => Pointer,
                             Input  => S)
   is
   begin
      X := S (Pointer);
      Pointer := Pointer - 1;
   end Pop;
begin
   -- initialized by package body statements
   Pointer := 0;
   S := (Index_Range => 0);
end Stack_14;
```

Initialized by mixture of declaration and statements

This example introduces an abstract state at the specification and refines it at the body. Some of the constituents of the abstract state are initialized during their declaration and the rest at the statements part of the body.

Specification in SPARK 2005:

```
package Stack_05
--# own Stack;
--# initializes Stack;
is
   procedure Push(X : in Integer);
   --# global in out Stack;

   procedure Pop(X : out Integer);
   --# global in out Stack;

end Stack_05;
```

Body in SPARK 2005:

```
package body Stack_05
--# own Stack is S, Pointer; -- state refinement
is
   Stack_Size : constant := 100;
   type    Pointer_Range is range 0 .. Stack_Size;
   subtype Index_Range   is Pointer_Range range 1..Stack_Size;
   type    Vector        is array(Index_Range) of Integer;
   S : Vector;

   Pointer : Pointer_Range := 0;
   -- initialization by elaboration of declaration

   procedure Push(X : in Integer)
   --# global in out S, Pointer;
   is
   begin
      Pointer := Pointer + 1;
      S(Pointer) := X;
   end Push;

   procedure Pop(X : out Integer)
   --# global in     S;
   --#        in out Pointer;
   is
   begin
      X := S(Pointer);
      Pointer := Pointer - 1;
   end Pop;
begin  -- initialization by body statements
   S := Vector'(Index_Range => 0);
end Stack_05;
```

Specification in SPARK 2014:

```
package Stack_14
  with SPARK_Mode,
       Abstract_State => Stack,
       Initializes    => Stack
is
   procedure Push(X : in Integer)
     with Global => (In_Out => Stack);

   procedure Pop(X : out Integer)
     with Global => (In_Out => Stack);
end Stack_14;
```

Body in SPARK 2014:

```
package body Stack_14
  with SPARK_Mode,
       Refined_State => (Stack => (S, Pointer)) -- state refinement
is
   Stack_Size : constant := 100;
```

(continues on next page)

(continued from previous page)

```
   type    Pointer_Range is range 0 .. Stack_Size;
   subtype Index_Range   is Pointer_Range range 1..Stack_Size;
   type    Vector        is array(Index_Range) of Integer;

   S       : Vector; -- left uninitialized
   Pointer : Pointer_Range := 0;
   -- initialization by elaboration of declaration

   procedure Push(X : in Integer)
     with Refined_Global => (In_Out => (S, Pointer))
   is
   begin
      Pointer := Pointer + 1;
      S (Pointer) := X;
   end Push;

   procedure Pop (X : out Integer)
     with Refined_Global => (Input  => S,
                             In_Out => Pointer)
   is
   begin
      X := S (Pointer);
      Pointer := Pointer - 1;
   end Pop;
begin
   -- partial initialization by body statements
   S := (Index_Range => 0);
end Stack_14;
```

Initial condition

This example introduces a new SPARK 2014 feature that did not exist in SPARK 2005. On top of declaring an abstract state and promising to initialize it, we also illustrate certain conditions that will be valid after initialization. There is a verification condition to show that immediately after the elaboration of the package that the specified Initial_Condition is True. Checks will be generated that have to be proven (or executed at run-time) to show that the initial condition is True.

Specification in SPARK 2014:

```
package Stack_14
  with SPARK_Mode,
       Abstract_State    => State,
       Initializes       => State,
       Initial_Condition => Is_Empty  -- Stating that Is_Empty holds
                                      -- after initialization
is
   function Is_Empty return Boolean
     with Global => State;

   function Is_Full return Boolean
     with Global => State;
```

(continues on next page)

(continued from previous page)

```

   function Top return Integer
     with Global => State,
          Pre    => not Is_Empty;

   procedure Push (X: in Integer)
     with Global => (In_Out => State),
          Pre    => not Is_Full,
          Post   => Top = X;

   procedure Pop (X: out Integer)
     with Global => (In_Out => State),
          Pre    => not Is_Empty;
end Stack_14;
```

Body in SPARK 2014:

```
package body Stack_14
  with SPARK_Mode,
       Refined_State => (State => (S,
                                   Pointer)) -- State refinement
is
   Max_Stack_Size : constant := 1024;
   type Pointer_Range is range 0 .. Max_Stack_Size;
   subtype Index_Range is Pointer_Range range 1 .. Max_Stack_Size;
   type Vector is array (Index_Range) of Integer;

   -- Declaration of constituents
   S       : Vector;
   Pointer : Pointer_Range;

   -- The subprogram contracts are refined in terms of the constituents.
   -- Expression functions could be used where applicable

   function Is_Empty  return Boolean is (Pointer = 0)
     with Refined_Global => Pointer;

   function Is_Full  return Boolean is (Pointer = Max_Stack_Size)
     with Refined_Global => Pointer;

   function Top return Integer is (S (Pointer))
     with Refined_Global => (Pointer, S);

   procedure Push(X: in Integer)
     with Refined_Global => (In_Out => (Pointer, S))
   is
   begin
      Pointer := Pointer + 1;
      S (Pointer) := X;
   end Push;

   procedure Pop(X: out Integer)
```

(continues on next page)

(continued from previous page)

```
      with Refined_Global => (Input  => S,
                              In_Out => Pointer)
   is
   begin
      X := S (Pointer);
      Pointer := Pointer - 1;
   end Pop;
begin
   -- Initialization - we promised to initialize the state
   -- and that initially the stack will be empty
   Pointer := 0;  -- Is_Empty is True.
   S := Vector'(Index_Range => 0);
end Stack_14;
```

Private, abstract state, refining onto state of private child

The following example shows a parent package Power that contains an own variable (a state abstraction). This state abstraction is refined onto state abstractions of two private children Source_A and Source_B.

Specification of Parent in SPARK 2005:

```
-- Use of child packages to encapsulate state
package Power_05
--# own State;
--# initializes State;
is
   procedure Read_Power(Level : out Integer);
   --# global State;
   --# derives Level from State;
end Power_05;
```

Body of Parent in SPARK 2005:

```
with Power_05.Source_A_05, Power_05.Source_B_05;

package body Power_05
--# own State is Power_05.Source_A_05.State,
--#              Power_05.Source_B_05.State;
is

  procedure Read_Power(Level : out Integer)
  --# global Source_A_05.State, Source_B_05.State;
  --# derives
  --#     Level
  --#     from
  --#         Source_A_05.State,
  --#         Source_B_05.State;
  is
     Level_A : Integer;
     Level_B : Integer;
  begin
     Source_A_05.Read (Level_A);
```

(continues on next page)

(continued from previous page)

```
    Source_B_05.Read (Level_B);
    Level := Level_A + Level_B;
  end Read_Power;

end Power_05;
```

Specifications of Private Children in SPARK 2005:

```
--# inherit Power_05;
private package Power_05.Source_A_05
--# own State;
--# initializes State;
is
   procedure Read (Level : out Integer);
   --# global State;
   --# derives Level from State;
end Power_05.Source_A_05;
```

```
--# inherit Power_05;
private package Power_05.Source_B_05
--# own State;
--# initializes State;
is
   procedure Read (Level : out Integer);
   --# global State;
   --# derives Level from State;
end Power_05.Source_B_05;
```

Bodies of Private Children in SPARK 2005:

```
package body Power_05.Source_A_05
--# own State is S;
is
   S : Integer := 0;

   procedure Read (Level : out Integer)
   --# global in S;
   --# derives Level from S;
   is
   begin
      Level := S;
   end Read;
end Power_05.Source_A_05;
```

```
package body Power_05.Source_B_05
--# own State is S;
is
   S : Integer := 0;

   procedure Read (Level : out Integer)
   --# global in S;
   --# derives Level from S;
```

(continues on next page)

(continued from previous page)

```
   is
   begin
      Level := S;
   end Read;
end Power_05.Source_B_05;
```

Specification of Parent in SPARK 2014:

```
-- Use of child packages to encapsulate state
package Power_14
  with SPARK_Mode,
       Abstract_State => State,
       Initializes    => State
is
   procedure Read_Power(Level : out Integer)
     with Global  => State,
          Depends => (Level => State);
end Power_14;
```

Body of Parent in SPARK 2014:

```
with Power_14.Source_A_14,
     Power_14.Source_B_14;

package body Power_14
  with SPARK_Mode,
       Refined_State => (State => (Power_14.Source_A_14.State,
                                   Power_14.Source_B_14.State))
is
   procedure Read_Power(Level : out Integer)
     with Refined_Global  => (Source_A_14.State, Source_B_14.State),
          Refined_Depends => (Level => (Source_A_14.State,
                                        Source_B_14.State))
   is
      Level_A : Integer;
      Level_B : Integer;
   begin
      Source_A_14.Read (Level_A);
      Source_B_14.Read (Level_B);
      Level := Level_A + Level_B;
   end Read_Power;
end Power_14;
```

Specifications of Private Children in SPARK 2014:

```
private package Power_14.Source_A_14
  with SPARK_Mode,
       Abstract_State => (State with Part_Of =>Power_14.State),
       Initializes    => State
is
   procedure Read (Level : out Integer)
     with Global => State,
          Depends => (Level => State);
```

(continues on next page)

(continued from previous page)

```
end Power_14.Source_A_14;
```

```
private package Power_14.Source_B_14
  with SPARK_Mode,
       Abstract_State => (State with Part_Of => Power_14.State),
       Initializes    => State
is
   procedure Read (Level : out Integer)
     with Global  => State,
          Depends => (Level => State);
end Power_14.Source_B_14;
```

Bodies of Private Children in SPARK 2014:

```
package body Power_14.Source_A_14
  with SPARK_Mode,
       Refined_State => (State => S)
is
   S : Integer := 0;

   procedure Read (Level : out Integer)
     with Refined_Global  => (Input => S),
          Refined_Depends => (Level => S)
   is
   begin
      Level := S;
   end Read;
end Power_14.Source_A_14;
```

```
package body Power_14.Source_B_14
  with SPARK_Mode,
       Refined_State => (State => S)
is
   S : Integer := 0;

   procedure Read (Level : out Integer)
     with Refined_Global  => (Input => S),
          Refined_Depends => (Level => S)
   is
   begin
      Level := S;
   end Read;
end Power_14.Source_B_14;
```

Private, abstract state, refining onto concrete state of embedded package

This example is based around the packages from section *Private, abstract state, refining onto concrete state of embedded package*, with the private child packages converted into embedded packages and the refinement onto concrete visible state.

Specification in SPARK 2005:

```
-- Use of embedded packages to encapsulate state
package Power_05
--# own State;
is
   procedure Read_Power(Level : out Integer);
   --# global State;
   --# derives Level from State;
end Power_05;
```

Body in SPARK 2005:

```
package body Power_05
--# own State is Source_A.State,
--#              Source_B.State;
is

  --  Embedded package spec for Source_A
  package Source_A
  --# own State;
  is
     procedure Read (Level : out Integer);
      --# global State;
      --# derives Level from State;
  end Source_A;

  --  Embedded package spec for Source_B.
  package Source_B
  --# own State;
  is
    procedure Read (Level : out Integer);
    --# global State;
    --# derives Level from State;
  end Source_B;

  --  Embedded package body for Source_A
  package body Source_A
  is
    State : Integer;

    procedure Read (Level : out Integer)
    is
    begin
      Level := State;
    end Read;
  end Source_A;

```

(continues on next page)

(continued from previous page)

```
  --  Embedded package body for Source_B
  package body Source_B
  is
    State : Integer;

    procedure Read (Level : out Integer)
    is
    begin
      Level := State;
    end Read;

  end Source_B;

  procedure Read_Power(Level : out Integer)
  --# global Source_A.State, Source_B.State;
  --# derives
  --#     Level
  --#     from
  --#         Source_A.State,
  --#         Source_B.State;
  is
     Level_A : Integer;
     Level_B : Integer;
  begin
     Source_A. Read (Level_A);
     Source_B.Read (Level_B);
     Level := Level_A + Level_B;
  end Read_Power;

end Power_05;
```

Specification in SPARK 2014:

```
-- Use of embedded packages to encapsulate state
package Power_14
  with SPARK_Mode,
       Abstract_State => State,
       Initializes    => State
is
   procedure Read_Power(Level : out Integer)
     with Global  => State,
          Depends => (Level => State);
end Power_14;
```

Body in SPARK 2014:

```
package body Power_14
  with SPARK_Mode,
       Refined_State => (State => (Source_A.State,
                                   Source_B.State))
is
   --  Embedded package spec for Source_A
```

(continues on next page)

(continued from previous page)

```
   package Source_A
     with Initializes => State
   is
      State : Integer := 0;

      procedure Read (Level : out Integer)
        with Global  => State,
             Depends => (Level => State);
   end Source_A;

   --  Embedded package spec for Source_B.
   package Source_B
     with Initializes => State
   is
      State : Integer := 0;

      procedure Read (Level : out Integer)
        with Global  => State,
             Depends => (Level => State);
   end Source_B;

   --  Embedded package body for Source_A
   package body Source_A is
      procedure Read (Level : out Integer) is
      begin
         Level := State;
      end Read;
   end Source_A;

   --  Embedded package body for Source_B
   package body Source_B is
      procedure Read (Level : out Integer) is
      begin
         Level := State;
      end Read;
   end Source_B;

   procedure Read_Power(Level : out Integer)
     with Refined_Global  => (Source_A.State,
                              Source_B.State),
          Refined_Depends => (Level => (Source_A.State,
                                        Source_B.State))
   is
      Level_A : Integer;
      Level_B : Integer;
   begin
      Source_A. Read (Level_A);
      Source_B.Read (Level_B);
      Level := Level_A + Level_B;
   end Read_Power;
end Power_14;
```

Private, abstract state, refining onto mixture of the above

This example is based around the packages from sections *Private, abstract state, refining onto state of private child* and *Private, abstract state, refining onto concrete state of embedded package*. Source_A is an embedded package, while Source_B is a private child. In order to avoid repetition, the code of this example is not being presented.

A.3.3 External Variables

Basic Input and Output Device Drivers

The following example shows a main program - Copy - that reads all available data from a given input port, stores it internally during the reading process in a stack and then outputs all the data read to an output port. The specifications of the stack packages are not presented since they are identical to previous examples.

Specification of main program in SPARK 2005:

```
with Input_Port_05, Output_Port_05, Stacks_05;
--# inherit Input_Port_05, Output_Port_05, Stacks_05;
--# main_program;
procedure Copy_05
--# global in     Input_Port_05.Input_State;
--#        out    Output_Port_05.Output_State;
--# derives Output_Port_05.Output_State from Input_Port_05.Input_State;
is
   The_Stack   : Stacks_05.Stack;
   Value       : Integer;
   Done        : Boolean;
   Final_Value : constant Integer := 999;
begin
   Stacks_05.Clear(The_Stack);
   loop
      Input_Port_05.Read_From_Port(Value);
      Stacks_05.Push(The_Stack, Value);
      Done := Value = Final_Value;
      exit when Done;
   end loop;
   loop
      Stacks_05.Pop(The_Stack, Value);
      Output_Port_05.Write_To_Port(Value);
      exit when Stacks_05.Is_Empty(The_Stack);
   end loop;
end Copy_05;
```

Specification of input port in SPARK 2005:

```
package Input_Port_05
  --# own in Input_State;
is
   procedure Read_From_Port(Input_Value : out Integer);
   --# global in Input_State;
   --# derives Input_Value from Input_State;

end Input_Port_05;
```

Body of input port in SPARK 2005:

```
package body Input_Port_05
is

   Input_State : Integer;
   for Input_State'Address use
     System.Storage_Elements.To_Address (16#ACECAE0#);
   pragma Volatile (Input_State);

   procedure Read_From_Port(Input_Value : out Integer)
   is
   begin
      Input_Value := Input_State;
   end Read_From_Port;

end Input_Port_05;
```

Specification of output port in SPARK 2005:

```
package Output_Port_05
  --# own out Output_State;
is
   procedure Write_To_Port(Output_Value : in Integer);
   --# global out Output_State;
   --# derives Output_State from Output_Value;
end Output_Port_05;
```

Body of output port in SPARK 2005:

```
package body Output_Port_05
is

   Output_State : Integer;
   for Output_State'Address use
     System.Storage_Elements.To_Address (16#ACECAF0#);
   pragma Volatile (Output_State);

   procedure Write_To_Port(Output_Value : in Integer)
   is
   begin
      Output_State := Output_Value;
   end Write_To_Port;

end Output_Port_05;
```

Specification of main program in SPARK 2014:

```
with Input_Port_14,
     Output_Port_14,
     Stacks_14;
--  No need to specify that Copy_14 is a main program

procedure Copy_14
```

(continues on next page)

(continued from previous page)

```
  with SPARK_Mode,
       Global  => (Input  => Input_Port_14.Input_State,
                   Output => Output_Port_14.Output_State),
       Depends => (Output_Port_14.Output_State => Input_Port_14.Input_State)
is
   The_Stack   : Stacks_14.Stack;
   Value       : Integer;
   Done        : Boolean;
   Final_Value : constant Integer := 999;
begin
   Stacks_14.Clear(The_Stack);
   loop
      Input_Port_14.Read_From_Port(Value);
      Stacks_14.Push(The_Stack, Value);
      Done := Value = Final_Value;
      exit when Done;
   end loop;
   loop
      Stacks_14.Pop(The_Stack, Value);
      Output_Port_14.Write_To_Port(Value);
      exit when Stacks_14.Is_Empty(The_Stack);
   end loop;
end Copy_14;
```

Specification of input port in SPARK 2014:

```
package Input_Port_14
  with SPARK_Mode,
       Abstract_State => (Input_State with External => Async_Writers)
is
   procedure Read_From_Port(Input_Value : out Integer)
     with Global  => (Input => Input_State),
          Depends => (Input_Value => Input_State);
end Input_Port_14;
```

Specification of output port in SPARK 2014:

```
package Output_Port_14
  with SPARK_Mode,
       Abstract_State => (Output_State with External => Async_Readers)
is
   procedure Write_To_Port(Output_Value : in Integer)
     with Global  => (Output => Output_State),
          Depends => (Output_State => Output_Value);
end Output_Port_14;
```

Body of input port in SPARK 2014:

This is as per SPARK 2005, but uses aspects instead of representation clauses and pragmas.

```
with System.Storage_Elements;

package body Input_Port_14
```

(continues on next page)

(continued from previous page)

```
  with SPARK_Mode,
       Refined_State => (Input_State => Input_S)
is
   Input_S : Integer
     with Volatile,
          Async_Writers,
          Address => System.Storage_Elements.To_Address (16#ACECAE0#);

   procedure Read_From_Port(Input_Value : out Integer)
     with Refined_Global  => (Input => Input_S),
          Refined_Depends => (Input_Value => Input_S)
   is
   begin
      Input_Value := Input_S;
   end Read_From_Port;
end Input_Port_14;
```

Body of output port in SPARK 2014:

This is as per SPARK 2005, but uses aspects instead of representation clauses and pragmas.

```
with System.Storage_Elements;

package body Output_Port_14
  with SPARK_Mode,
       Refined_State => (Output_State => Output_S)
is
   Output_S : Integer
     with Volatile,
          Async_Readers,
          Address => System.Storage_Elements.To_Address (16#ACECAF0#);

   procedure Write_To_Port(Output_Value : in Integer)
     with Refined_Global  => (Output => Output_S),
          Refined_Depends => (Output_S => Output_Value)
   is
   begin
      Output_S := Output_Value;
   end Write_To_Port;
end Output_Port_14;
```

Input driver using 'Tail in a contract

This example uses the Input_Port package from section *Basic Input and Output Device Drivers* and adds a contract using the 'Tail attribute. The example also use the Always_Valid attribute in order to allow proof to succeed (otherwise, there is no guarantee in the proof context that the value read from the port is of the correct type).

SPARK 2014 does not have the attribute 'Tail but, often, an equivalent proof can be achieved using assert pragmas. Neither is there a direct equivalent of the Always_Valid attribute but the paragma Assume may be used to the same effect.

Specification in SPARK 2005:

```
package Input_Port
  --# own in Inputs : Integer;
is
   procedure Read_From_Port(Input_Value : out Integer);
   --# global in Inputs;
   --# derives Input_Value from Inputs;
   --# post (Inputs~ = 0  -> (Input_Value = Inputs'Tail (Inputs~))) and
   --#      (Inputs~ /= 0 -> (Input_Value = Inputs~));

end Input_Port;
```

Body in SPARK 2005:

```
package body Input_Port
is

   Inputs : Integer;
   for Inputs'Address use
     System.Storage_Elements.To_Address (16#ACECAF0#);

   --# assert Inputs'Always_Valid;
   pragma Volatile (Inputs);

   procedure Read_From_Port(Input_Value : out Integer)
   is
   begin
      Input_Value := Inputs;
      if Input_Value = 0 then
         Input_Value := Inputs;
      end if;
   end Read_From_Port;

end Input_Port;
```

Specification in SPARK 2014:

```
package Input_Port_14
  with SPARK_Mode,
       Abstract_State => (Inputs with External => Async_Writers)
is
   procedure Read_From_Port(Input_Value : out Integer)
     with Global  => Inputs,
          Depends => (Input_Value => Inputs);
end Input_Port_14;
```

Body in SPARK 2014:

```
with System.Storage_Elements;

package body Input_Port_14
  with SPARK_Mode,
       Refined_State => (Inputs => Input_Port)
is
```

(continues on next page)

(continued from previous page)

```
   Input_Port : Integer
     with Volatile,
          Async_Writers,
          Address => System.Storage_Elements.To_Address (16#ACECAF0#);

   procedure Read_From_Port(Input_Value : out Integer)
     with Refined_Global  => Input_Port,
          Refined_Depends => (Input_Value => Input_Port)
   is
      First_Read  : Integer;
      Second_Read : Integer;
   begin
      Second_Read := Input_Port;    -- Ensure Second_Read is initialized
      pragma Assume (Second_Read'Valid);
      First_Read  := Second_Read;   -- but it is infact the First_Read.
      if First_Read = 0 then
         Second_Read := Input_Port; -- Now it is the Second_Read
         pragma Assume (Second_Read'Valid);
         Input_Value := Second_Read;
      else
         Input_Value := First_Read;
      end if;
      pragma Assert ((First_Read = 0 and then Input_Value = Second_Read)
                     or else (Input_Value = First_Read));
   end Read_From_Port;
end Input_Port_14;
```

Output driver using 'Append in a contract

This example uses the Output package from section *Basic Input and Output Device Drivers* and adds a contract using the 'Append attribute.

SPARK 2014 does not have the attribute 'Append but, often, an equivalent proof can be achieved using assert pragmas.

Specification in SPARK 2005:

```
package Output_Port
  --# own out Outputs : Integer;
is
   procedure Write_To_Port(Output_Value : in Integer);
   --# global out Outputs;
   --# derives Outputs from Output_Value;
   --# post ((Output_Value = -1) ->
   --#        (Outputs =
   --#           Outputs'Append (Outputs'Append (Outputs~, 0), Output_Value))
   --#  and
   --#      ((Output_Value /= -1) ->
   --#         (Outputs =
   --#           Outputs'Append (Outputs~, Output_Value));
end Output_Port;
```

Body in SPARK 2005:

```
package body Output_Port
is

   Outputs : Integer;
   for Outputs'Address use System.Storage_Elements.To_Address (16#ACECAF10#);
   pragma Volatile (Outputs);

   procedure Write_To_Port(Output_Value : in Integer)
   is
   begin
      if Output_Value = -1 then
         Outputs := 0;
      end if;

      Outputs := Output_Value;
   end Write_To_Port;

end Output_Port;
```

Specification in SPARK 2014:

```
package Output_Port_14
  with SPARK_Mode,
       Abstract_State => (Outputs with External => Async_Readers)
is
   procedure Write_To_Port(Output_Value : in Integer)
     with Global  => (Output => Outputs),
          Depends => (Outputs => Output_Value);
end Output_Port_14;
```

Body in SPARK 2014:

```
with System.Storage_Elements;

package body Output_Port_14
  with SPARK_Mode,
       Refined_State => (Outputs => Output_Port)
is
   Output_Port : Integer
     with Volatile,
          Async_Readers,
          Address => System.Storage_Elements.To_Address (16#ACECAF10#);

   -- This is a simple subprogram that always updates the Output_Shadow with
   -- the single value which is written to the output port.
   procedure Write_It (Output_Value : in Integer; Output_Shadow : out Integer)
     with Global  => (Output => Output_Port),
          Depends => ((Output_Port, Output_Shadow) => Output_Value),
          Post    => Output_Shadow = Output_Value
   is
   begin
      Output_Shadow := Output_Value;
      Output_Port := Output_Shadow;
```

(continues on next page)

(continued from previous page)

```
   end Write_It;


   procedure Write_To_Port(Output_Value : in Integer)
     with Refined_Global  => (Output => Output_Port),
          Refined_Depends => (Output_Port => Output_Value)
   is
      Out_1, Out_2 : Integer;
   begin
      if Output_Value = -1 then
         Write_It (0, Out_1);
         Write_It (Output_Value, Out_2);
      else
         Write_It (Output_Value, Out_1);
         Out_2 := Out_1;  -- Avoids flow error.
      end if;

      pragma Assert (if Output_Value = -1 then
                        Out_1 = 0 and Out_2 = Output_Value
                     else
                        Out_1 = Output_Value);
   end Write_To_Port;
end Output_Port_14;
```

Refinement of external state - voting input switch

The following example presents an abstract view of the reading of 3 individual switches and the voting performed on the values read.

Abstract Switch specification in SPARK 2005:

```
package Switch
--# own in State;
is

   type Reading is (on, off, unknown);

   function ReadValue return Reading;
   --# global in State;

end Switch;
```

Component Switch specifications in SPARK 2005:

```
--# inherit Switch;
private package Switch.Val1
--# own in State;
is
   function Read return Switch.Reading;
   --# global in State;

end Switch.Val1;
```

```
--# inherit Switch;
private package Switch.Val2
--# own in State;
is
   function Read return Switch.Reading;
   --# global in State;

end Switch.Val2;
```

```
--# inherit Switch;
private package Switch.Val3
--# own in State;
is
   function Read return Switch.Reading;
   --# global in State;

end Switch.Val3;
```

Switch body in SPARK 2005:

```
with Switch.Val1;
with Switch.Val2;
with Switch.Val3;
package body Switch
--# own State is in Switch.Val1.State,
--#              in Switch.Val2.State,
--#              in Switch.Val3.State;
is

   subtype Value is Integer range -1 .. 1;
   subtype Score is Integer range -3 .. 3;
   type ConvertToValueArray is array (Reading) of Value;
   type ConvertToReadingArray is array (Score) of Reading;

   ConvertToValue : constant ConvertToValueArray := ConvertToValueArray'(on => 1,
                                                                          unknown => 0,
                                                                          off => -1);
   ConvertToReading : constant ConvertToReadingArray :=
                                   ConvertToReadingArray'(-3 .. -2 => off,
                                                          -1 .. 1 => unknown,
                                                          2 ..3 => on);

   function ReadValue return Reading
   --# global in Val1.State;
   --#        in Val2.State;
   --#        in Val3.State;
   is
      A, B, C : Reading;
   begin
       A := Val1.Read;
       B := Val2.Read;
       C := Val3.Read;
       return ConvertToReading (ConvertToValue (A) +
```

(continues on next page)

(continued from previous page)

```
         ConvertToValue (B) + ConvertToValue (C));
   end ReadValue;

end Switch;
```

Abstract Switch specification in SPARK 2014:

```
package Switch
  with SPARK_Mode,
       Abstract_State => (State with External => Async_Writers)
is
   type Reading is (on, off, unknown);

   function ReadValue return Reading
     with Volatile_Function,
          Global => (Input => State);
end Switch;
```

Component Switch specifications in SPARK 2014:

```
private package Switch.Val1
  with SPARK_Mode,
       Abstract_State => (State with External => Async_Writers,
                                     Part_Of  => Switch.State)
is
   function Read return Switch.Reading
     with Volatile_Function,
          Global => (Input => State);
end Switch.Val1;
```

```
private package Switch.Val2
  with SPARK_Mode,
       Abstract_State => (State with External => Async_Writers,
                                     Part_Of  => Switch.State)
is
   function Read return Switch.Reading
     with Volatile_Function,
          Global => (Input => State);
end Switch.Val2;
```

```
private package Switch.Val3
  with SPARK_Mode,
       Abstract_State => (State with External => Async_Writers,
                                     Part_Of  => Switch.State)
is
   function Read return Switch.Reading
     with Volatile_Function,
          Global => (Input => State);
end Switch.Val3;
```

Switch body in SPARK 2014:

```
with Switch.Val1,
     Switch.Val2,
     Switch.Val3;

package body Switch
   -- State is refined onto three states, each of which has properties
   --  Volatile and Input
  with SPARK_Mode,
       Refined_State => (State => (Switch.Val1.State,
                                    Switch.Val2.State,
                                    Switch.Val3.State))
is
   subtype Value is Integer range -1 .. 1;
   subtype Score is Integer range -3 .. 3;
   type ConvertToValueArray is array (Reading) of Value;
   type ConvertToReadingArray is array (Score) of Reading;

   ConvertToValue : constant ConvertToValueArray :=
     ConvertToValueArray'(on => 1,
                          unknown => 0,
                          off => -1);
   ConvertToReading : constant ConvertToReadingArray :=
     ConvertToReadingArray'(-3 .. -2 => off,
                            -1 .. 1  => unknown,
                             2 .. 3  => on);

   function ReadValue return Reading
     with Refined_Global => (Input => (Val1.State, Val2.State, Val3.State))
   is
      A, B, C : Reading;
   begin
      A := Val1.Read;
      B := Val2.Read;
      C := Val3.Read;
      return ConvertToReading (ConvertToValue (A) +
        ConvertToValue (B) + ConvertToValue (C));
   end ReadValue;
end Switch;
```

Complex I/O Device

The following example illustrates a more complex I/O device: the device is fundamentally an output device but an acknowledgement has to be read from it. In addition, a local register stores the last value written to avoid writes that would just re-send the same value. The own variable is then refined into a normal variable, an input external variable and an output external variable.

Specification in SPARK 2005:

```
package Device
--# own State;
--# initializes State;
is
```

(continues on next page)

(continued from previous page)

```
  procedure Write (X : in Integer);
  --# global in out State;
  --# derives State from State, X;
end Device;
```

Body in SPARK 2005:

```
package body Device
--# own State is        OldX,
--#             in      StatusPort,
--#                 out Register;
-- refinement on to mix of external and ordinary variables
is
   type Status_Port_Type is mod 2**32;

  OldX : Integer := 0; -- only component that needs initialization
  StatusPort : Status_Port_Type;
  pragma Volatile (StatusPort);
  -- address clause would be added here

  Register : Integer;
  pragma Volatile (Register);
  -- address clause would be added here

  procedure WriteReg (X : in Integer)
  --# global out Register;
  --# derives Register from X;
  is
  begin
    Register := X;
  end WriteReg;

  procedure ReadAck (OK : out Boolean)
  --# global in StatusPort;
  --# derives OK from StatusPort;
  is
    RawValue : Status_Port_Type;
  begin
    RawValue := StatusPort; -- only assignment allowed here
    OK := RawValue = 16#FFFF_FFFF#;
  end ReadAck;

  procedure Write (X : in Integer)
  --# global in out OldX;
  --#           out Register;
  --#        in     StatusPort;
  --# derives OldX,Register from OldX, X &
  --#         null          from StatusPort;
  is
    OK : Boolean;
  begin
    if X /= OldX then
```

(continues on next page)

(continued from previous page)

```
      OldX := X;
      WriteReg (X);
      loop
        ReadAck (OK);
        exit when OK;
      end loop;
    end if;
  end Write;
end Device;
```

Specification in SPARK 2014:

```
package Device
  with SPARK_Mode,
       Abstract_State => (State with External => (Async_Readers,
                                                  Async_Writers)),
       Initializes    => State
is
  procedure Write (X : in Integer)
    with Global  => (In_Out => State),
         Depends => (State =>+ X);
end Device;
```

Body in SPARK 2014:

```
package body Device
  with SPARK_Mode,
       Refined_State => (State => (OldX,
                                   StatusPort,
                                   Register))
   -- refinement on to mix of external and ordinary variables
is
   type Status_Port_Type is mod 2**32;

   OldX : Integer := 0; -- only component that needs initialization

   StatusPort : Status_Port_Type
     with Volatile,
          Async_Writers;
   -- address clause would be added here

   Register : Integer
     with Volatile,
          Async_Readers;
   -- address clause would be added here

   procedure WriteReg (X : in Integer)
     with Global  => (Output => Register),
          Depends => (Register => X)
   is
   begin
      Register := X;
```

(continues on next page)

(continued from previous page)

```
   end WriteReg;

   procedure ReadAck (OK : out Boolean)
     with Global  => (Input => StatusPort),
          Depends => (OK => StatusPort)
   is
      RawValue : Status_Port_Type;
   begin
      RawValue := StatusPort; -- only assignment allowed here
      OK := RawValue = 16#FFFF_FFFF#;
   end ReadAck;

   procedure Write (X : in Integer)
     with Refined_Global  => (Input  => StatusPort,
                              Output => Register,
                              In_Out => OldX),
          Refined_Depends => ((OldX,
                               Register) => (OldX,
                                             X),
                               null => StatusPort)
   is
      OK : Boolean;
   begin
      if X /= OldX then
         OldX := X;
         WriteReg (X);
         loop
            ReadAck (OK);
            exit when OK;
         end loop;
      end if;
   end Write;
end Device;
```

Increasing values in input stream

The following example illustrates an input port from which values are read. According to its postcondition, procedure Increases checks whether the first values read from the sequence are in ascending order. This example shows that postconditions can refer to multiple individual elements of the input stream.

In SPARK 2014 we can use assert pragmas in the subprogram instead of specifying the action in the postcondition, as was done in *Input driver using 'Tail in a contract*. Another alternative, as shown in this example, is to use a formal parameter of a private type to keep a trace of the values read.

Specification in SPARK 2005:

```
 package Inc
--# own in Sensor : Integer;
is
   procedure Increases (Result : out Boolean;
                        Valid  : out Boolean);
   --# global in Sensor;
```

(continues on next page)

(continued from previous page)

```
   --# post Valid -> (Result <-> Sensor'Tail (Sensor~) > Sensor~);

end Inc;
```

Body in SPARK 2005:

```
with System.Storage_Elements;
package body Inc
-- Cannot refine own variable Sensor as it has been given a concrete type.
is
   Sensor : Integer;
   for Sensor'Address use System.Storage_Elements.To_Address (16#DEADBEE0#);
   pragma Volatile (Sensor);

   procedure Read (V     : out Integer;
                   Valid : out Boolean)
   --# global in Sensor;
   --# post (Valid -> V = Sensor~) and
   --#      (Sensor = Sensor'Tail (Sensor~));
   is
      Tmp : Integer;
   begin
      Tmp := Sensor;
      if Tmp'Valid then
         V := Tmp;
         Valid := True;
         --# check Sensor = Sensor'Tail (Sensor~);
      else
         V := 0;
         Valid := False;
      end if;
   end Read;

   procedure Increases (Result : out Boolean;
                        Valid  : out Boolean)
   is
      A, B : Integer;
   begin
      Result := False;
      Read (A, Valid);
      if Valid then
         Read (B, Valid);
         if Valid then
            Result := B > A;
         end if;
      end if;
   end Increases;

end Inc;
```

Specification in SPARK 2014:

```ada
package Inc
  with SPARK_Mode,
       Abstract_State => (Sensor with External => Async_Writers)
is
   -- Declare a private type which will keep a trace of the
   -- values read.
   type Increasing_Indicator is private;

   -- Access (ghost) functions for the private type only intended for
   -- use in pre and post conditions or other assertion expressions
   function First (Indicator : Increasing_Indicator) return Integer
     with Ghost;

   function Second (Indicator : Increasing_Indicator) return Integer
     with Ghost;

   -- Used to check that the value returned by procedure Increases
   -- is valid (Invalid values have not been read from the Sensor).
   function Is_Valid (Indicator : Increasing_Indicator) return Boolean;

   -- Use this function to determine whether the result of the procedure
   -- Increases indicates an increasing value.
   -- It can only be called if Is_Valid (Indicator)
   function Is_Increasing (Indicator : Increasing_Indicator) return Boolean
     with Pre => Is_Valid (Indicator);

   procedure Increases (Result : out Increasing_Indicator)
     with Global => Sensor,
          Post   => (if Is_Valid (Result) then Is_Increasing (Result)=
                        (Second (Result) > First (Result)));

private
   type Increasing_Indicator is record
      Valid : Boolean;
      First, Second : Integer;
   end record;
end Inc;
```

Body in SPARK 2014:

```ada
with System.Storage_Elements;

package body Inc
  with SPARK_Mode,
       Refined_State => (Sensor => S)
is
   pragma Warnings (Off);
   S : Integer
     with Volatile,
          Async_Writers,
          Address => System.Storage_Elements.To_Address (16#DEADBEE0#);
   pragma Warnings (On);

```

(continues on next page)

(continued from previous page)

```
   function First (Indicator : Increasing_Indicator) return Integer is
     (Indicator.First);

   function Second (Indicator : Increasing_Indicator) return Integer is
     (Indicator.Second);

   function Is_Valid (Indicator : Increasing_Indicator) return Boolean is
     (Indicator.Valid);

   function Is_Increasing (Indicator : Increasing_Indicator) return Boolean is
     (Indicator.Second > Indicator.First);

   pragma Warnings (Off);
   procedure Read (V     : out Integer;
                   Valid : out Boolean)
     with Global => S,
          Post   => (if Valid then V'Valid)
   is
      Tmp : Integer;
   begin
      pragma Warnings (On);
      Tmp := S;
      pragma Warnings (Off);
      if Tmp'Valid then
      pragma Warnings (On);
         V := Tmp;
         Valid := True;
      else
         V := 0;
         Valid := False;
      end if;
   end Read;

   procedure Increases (Result : out Increasing_Indicator)
     with Refined_Global => S
   is
   begin
      Read (Result.First, Result.Valid);
      if Result.Valid then
         Read (Result.Second, Result.Valid);
      else
         Result.Second := 0;
      end if;
   end Increases;
end Inc;
```

A.3.4 Package Inheritance

SPARK 2014 does not have the SPARK 2005 concept of package inheritance. It has the same package visibility rules as Ada.

Contracts with remote state

The following example illustrates indirect access to the state of one package by another via an intermediary. Raw_Data stores some data, which has preprocessing performed on it by Processing and on which Calculate performs some further processing (although the corresponding bodies are not given, Read_Calculated_Value in Calculate calls through to Read_Processed_Data in Processing, which calls through to Read in Raw_Data).

Specifications in SPARK 2005:

```
package Raw_Data
--# own State;
--# Initializes State;
is

   function Data_Is_Valid return Boolean;
   --# global State;

   function Get_Value return Integer;
   --# global State;

   procedure Read_Next;
   --# global in out State;
   --# derives State from State;


end Raw_Data;
```

```
with Raw_Data;
--# inherit Raw_Data;
package Processing
--# own State;
--# Initializes State;
is

   procedure Get_Processed_Data (Value : out Integer);
   --# global in     Raw_Data.State;
   --#        in out State;
   --# derives Value, State  from State, Raw_Data.State;
   --# pre Raw_Data.Data_Is_Valid (Raw_Data.State);

end Processing;
```

```
with Processing;
--# inherit Processing, Raw_Data;
package Calculate
is

   procedure Read_Calculated_Value (Value : out Integer);
```

(continues on next page)

(continued from previous page)

```
   --# global in out Processing.State;
   --#        in     Raw_Data.State;
   --# derives Value, Processing.State from Processing.State, Raw_Data.State;
   --# pre Raw_Data.Data_Is_Valid (Raw_Data.State);

end Calculate;
```

Specifications in SPARK 2014:

```
package Raw_Data
  with SPARK_Mode,
       Abstract_State => (State with External => Async_Writers),
       Initializes    => State
is
   function Data_Is_Valid return Boolean
     with Volatile_Function,
          Global => State;

   function Get_Value return Integer
     with Volatile_Function,
          Global => State;

   procedure Read_Next
     with Global  => (In_Out => State),
          Depends => (State => State);
end Raw_Data;
```

```
with Raw_Data;

package Processing
  with SPARK_Mode,
       Abstract_State => State
is
   procedure Get_Processed_Data (Value : out Integer)
     with Global  => (Input  => Raw_Data.State,
                      In_Out => State),
          Depends => ((Value,
                       State) => (State,
                                  Raw_Data.State)),
          Pre     => Raw_Data.Data_Is_Valid;
end Processing;
```

```
with Processing,
     Raw_Data;

package Calculate
  with SPARK_Mode
is
   procedure Read_Calculated_Value (Value : out Integer)
     with Global  => (In_Out => Processing.State,
                      Input  => Raw_Data.State),
          Depends => ((Value,
```

(continues on next page)

(continued from previous page)

```
                       Processing.State) => (Processing.State,
                                             Raw_Data.State)),
          Pre      => Raw_Data.Data_Is_Valid;
end Calculate;
```

Package nested inside package

See section *Private, abstract state, refining onto concrete state of embedded package*.

Package nested inside subprogram

This example is a modified version of that given in section *Refinement of external state - voting input switch*. It illustrates the use of a package nested within a subprogram.

Abstract Switch specification in SPARK 2005:

```
package Switch
--# own in State;
is

   type Reading is (on, off, unknown);

   function ReadValue return Reading;
   --# global in State;

end Switch;
```

Component Switch specifications in SPARK 2005:

As in *Refinement of external state - voting input switch*

Switch body in SPARK 2005:

```
with Switch.Val1;
with Switch.Val2;
with Switch.Val3;
package body Switch
--# own State is in Switch.Val1.State,
--#             in Switch.Val2.State,
--#             in Switch.Val3.State;
is

   subtype Value is Integer range -1 .. 1;
   subtype Score is Integer range -3 .. 3;


   function ReadValue return Reading
   --# global in Val1.State;
   --#        in Val2.State;
   --#        in Val3.State;
   is
      A, B, C : Reading;
```

(continues on next page)

(continued from previous page)

```

      --  Embedded package to provide the capability to synthesize three inputs
      --  into one.
      --# inherit Switch;
      package Conversion
      is

         function Convert_To_Reading
            (Val_A : Switch.Reading;
             Val_B : Switch.Reading;
             Val_C : Switch.Reading) return Switch.Reading;

      end Conversion;

      package body Conversion
      is

         type ConvertToValueArray is array (Switch.Reading) of Switch.Value;
         type ConvertToReadingArray is array (Switch.Score) of Switch.Reading;
         ConvertToValue : constant ConvertToValueArray := ConvertToValueArray'(Switch.on => 1,
                                                                              Switch.unknown => 0,
                                                                              Switch.off => -1);

         ConvertToReading : constant ConvertToReadingArray :=
                                      ConvertToReadingArray'(-3 .. -2 => Switch.off,
                                                             -1 .. 1  => Switch.unknown,
                                                             2 ..3    => Switch.on);

         function Convert_To_Reading
            (Val_A : Switch.Reading;
             Val_B : Switch.Reading;
             Val_C : Switch.Reading) return Switch.Reading
         is
         begin

            return ConvertToReading (ConvertToValue (Val_A) +
                   ConvertToValue (Val_B) + ConvertToValue (Val_C));
         end Convert_To_Reading;

      end Conversion;

   begin
       A := Val1.Read;
       B := Val2.Read;
       C := Val3.Read;
       return Conversion.Convert_To_Reading
                (Val_A => A,
                 Val_B => B,
                 Val_C => C);
```

(continues on next page)

(continued from previous page)

```
   end ReadValue;

end Switch;
```

Abstract Switch specification in SPARK 2014:

```
package Switch
  with SPARK_Mode,
       Abstract_State => (State with External => Async_Writers)
is
   type Reading is (on, off, unknown);

   function ReadValue return Reading
     with Volatile_Function,
          Global => (Input => State);
end Switch;
```

Component Switch specification in SPARK 2014:

As in *Refinement of external state - voting input switch*

Switch body in SPARK 2014:

```
with Switch.Val1,
     Switch.Val2,
     Switch.Val3;

package body Switch
  --  State is refined onto three states, each of which has properties
  --  Volatile and Input.
  with SPARK_Mode,
       Refined_State => (State => (Switch.Val1.State,
                                   Switch.Val2.State,
                                   Switch.Val3.State))
is
   subtype Value is Integer range -1 .. 1;
   subtype Score is Integer range -3 .. 3;

   function ReadValue return Reading
     with Refined_Global => (Input => (Val1.State, Val2.State, Val3.State))
   is
      A, B, C : Reading;

      --  Embedded package to provide the capability to synthesize three inputs
      --  into one.
      package Conversion is
         function Convert_To_Reading
           (Val_A : Switch.Reading;
            Val_B : Switch.Reading;
            Val_C : Switch.Reading) return Switch.Reading;
      end Conversion;

      package body Conversion is
```

(continues on next page)

(continued from previous page)

```
        type ConvertToValueArray is array (Switch.Reading) of Switch.Value;
        type ConvertToReadingArray is array (Switch.Score) of Switch.Reading;
        ConvertToValue : constant ConvertToValueArray :=
          ConvertToValueArray'(Switch.on => 1,
                               Switch.unknown => 0,
                               Switch.off => -1);

        ConvertToReading : constant ConvertToReadingArray :=
          ConvertToReadingArray'(-3 .. -2 => Switch.off,
                                 -1 .. 1  => Switch.unknown,
                                  2 .. 3  => Switch.on);

        function Convert_To_Reading
           (Val_A : Switch.Reading;
            Val_B : Switch.Reading;
            Val_C : Switch.Reading) return Switch.Reading is
          (ConvertToReading (ConvertToValue (Val_A) +
                             ConvertToValue (Val_B) +
                             ConvertToValue (Val_C)));
     end Conversion;
  begin  --  begin statement of ReadValue function
     A := Val1.Read;
     B := Val2.Read;
     C := Val3.Read;
     return Conversion.Convert_To_Reading
              (Val_A => A,
               Val_B => B,
               Val_C => C);
  end ReadValue;
end Switch;
```

Circular dependence and elaboration order

SPARK 2005 avoided issues of circular dependence and elaboration order dependencies through a combination of the inherit annotation and the restrictions that initialization expressions are constant, user defined subprograms cannot be called in the sequence of statements of a package body and a package can only initialize variables declared in its delarative part.

SPARK 2014 does not have the inherit annotation and only enforces the restriction that a package can only initialize an object declared in its declarative region. Hence, in SPARK 2014 two package bodies that depend on each other's specification may be legal, as is calling a user defined subprogram.

Instead of the elaboration restrictions of SPARK 2005 a set of rules is applied in SPARK 2014 which determines when elaboration order control pragmas such as Elaborate_Body or Elaborate_All are required. These rules ensure the absence of elaboration order dependencies.

Examples of the features of SPARK 2014 elaboration order rules are given below. In the example described below the partial elaboration order would be either of P_14 or Q_14 specifications first followed by P_14 body because of the Elaborate_All on the specification of R_14 specification and the body of Q_14, then the elaboration of Q_14 body or the specification of R_14 and the body of R_14 after the elaboration of Q_14. Elaboration order dependencies are avoided by the (required) use of elaboration control pragmas.

Package Specifications in SPARK 2014:

```
package P_14
  with SPARK_Mode,
       Abstract_State => P_State,
       Initializes    => (P_State, Global_Var),
       Elaborate_Body
is
   Global_Var : Integer;

   procedure Init (S : out Integer);
end P_14;
```

```
package Q_14
  with SPARK_Mode,
       Abstract_State => Q_State,
       Initializes    => Q_State
is
   type T is new Integer;

   procedure Init (S : out T);
end Q_14;
```

```
with P_14;
pragma Elaborate_All (P_14); -- Required because P_14.Global_Var
                             -- Is mentioned as input in the Initializes aspect
package R_14
  with SPARK_Mode,
       Abstract_State => State,
       Initializes    => (State => P_14.Global_Var)
is
   procedure Op ( X : in Positive)
     with Global => (In_Out => State);
end R_14;
```

Package Bodies in SPARK 2014

```
with Q_14;

package body P_14
  with SPARK_Mode,
       Refined_State => (P_State => P_S)
is
   P_S : Q_14.T;  -- The use of type Q.T does not require
                  -- the body of Q to be elaborated.

   procedure Init (S : out Integer) is
   begin
      S := 5;
   end Init;
begin
   -- Cannot call Q_14.Init here beacuse
   -- this would require an Elaborate_All for Q_14
   -- and would be detected as a circularity
```

(continues on next page)

(continued from previous page)

```
   Init (Global_Var);
   P_S := Q_14.T (Global_Var);
end P_14;
```

```
with P_14;
pragma Elaborate_All (P_14); -- Required because the elaboration of the
                             -- body of Q_14 (indirectly) calls P_14.Init
package body Q_14
  with SPARK_Mode,
       Refined_State => (Q_State => Q_S)
is
   Q_S : T;

   procedure Init (S : out T) is
      V : Integer;
   begin
      P_14.Init (V);
      if V > 0 and then  V <= Integer'Last - 5 then
         S := T(V + 5);
      else
         S := 5;
      end if;
   end Init;
begin
   Init (Q_S);
end Q_14;
```

```
with Q_14;
pragma Elaborate_All (Q_14); -- Required because Q_14.Init is called
                             -- in the elaboration of the body of R_14
use type Q_14.T;

package body R_14
  with SPARK_Mode,
       Refined_State => (State => R_S)
is
   R_S : Q_14.T;
   procedure Op ( X : in Positive)
     with Refined_Global => (In_Out => R_S)
   is
   begin
      if R_S <= Q_14.T'Last - Q_14.T (X) then
         R_S := R_S + Q_14.T (X);
      else
         R_S := 0;
      end if;
   end Op;
begin
   Q_14.Init (R_S);
   if P_14.Global_Var > 0
     and then R_S <= Q_14.T'Last - Q_14.T (P_14.Global_Var)
```

(continues on next page)

(continued from previous page)

```
   then
      R_S := R_S + Q_14.T (P_14.Global_Var);
   else
      R_S := Q_14.T (P_14.Global_Var);
   end if;
end R_14;
```

A.4 Bodies and Proof

A.4.1 Assert, Assume, Check contracts

Assert (in loop) contract

The following example demonstrates how the SPARK 2005 *assert* annotation is used inside a loop as a loop invariant. It cuts the loop and on each iteration of the loop the list of existing hypotheses for the path is cleared. A verification condition is generated to prove that the assert expression is True, and the expression is the basis of the new hypotheses.

SPARK 2014 has a specific pragma for defining a loop invariant, *pragma Loop_Invariant* which is more sophisticated than the SPARK 2005 assert annotation and often requires less conditions in the invariant expression than in SPARK 2005. As in SPARK 2005 a default loop invariant will be used if one is not provided which, often, may be sufficient to prove absence of run-time exceptions. Like all SPARK 2014 assertion expressions the loop invariant is executable.

Note in the example below the SPARK 2014 version proves absence of run-time exceptions without an explicit loop invariant being provided.

Specification in SPARK 2005:

```
package Assert_Loop_05
is
   subtype Index is Integer range 1 .. 10;
   type A_Type is Array (Index) of Integer;

   function Value_present (A: A_Type; X : Integer) return Boolean;
   --# return for some M in Index => (A (M) = X);
end Assert_Loop_05;
```

Body in SPARK 2005:

```
package body Assert_Loop_05
is
   function Value_Present (A: A_Type; X : Integer) return Boolean
   is
      I : Index := Index'First;
   begin
      while A (I) /= X and I < Index'Last loop
         --# assert I < Index'Last and
         --#        (for all M in Index range Index'First .. I => (A (M) /= X));
         I := I + 1;
      end loop;
      return A (I) = X;
   end Value_Present;
end Assert_Loop_05;
```

Specification in SPARK 2014:

```
package Assert_Loop_14
  with SPARK_Mode
is
   subtype Index is Integer range 1 .. 10;
   type A_Type is Array (Index) of Integer;

   function Value_present (A : A_Type; X : Integer) return Boolean
     with Post => Value_present'Result = (for some M in Index => A (M) = X);
end Assert_Loop_14;
```

Body in SPARK 2014:

```
package body Assert_Loop_14
  with SPARK_Mode
is
   function Value_Present (A : A_Type; X : Integer) return Boolean is
      I : Index := Index'First;
   begin
      while A (I) /= X and I < Index'Last loop
         pragma Loop_Variant (Increases => I);
         pragma Loop_Invariant
           (I < Index'Last
            and (for all M in Index'First .. I => A (M) /= X));
         I := I + 1;
      end loop;

      return A (I) = X;
   end Value_Present;
end Assert_Loop_14;
```

Assert (no loop) contract

While not in a loop, the SPARK 2005 *assert* annotation maps to *pragma Assert_And_Cut* in SPARK 2014. Both the assert annotation and pragma assert clear the list of hypotheses on the path, generate a verification condition to prove the assertion expression and use the assertion expression as the basis of the new hypotheses.

Assume contract

The following example illustrates use of an Assume annotation. The assumed expression does not generate a verification condition and is not proved (although it is executed in SPARK 2014 if assertion expressions are not ignored at run-time).

In this example, the Assume annotation is effectively being used to implement the SPARK 2005 Always_Valid attribute.

Specification for Assume annotation in SPARK 2005:

```
package Input_Port
  --# own in Inputs;
is
   procedure Read_From_Port(Input_Value : out Integer);
   --# global in Inputs;
   --# derives Input_Value from Inputs;
```

(continues on next page)

(continued from previous page)

```

end Input_Port;
```

Body for Assume annotation in SPARK 2005:

```
with System.Storage_Elements;
package body Input_Port
is

   Inputs : Integer;
   for Inputs'Address use System.Storage_Elements.To_Address (16#CAFE0#);
   pragma Volatile (Inputs);

   procedure Read_From_Port(Input_Value : out Integer)
   is
   begin
      --# assume Inputs in Integer;
      Input_Value := Inputs;
   end Read_From_Port;

end Input_Port;
```

Specification for Assume annotation in SPARK 2014:

```
package Input_Port
  with SPARK_Mode,
       Abstract_State => (State_Inputs with External => Async_Writers)
is
   procedure Read_From_Port(Input_Value : out Integer)
     with Global  => (Input => State_Inputs),
          Depends => (Input_Value => State_Inputs);
end Input_Port;
```

Body for Assume annotation in SPARK 2014:

```
with System.Storage_Elements;

package body Input_Port
  with SPARK_Mode,
       Refined_State => (State_Inputs => Inputs)
is
   Inputs : Integer
     with Volatile,
          Async_Writers,
          Address => System.Storage_Elements.To_Address (16#CAFE0#);

   procedure Read_From_Port(Input_Value : out Integer)
     with Refined_Global  => (Input => Inputs),
          Refined_Depends => (Input_Value => Inputs)
   is
   begin
      Input_Value := Inputs;
      pragma Assume(Input_Value in Integer);
```

(continues on next page)

(continued from previous page)

```
   end Read_From_Port;
end Input_Port;
```

Check contract

The SPARK 2005 *check* annotation is replaced by *pragma assert* in SPARK 2014. This annotation generates a verification condition to prove the checked expression and adds the expression as a new hypothesis to the list of existing hypotheses.

Specification for Check annotation in SPARK 2005:

```
package Check_05
is
   subtype Small is Integer range 1 .. 10;
   subtype Big   is Integer range 1 .. 21;

   procedure Compare(A, B : in Small; C : in out Big);
end Check_05;
```

Body for Check annotation in SPARK 2005:

```
package body Check_05
is
   procedure Compare(A, B : in Small; C : in out Big)
   is
   begin
      if (A + B >= C) then
         C := A;
         C := C + B;
         C := C + 1;
      end if;
      --# check A + B < C;
   end Compare;
end Check_05;
```

Specification for Check annotation in SPARK 2014:

```
package Check_14
  with SPARK_Mode
is
   subtype Small is Integer range 1 .. 10;
   subtype Big   is Integer range 1 .. 21;

   procedure Compare (A, B : in Small; C : in out Big);
end Check_14;
```

Body for Check annotation in SPARK 2014:

```
package body Check_14
  with SPARK_Mode
is
   procedure Compare(A, B : in Small; C : in out Big) is
```

(continues on next page)

(continued from previous page)

```
   begin
      if A + B >= C then
         C := A;
         C := C + B;
         C := C + 1;
      end if;
      pragma Assert (A + B < C);
   end Compare;
end Check_14;
```

A.4.2 Assert used to control path explosion

This capability is in general not needed with the SPARK 2014 toolset where path explosion is handled automatically. In the rare cases where this is needed you can use *pragma Assert_And_Cut*.

A.5 Other Contracts and Annotations

A.5.1 Always_Valid assertion

See section *Input driver using 'Tail in a contract* for use of an assertion involving the Always_Valid attribute.

A.5.2 Rule declaration annotation

See section *Proof types and proof functions*.

A.5.3 Proof types and proof functions

The following example gives pre- and postconditions on operations that act upon the concrete representation of an abstract own variable. This means that proof functions and proof types are needed to state those pre- and postconditions. In addition, it gives an example of the use of a rule declaration annotation - in the body of procedure Initialize - to introduce a rule related to the components of a constant record value.

SPARK 2014 does not have a direct equivalent of proof types and proof functions. State abstractions cannot have a type and all functions in SPARK 2014 are Ada functions. Functions may be defined to be ghost functions which means that they can only be called within an assertion expression such as a pre or postcondition. Assertion expressions may be executed or ignored at run-time and if they are ignored Ghost functions behave much like SPARK 2005 proof functions.

Rule declaration annotations for structured constants are not required in SPARK 2014.

The SPARK 2005 version of the example given below will require user defined proof rules to discharge the proofs because refined definitions of some of the proof functions cannot be provided as they would have different formal parameters. The SPARK 2014 version does not suffer from this problem as functions called within assertion expressions may have global items.

Specification in SPARK 2005:

```
package Stack
--# own State : Abstract_Stack;
is
```

(continues on next page)

(continued from previous page)

```
   --  It is not possible to specify that the stack will be
   --  initialized to empty except by having an initialization
   --  subprogram called during program execution (as opposed to
   --  package elaboration).

   --  Proof functions to indicate whether or not the Stack is empty
   --  and whether or not it is full.
   --# type Abstract_Stack is abstract;

   --# function Max_Stack_Size return Natural;

   --  Proof function to give the number of elements on the stack.
   --# function Count(Input : Abstract_Stack) return Natural;

   --  Proof function returns the Nth entry on the stack.
   --  Stack_Entry (Count (State)) is the top of stack
   --# function Stack_Entry (N : Natural; S : Abstract_Stack) return Integer;
   --# pre N in 1 .. Count (S);
   --  A refined version of this function cannot be written because
   --  the abstract view has a formal parameter of type Abstract_Stack
   --  whereas the refined view would not have this parameter but use
   --  a global. A user defined proof rule would be required to define
   --  this function. Alternatively, it could be written as an Ada
   --  function where the the global and formal parameter views would
   --  be available. However, the function would then be callable and
   --  generate implementation code.

   --# function Is_Empty(Input : Abstract_Stack) return Boolean;
   --# return Count (Input) = 0;

   --# function Is_Full(Input : Abstract_Stack) return Boolean;
   --# return Count (Input) = Max_Stack_Size;

   --  The precondition requires the stack is not full when a value, X,
   --  is pushed onto it.
   --  The postcondition indicates that the count of the stack will be
   --  incremented after a push and therefore the stack will be non-empty.
   --  The item X is now the top of the stack.
   procedure Push(X : in Integer);
   --# global in out State;
   --# pre  not Is_Full(State);
   --# post Count (State) = Count (State~) + 1 and
   --#      Count (State) <= Max_Stack_Size and
   --#      Stack_Entry (Count (State), State) = X;

   --  The precondition requires the stack is not empty when we
   --  pull a value from it.
   --  The postcondition indicates the stack count is decremented.
   procedure Pop (X : out Integer);
   --# global in out State;
   --# pre not Is_Empty (State);
   --# post Count (State) = Count (State~) - 1;
```

(continues on next page)

(continued from previous page)

```

   --  Procedure that swaps the first two elements in a stack.
   procedure Swap2;
   --# global in out State;
   --# pre  Count(State) >= 2;
   --# post Count(State) =  Count(State~) and
   --#      Stack_Entry (Count (State), State) =
   --#         Stack_Entry (Count (State) - 1, State~) and
   --#      Stack_Entry (Count (State) - 1, State) =
   --#         Stack_Entry (Count (State), State~);

   --  Initializes the Stack.
   procedure Initialize;
   --# global out State;
   --# post Is_Empty (State);
end Stack;
```

Body in SPARK 2005:

```
package body Stack
--# own State is My_Stack;
is
   Stack_Size : constant := 100;
   type    Pointer_Range is range 0 .. Stack_Size;
   subtype Index_Range   is Pointer_Range range 1..Stack_Size;
   type    Vector        is array(Index_Range) of Integer;

   type Stack_Type is record
      S : Vector;
      Pointer : Pointer_Range;
   end record;

   Initial_Stack : constant Stack_Type :=
      Stack_Type'(S       => Vector'(others => 0),
                  Pointer => 0);

   My_Stack : Stack_Type;

   procedure Push(X : in Integer)
   --# global in out My_Stack;
   --# pre My_Stack.Pointer < Stack_Size;
   is
   begin
      My_Stack.Pointer := My_Stack.Pointer + 1;
      My_Stack.S(My_Stack.Pointer) := X;
   end Push;

   procedure Pop (X : out Integer)
   --# global in out My_Stack;
   --# pre My_Stack.Pointer >= 1;
   is
   begin
```

(continues on next page)

(continued from previous page)

```
      X := My_Stack.S (My_Stack.Pointer);
      My_Stack.Pointer := My_Stack.Pointer - 1;
   end Pop;

   procedure Swap2
   --# global in out My_Stack;
   --# post My_Stack.Pointer = My_Stack~.Pointer;
   is
      Temp : Integer;
   begin
      Temp := My_Stack.S (1);
      My_Stack.S (1) := My_Stack.S (2);
      My_Stack.S (2) := Temp;
   end Swap2;

   procedure Initialize
   --# global out My_Stack;
   --# post My_Stack.Pointer = 0;
   is
      --# for Initial_Stack declare Rule;
   begin
      My_Stack := Initial_Stack;
   end Initialize;
end Stack;
```

Specification in SPARK 2014

```
package Stack
  with SPARK_Mode,
       Abstract_State    => State,
       Initializes       => State,
       Initial_Condition => Is_Empty
is
   --  In SPARK 2014 we can specify an initial condition for the
   --  elaboration of a package and so initialization may be done
   --  during the elaboration of the package Stack, rendering the need
   --  for an initialization procedure unnecessary.

   --  Abstract states do not have types in SPARK 2014 they can only
   --  be directly referenced in Global and Depends aspects.

   --  Proof functions are actual functions but they may have the
   --  convention Ghost meaning that they can only be called from
   --  assertion expressions, e.g., pre and postconditions
   function Max_Stack_Size return Natural
     with Ghost;

   -- Returns the number of elements on the stack
   function Count return Natural
     with Global     => (Input => State),
          Ghost;

```

(continues on next page)

(continued from previous page)

```
   --  Returns the Nth entry on the stack. Stack_Entry (Count) is the
   --  top of stack
   function Stack_Entry (N : Natural) return Integer
     with Global     => (Input => State),
          Pre        => N in 1 .. Count,
          Ghost;
   --  A body (refined) version of this function can (must) be
   --  provided in the body of the package.

   function Is_Empty return Boolean is (Count = 0)
     with Global     => State,
          Ghost;

   function Is_Full return Boolean is (Count = Max_Stack_Size)
     with Global     => State,
          Ghost;

   --  The precondition requires the stack is not full when a value,
   --  X, is pushed onto it. Functions with global items (Is_Full
   --  with global State in this case) can be called in an assertion
   --  expression such as the precondition here.  The postcondition
   --  indicates that the count of the stack will be incremented after
   --  a push and therefore the stack will be non-empty.  The item X
   --  is now the top of the stack.
   procedure Push (X : in Integer)
     with Global => (In_Out => State),
          Pre    => not Is_Full,
          Post   => Count = Count'Old + 1 and
                    Count <= Max_Stack_Size and
                    Stack_Entry (Count) = X;

   --  The precondition requires the stack is not empty when we pull a
   --  value from it. The postcondition indicates the stack count is
   --  decremented.
   procedure Pop (X : out Integer)
     with Global => (In_Out => State),
          Pre    => not Is_Empty,
          Post   => Count = Count'Old - 1;

   --  Procedure that swaps the top two elements in a stack.
   procedure Swap2
     with Global => (In_Out => State),
          Pre    => Count >= 2,
          Post   => Count = Count'Old and
                    Stack_Entry (Count) = Stack_Entry (Count - 1)'Old and
                    Stack_Entry (Count - 1) = Stack_Entry (Count)'Old;
end Stack;
```

Body in SPARK 2014:

```
package body Stack
  with SPARK_Mode,
```

(continues on next page)

(continued from previous page)

```
      Refined_State => (State => My_Stack)
is
   Stack_Size : constant := 100;
   type    Pointer_Range is range 0 .. Stack_Size;
   subtype Index_Range   is Pointer_Range range 1 .. Stack_Size;
   type    Vector        is array(Index_Range) of Integer;

   type Stack_Type is record
      S : Vector;
      Pointer : Pointer_Range;
   end record;

   Initial_Stack : constant Stack_Type :=
     Stack_Type'(S       => Vector'(others => 0),
                 Pointer => 0);
   My_Stack : Stack_Type;

   function Max_Stack_Size return Natural is (Stack_Size);

   function Count return Natural is (Natural (My_Stack.Pointer))
     with Refined_Global => My_Stack;

   function Stack_Entry (N : Natural) return Integer is
     (My_Stack.S (Index_Range (N)))
     with Refined_Global => My_Stack;


   procedure Push(X : in Integer)
     with Refined_Global => (In_Out => My_Stack)
   is
   begin
      My_Stack.Pointer := My_Stack.Pointer + 1;
      My_Stack.S(My_Stack.Pointer) := X;
   end Push;

   procedure Pop (X : out Integer)
     with Refined_Global => (In_Out => My_Stack)
   is
   begin
      X := My_Stack.S (My_Stack.Pointer);
      My_Stack.Pointer := My_Stack.Pointer - 1;
   end Pop;

   procedure Swap2
     with Refined_Global => (In_Out => My_Stack)
   is
      Temp : Integer;
   begin
      Temp := My_Stack.S (My_Stack.Pointer);
      My_Stack.S (My_Stack.Pointer) := My_Stack.S (My_Stack.Pointer - 1);
      My_Stack.S (My_Stack.Pointer - 1) := Temp;
   end Swap2;
```

(continues on next page)

(continued from previous page)

```
begin
   My_Stack := Initial_Stack;
end Stack;
```

A.5.4 Using an External Prover

One may wish to use an external prover such as Isabelle, with rules defining a ghost function written in the prover input language. This can be done in SPARK 2014 by denoting the ghost function as an Import in lieu of providing a body for it. Of course such ghost functions cannot be executed.

Specification in SPARK 2014 using an external prover:

```
package Stack_External_Prover
  with SPARK_Mode,
       Abstract_State    => State,
       Initializes       => State,
       Initial_Condition => Is_Empty
is
   --  A Ghost function may be an Import which means that no body is
   --  given in the SPARK 2014 code and the proof has to be discharged
   --  by an external prover. Of course, such functions are not
   --  executable.
   function Max_Stack_Size return Natural
     with Global   => null,
          Ghost,
          Import;

   --  Returns the number of elements on the stack
   function Count return Natural
     with Global   => (Input => State),
          Ghost,
          Import;

   --  Returns the Nth entry on the stack. Stack_Entry (Count) is the
   --  top of stack
   function Stack_Entry (N : Natural) return Integer
     with Global   => (Input => State),
          Ghost,
          Import;

   function Is_Empty return Boolean
     with Global   => State,
          Ghost,
          Import;

   function Is_Full return Boolean
     with Global   => State,
          Ghost,
          Import;

   procedure Push (X : in Integer)
```

(continues on next page)

(continued from previous page)

```
    with Global   => (In_Out => State),
         Always_Terminates,
         Pre      => not Is_Full,
         Post     => Count = Count'Old + 1 and Count <= Max_Stack_Size and
                        Stack_Entry (Count) = X;

   procedure Pop (X : out Integer)
     with Global   => (In_Out => State),
         Always_Terminates,
         Pre      => not Is_Empty,
         Post     => Count = Count'Old - 1;

   procedure Swap2
     with Global   => (In_Out => State),
         Always_Terminates,
         Pre      => Count >= 2,
         Post     => Count = Count'Old and
                    Stack_Entry (Count) = Stack_Entry (Count - 1)'Old and
                    Stack_Entry (Count - 1) = Stack_Entry (Count)'Old;
end Stack_External_Prover;
```

A.5.5 Quoting an Own Variable in a Contract

Sometimes it is necessary to reference an own variable (a state abstraction) in a contract. In SPARK 2005 this was achieved by declaring the own variable with a type, either concrete or abstract. As seen in *Proof types and proof functions*. Once the own variable has a type it can be used in a SPARK 2005 proof context.

A state abstraction in SPARK 2014 does not have a type. Instead, an Ada type to represent the abstract state is declared. A function which has the state abstraction as a global item is then declared which returns an object of the type. This function may have the same name as the state abstraction (the name is overloaded). References which appear to be the abstract state in an assertion expression are in fact calls to the overloaded function.

An example of this technique is given in the following example which is a version of the stack example given in *Proof types and proof functions* but with the post conditions extended to express the functional properties of the stack.

The extension requires the quoting of the own variable/state abstraction in the postcondition in order to state that the contents of the stack other than the top entries are not changed.

Specification in SPARK 2005:

```
package Stack_Functional_Spec
--# own State : Abstract_Stack;
is
   --  It is not possible to specify that the stack will be
   --  initialized to empty except by having an initialization
   --  subprogram called during program execution (as opposed to
   --  package elaboration).

   --  Proof functions to indicate whether or not the Stack is empty
   --  and whether or not it is full.
   --# type Abstract_Stack is abstract;

   --# function Max_Stack_Size return Natural;
```

(continues on next page)

(continued from previous page)

```

   --  Proof function to give the number of elements on the stack.
   --# function Count(Input : Abstract_Stack) return Natural;

   --  Proof function returns the Nth entry on the stack.
   --  Stack_Entry (Count (State)) is the top of stack
   --# function Stack_Entry (S : Abstract_Stack; N : Natural) return Integer;
   --# pre N in 1 .. Count (S);
   --  A refined version of this function cannot be written because
   --  the abstract view has a formal parameter of type Abstract_Stack
   --  whereas the refined view would not have this parameter but use
   --  a global. A user defined proof rule would be required to
   --  define this function. Alternatively, it could be written as an
   --  Ada function where the the global and formal parameter views
   --  would be available. However, the function would then be
   --  callable and generate implementation code.

   --# function Is_Empty(Input : Abstract_Stack) return Boolean;
   --# return Count (Input) = 0;

   --# function Is_Full(Input : Abstract_Stack) return Boolean;
   --# return Count (Input) = Max_Stack_Size;

   --  The precondition requires the stack is not full when a value, X,
   --  is pushed onto it.
   --  Functions with global items (Is_Full with global State in this case)
   --  can be called in an assertion expression such as the precondition here.
   --  The postcondition indicates that the count of the stack will be
   --  incremented after a push and therefore the stack will be non-empty.
   --  The item X is now the top of the stack and the contents of the rest of
   --  the stack are unchanged.
   procedure Push(X : in Integer);
   --# global in out State;
   --# pre  not Is_Full(State);
   --# post Count (State) = Count (State~) + 1 and
   --#      Count (State) <= Max_Stack_Size and
   --#      Stack_Entry (State, Count (State)) = X and
   --#      (for all I in Natural range 1 .. Count (State~) =>
   --#          (Stack_Entry (State, I) = Stack_Entry (State~, I)));

   --  The precondition requires the stack is not empty when we
   --  pull a value from it.
   --  The postcondition indicates that the X = the old top of stack,
   --  the stack count is decremented, and the contents of the stack excluding
   --  the old top of stack are unchanged.
   procedure Pop (X : out Integer);
   --# global in out State;
   --# pre not Is_Empty (State);
   --# post Count (State) = Count (State~) - 1 and
   --#      X = Stack_Entry (State~, Count (State~)) and
   --#      (for all I in Natural range 1 .. Count (State) =>
   --#          (Stack_Entry (State, I) = Stack_Entry (State~, I)));
```

(continues on next page)

(continued from previous page)

```

   --  The precondition requires that the stack has at least 2 entries
   --  (Count >= 2).
   --  The postcondition states that the top two elements of the stack are
   --  transposed but the remainder of the stack is unchanged.
   procedure Swap2;
   --# global in out State;
   --# pre  Count(State) >= 2;
   --# post Count(State) =  Count(State~) and
   --#      Stack_Entry (State, Count (State)) =
   --#         Stack_Entry (State~, Count (State) - 1) and
   --#      Stack_Entry (State, Count (State) - 1) =
   --#         Stack_Entry (State~, Count (State)) and
   --#      (for all I in Natural range 1 .. Count (State) =>
   --#          (Stack_Entry (State, I) = Stack_Entry (State~, I)));

   --  Initializes the Stack.
   procedure Initialize;
   --# global out State;
   --# post Is_Empty (State);
end Stack_Functional_Spec;
```

Body in SPARK 2005:

```
package body Stack_Functional_Spec
--# own State is My_Stack;
is
   Stack_Size : constant := 100;
   type    Pointer_Range is range 0 .. Stack_Size;
   subtype Index_Range   is Pointer_Range range 1..Stack_Size;
   type    Vector        is array(Index_Range) of Integer;

   type Stack_Type is
      record
         S : Vector;
         Pointer : Pointer_Range;
      end record;

   Initial_Stack : constant Stack_Type :=
      Stack_Type'(S       => Vector'(others => 0),
                  Pointer => 0);

   My_Stack : Stack_Type;

   procedure Push(X : in Integer)
   --# global in out My_Stack;
   --# pre My_Stack.Pointer < Stack_Size;
   is
   begin
      My_Stack.Pointer := My_Stack.Pointer + 1;
      My_Stack.S(My_Stack.Pointer) := X;
   end Push;
```

(continues on next page)

(continued from previous page)

```

   procedure Pop (X : out Integer)
   --# global in out My_Stack;
   --# pre My_Stack.Pointer >= 1;
   is
   begin
      X := My_Stack.S (My_Stack.Pointer);
      My_Stack.Pointer := My_Stack.Pointer - 1;
   end Pop;

   procedure Swap2
   --# global in out My_Stack;
   --# post My_Stack.Pointer = My_Stack~.Pointer;
   is
      Temp : Integer;
   begin
      Temp := My_Stack.S (1);
      My_Stack.S (1) := My_Stack.S (2);
      My_Stack.S (2) := Temp;
   end Swap2;

   procedure Initialize
   --# global out My_Stack;
   --# post My_Stack.Pointer = 0;
   is
      --# for Initial_Stack declare Rule;
   begin
      My_Stack := Initial_Stack;
   end Initialize;

end Stack_Functional_Spec;
```

Specification in SPARK 2014

```
pragma Unevaluated_Use_Of_Old(Allow);
package Stack_Functional_Spec
  with SPARK_Mode,
       Abstract_State    => State,
       Initializes       => State,
       Initial_Condition => Is_Empty
is
   --  Abstract states do not have types in SPARK 2014 but to provide
   --  functional specifications it is sometimes necessary to refer to
   --  the abstract state in an assertion expression such as a post
   --  condition. To do this in SPARK 2014 an Ada type declaration is
   --  required to represent the type of the abstract state, then a
   --  function applied to the abstract state (as a global) can be
   --  written which returns an object of the declared type.
   type Stack_Type is private;

   --  The Abstract_State name may be overloaded by the function which
   --  represents it in assertion expressions.
```

(continues on next page)

(continued from previous page)

```
   function State return Stack_Type
     with Global => State;

   function Max_Stack_Size return Natural
     with Ghost;

   --  Returns the number of elements on the stack
   --  A function may have a formal parameter (or return a value)
   --  of the abstract state.
   function Count (S : Stack_Type) return Natural
     with Ghost;

   -- Returns the Nth entry on the stack.
   -- Stack_Entry (S, Count (S)) is the top of stack
   function Stack_Entry (S : Stack_Type; N : Natural) return Integer
     with Pre      => N in 1 .. Count (S),
          Ghost;

   -- The ghost function Count can be called in the function
   -- expression because Is_Empty is also a ghost function.
   function Is_Empty return Boolean is (Count (State) = 0)
     with Global     => State,
          Ghost;

   function Is_Full return Boolean is (Count(State) = Max_Stack_Size)
     with Global     => State,
          Ghost;

   --  The precondition requires the stack is not full when a value, X,
   --  is pushed onto it.
   --  Functions with global items (Is_Full with global State in this case)
   --  can be called in an assertion expression such as the precondition here.
   --  The postcondition indicates that the count of the stack will be
   --  incremented after a push and therefore the stack will be non-empty.
   --  The item X is now the top of the stack and the contents of the rest of
   --  the stack are unchanged.
   procedure Push (X : in Integer)
     with Global => (In_Out => State),
          Pre    => not Is_Full,
          Post   => Count (State) = Count (State'Old) + 1 and
                    Count (State) <= Max_Stack_Size and
                    Stack_Entry (State, Count (State)) = X and
                    (for all I in 1 .. Count (State'Old) =>
                        Stack_Entry (State, I) = Stack_Entry (State'Old, I));

   --  The precondition requires the stack is not empty when we
   --  pull a value from it.
   --  The postcondition indicates that the X = the old top of stack,
   --  the stack count is decremented, and the contents of the stack excluding
   --  the old top of stack are unchanged.
   procedure Pop (X : out Integer)
     with Global => (In_Out => State),
```

(continues on next page)

(continued from previous page)

```
         Pre    => not Is_Empty,
         Post   => Count (State) = Count (State'Old) - 1 and
                   X = Stack_Entry (State'Old, Count (State'Old)) and
                  (for all I in 1 .. Count (State) =>
                      Stack_Entry (State, I) = Stack_Entry (State'Old, I));

   --  The precondition requires that the stack has at least 2 entries
   --  (Count >= 2).
   --  The postcondition states that the top two elements of the stack are
   --  transposed but the remainder of the stack is unchanged.
   procedure Swap2
     with Global => (In_Out => State),
          Pre    => Count (State) >= 2,
          Post   => Count(State) = Count (State'Old) and
                    Stack_Entry (State, Count (State)) =
                       Stack_Entry (State'Old, Count (State) - 1) and
                    Stack_Entry (State, Count (State) - 1) =
                       Stack_Entry (State'Old, Count (State)) and
                    (for all I in 1 .. Count (State) - 2 =>
                        Stack_Entry (State, I) = Stack_Entry (State'Old, I));

private
   -- The full type declarion used to represent the abstract state.
   Stack_Size : constant := 100;
   type    Pointer_Range is range 0 .. Stack_Size;
   subtype Index_Range   is Pointer_Range range 1 .. Stack_Size;
   type    Vector        is array(Index_Range) of Integer;

   type Stack_Type is record
      S : Vector;
      Pointer : Pointer_Range;
   end record;
end Stack_Functional_Spec;
```

Body in SPARK 2014:

```
package body Stack_Functional_Spec
  with SPARK_Mode,
       Refined_State => (State => My_Stack)
is
   Initial_Stack : constant Stack_Type :=
     Stack_Type'(S       => Vector'(others => 0),
                 Pointer => 0);

   --  In this example the type used to represent the state
   --  abstraction and the actual type used in the implementation are
   --  the same, but they need not be. For instance S and Pointer
   --  could have been declared as distinct objects rather than
   --  composed into a record. Where the type representing the
   --  abstract state and the implementation of that state are
   --  different the function representing the abstract state has to
   --  convert implementation representation into the abstract
```

(continues on next page)

(continued from previous page)

```
   --  representation. For instance, if S and Pointer were distinct
   --  objects the function State would have to return (S => S,
   --  Pointer => Pointer).
   My_Stack : Stack_Type;

   --  No conversion necessary as the abstract and implementation type
   --  is the same.
   function State return Stack_Type is (My_Stack)
     with Refined_Global => My_Stack;

   function Max_Stack_Size return Natural is (Stack_Size);

   function Count (S : Stack_Type) return Natural is (Natural (S.Pointer));

   function Stack_Entry (S : Stack_Type; N : Natural) return Integer is
     (S.S (Index_Range (N)));

   procedure Push(X : in Integer)
     with Refined_Global => (In_Out => My_Stack)
   is
   begin
      My_Stack.Pointer := My_Stack.Pointer + 1;
      My_Stack.S(My_Stack.Pointer) := X;
   end Push;

   procedure Pop (X : out Integer)
     with Refined_Global => (In_Out => My_Stack)
   is
   begin
      X := My_Stack.S (My_Stack.Pointer);
      My_Stack.Pointer := My_Stack.Pointer - 1;
   end Pop;

   procedure Swap2
     with Refined_Global => (In_Out => My_Stack)
   is
      Temp : Integer;
   begin
      Temp := My_Stack.S (My_Stack.Pointer);
      My_Stack.S (My_Stack.Pointer) := My_Stack.S (My_Stack.Pointer - 1);
      My_Stack.S (My_Stack.Pointer - 1) := Temp;
   end Swap2;
begin
   My_Stack := Initial_Stack;
end Stack_Functional_Spec;
```

A.5.6 Main_Program annotation

This annotation isn't needed. Currently any parameterless procedure declared at library-level is considered as a potential main program and analyzed as such.

A.6 Update Expressions

SPARK 2005 has update expressions for updating records and arrays. They can only be used in SPARK 2005 proof contexts.

The equivalent in SPARK 2014 is a delta aggregate. This can be used in any Ada expression.

Specification in SPARK 2005:

```
package Update_Examples
is
   type Rec is record
      X, Y : Integer;
   end record;

   type Index is range 1 ..3;

   type Arr is array (Index) of Integer;

   type Arr_2D is array (Index, Index) of Integer;

   type Nested_Rec is record
      A : Integer;
      B : Rec;
      C : Arr;
      D : Arr_2D;
   end record;

   type Nested_Arr is array (Index) of Nested_Rec;

   -- Simple record update
   procedure P1 (R : in out Rec);
   --# post R = R~ [X => 1];

   -- Simple 1D array update
   procedure P2 (A : in out Arr);
   --# post A = A~ [1 => 2];

   -- 2D array update
   procedure P3 (A2D : in out Arr_2D);
   --# post A2D = A2D~ [1, 1 => 1;
   --#                  2, 2 => 2;
   --#                  3, 3 => 3];

   -- Nested record update
   procedure P4 (NR : in out Nested_Rec);
   --# post NR = NR~ [A => 1;
   --#                B => NR~.B [X => 1];
```

(continues on next page)

(continued from previous page)

```
   --#                 C => NR~.C [1 => 5]];

   -- Nested array update
   procedure P5 (NA : in out Nested_Arr);
   --# post NA = NA~ [1 => NA~ (1) [A => 1;
   --#                                D => NA~ (1).D [2, 2 => 0]];
   --#                 2 => NA~ (2) [B => NA~ (2).B [X => 2]];
   --#                 3 => NA~ (3) [C => NA~ (3).C [1 => 5]]];
end Update_Examples;
```

Specification in SPARK 2014

```
package Update_Examples
  with SPARK_Mode
is
   type Rec is record
      X, Y : Integer;
   end record;

   type Arr is array (1 .. 3) of Integer;

   type Nested_Rec is record
      A : Integer;
      B : Rec;
      C : Arr;
   end record;

   type Nested_Arr is array (1 .. 3) of Nested_Rec;

   -- Simple record update
   procedure P1 (R : in out Rec)
     with Post => R = (R'Old with delta X => 1);
   -- this is equivalent to:
   --   R = (X => 1,
   --        Y => R'Old.Y)

   -- Simple 1D array update
   procedure P2 (A : in out Arr)
     with Post => A = (A'Old with delta 1 => 2);
   -- this is equivalent to:
   --   A = (1 => 2,
   --        2 => A'Old (2),
   --        3 => A'Old (3));

   -- Nested record update
   procedure P3 (NR : in out Nested_Rec)
     with Post => NR = (NR'Old with delta A => 1,
                                          B => (NR'Old.B with delta X => 1),
                                          C => (NR'Old.C with delta 1 => 5));
   -- this is equivalent to:
   --   NR = (A => 1,
   --         B.X => 1,
```

(continues on next page)

(continued from previous page)

```
   --         B.Y => NR'Old.B.Y,
   --         C (1) => 5,
   --         C (2) => NR'Old.C (2),
   --         C (3) => NR'Old.C (3))

   -- Nested array update
   procedure P4 (NA : in out Nested_Arr)
     with Post =>
       NA = (NA'Old with delta
               1 => (NA'Old (1) with delta A => 1),
               2 => (NA'Old (2) with delta
                       B => (NA'Old (2).B with delta X => 2)),
               3 => (NA'Old (3) with delta
                       C => (NA'Old (3).C with delta 1 => 5)));
   -- this is equivalent to:
   --   NA = (1 => (A => 1,
   --               B => NA'Old (1).B,
   --               C => NA'Old (1).C),
   --         2 => (B.X => 2,
   --               B.Y => NA'Old (2).B.Y,
   --               A => NA'Old (2).A,
   --               C => NA'Old (2).C),
   --         3 => (C => (1 => 5,
   --                     2 => NA'Old (3).C (2),
   --                     3 => NA'Old (3).C (3)),
   --               A => NA'Old (3).A,
   --               B => NA'Old (3).B));

end Update_Examples;
```

A.7 Value of Variable on Entry to a Loop

In SPARK 2005 the entry value of a for loop variable variable, X, can be referenced using the notation X%. This notation is required frequently when the variable is referenced in a proof context within the loop. Often it is needed to state that the value of X is not changed within the loop by stating X = X%. This notation is restricted to a variable which defines the lower or upper range of a for loop.

SPARK 2014 has a more general scheme whereby the loop entry value of any variable can be denoted within any sort of loop using the '*Loop_Entry* attribute. However, its main use is not for showing that the value of a for loop variable has not changed as the SPARK 2014 tools are able to determine this automatically. Rather it is used instead of ~ in loops because the attribute 'Old is only permitted in postconditions (including Contract_Cases).

Specification in SPARK 2005:

```
package Loop_Entry
is

  subtype ElementType is Natural range 0..1000;
  subtype IndexType is Positive range 1..100;
  type ArrayType is array (IndexType) of ElementType;

```

(continues on next page)

(continued from previous page)

```
  procedure Clear (A: in out ArrayType; L,U: in IndexType);
  --# derives A from A, L, U;
  --# post (for all N in IndexType range L..U => (A(N) = 0)) and
  --#      (for all N in IndexType => ((N<L or N>U) -> A(N) = A~(N)));

end Loop_Entry;
```

Body in SPARK 2005:

```
package body Loop_Entry
is

  procedure Clear (A: in out ArrayType; L,U: in IndexType)
  is
  begin
    for I in IndexType range L..U loop
      A(I) := 0;
      --# assert (for all N in IndexType range L..I => (A(N) = 0)) and
      --#        (for all N in IndexType => ((N<L or N>I) -> A(N) = A~(N))) and
      --#        U = U% and L <= I;
      -- Note U = U% is required to show that the vaule of U does not change
      -- within the loop.
    end loop;
  end Clear;

end Loop_Entry;
```

Specification in SPARK 2014:

```
pragma SPARK_Mode (On);
package Loop_Entry
is

  subtype ElementType is Natural range 0..1000;
  subtype IndexType is Positive range 1..100;
  type ArrayType is array (IndexType) of ElementType;

   procedure Clear (A: in out ArrayType; L,U: in IndexType)
     with Depends => (A => (A, L, U)),
          Post    => (for all N in L..U => A(N) = 0) and
                     (for all N in IndexType =>
                         (if N<L or N>U then A(N) = A'Old(N)));

end Loop_Entry;
```

Body in SPARK 2014:

```
pragma SPARK_Mode (On);
package body Loop_Entry
is

  procedure Clear (A: in out ArrayType; L,U: in IndexType)
  is
```

(continues on next page)

(continued from previous page)

```
  begin
    for I in IndexType range L..U loop
      A(I) := 0;
         pragma Loop_Invariant ((for all N in L..I => (A(N) = 0)) and
           (for all N in IndexType =>
              (if N < L or N > I then A(N) = A'Loop_Entry(N))));
      -- Note it is not necessary to show that the vaule of U does not change
      -- within the loop.
      -- However 'Loop_Entry must be used rather than 'Old.
    end loop;
  end Clear;

end Loop_Entry;
```

APPENDIX

B

GNU FREE DOCUMENTATION LICENSE

Version 1.3, 3 November 2008

Copyright (C) 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc.

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

B.1 PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document 'free' in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of 'copyleft', which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

B.2 APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The 'Document', below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as 'you'. You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A 'Modified Version' of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A 'Secondary Section' is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The ‘Invariant Sections’ are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The ‘Cover Texts’ are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A ‘Transparent’ copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not ‘Transparent’ is called ‘Opaque’.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The ‘Title Page’ means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, ‘Title Page’ means the text near the most prominent appearance of the work’s title, preceding the beginning of the body of the text.

The ‘publisher’ means any person or entity that distributes copies of the Document to the public.

A section ‘Entitled XYZ’ means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as ‘Acknowledgements’, ‘Dedications’, ‘Endorsements’, or ‘History’.) To ‘Preserve the Title’ of such a section when you modify the Document means that it remains a section ‘Entitled XYZ’ according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

B.3 VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

B.4 COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

B.5 MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
- State on the Title page the name of the publisher of the Modified Version, as the publisher.
- Preserve all the copyright notices of the Document.
- Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- Include an unaltered copy of this License.
- Preserve the section Entitled 'History', Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled 'History' in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

- Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the 'History' section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- For any section Entitled 'Acknowledgements' or 'Dedications', Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- Delete any section Entitled 'Endorsements'. Such a section may not be included in the Modified Version.
- Do not retitle any existing section to be Entitled 'Endorsements' or to conflict in title with any Invariant Section.
- Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled 'Endorsements', provided it contains nothing but endorsements of your Modified Version by various parties – for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

B.6 COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled 'History' in the various original documents, forming one section Entitled 'History'; likewise combine any sections Entitled 'Acknowledgements', and any sections Entitled 'Dedications'. You must delete all sections Entitled 'Endorsements'.

B.7 COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

B.8 AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an 'aggregate' if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

B.9 TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled 'Acknowledgements', 'Dedications', or 'History', the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

B.10 TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

B.11 FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License 'or any later version' applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

B.12 RELICENSING

'Massive Multiauthor Collaboration Site' (or 'MMC Site') means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A 'Massive Multiauthor Collaboration' (or 'MMC') contained in the site means any set of copyrightable works thus published on the MMC site.

'CC-BY-SA' means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

'Incorporate' means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is 'eligible for relicensing' if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

B.13 ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  YEAR  YOUR NAME.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.3
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.
A copy of the license is included in the section entitled 'GNU
Free Documentation License'.
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the 'with … Texts.' line with this:

```
with the Invariant Sections being LIST THEIR TITLES, with the
Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

INDEX